Lines of Battle: A Distributed Multiplayer Programming Game Environment

Bryan McClain, Jonathan Keesling, Aryan Banyal, Croix Gyurek
Department of Computer & Informational Science
Indiana University-Purdue University Indianapolis
Indianapolis, Indiana 46202, USA

***Abstract*—In this paper, we present a new multiplayer programming game environment entitled "Lines of Battle". We build our game environment using a central server architecture and implement the game engine logic. Clients connect to the game server using a public API interface and send messages to control the players in the game. In this way, clients serve as autonomous agents that run algorithms to decide which actions to perform in the game. We implement multiple client algorithms to serve as examples for future implementations. We also implement a visual interface that can be used to watch the agents playing a game, similar to an arena view in a sports match. This project serves as a helpful learning tool for teaching programming and algorithms in a classroom setting.**

***Index Terms*—multiplayer gaming; distributed environment; WebSocket API; educational games**

CONTENTS

I. INTRODUCTION AND PROBLEM STATEMENT

The principle of multiplayer programming games is not a new concept. In Screeps [1], players write JavaScript code to implement an AI algorithm for controlling units in the game world. The code is executed on the global game server and the game world is updated according to the actions taken by the AI agents. A similar game is Core Wars [2], where players write programs for a virtualized environment. All programs are loaded into the same memory space, and the goal is to inject malicious commands into other players' programs to try and crash their code.

One major limitation with existing multiplayer programming games is that players must use the programming languages and environments provided by the game developer. For instance, Screeps requires players to write their code in JavaScript and it must be run on the Screeps servers. Core Wars uses a custom assembly language called Redcode, which runs on a virtual machine to play

the game. Other multiplayer programming games have similar limitations.

For our project, we wanted to design a multiplayer programming game where players could write the algorithm in whatever language desired. The game server would provide a public API for clients to use when controlling in-game agents. We also wanted our game to be usable in an educational setting as a helpful tool for teaching programming concepts. Specifically, our project goals were as follows:

1) Design a multiplayer programming game environment with a public API
2) Players can implement an algorithm to control agents in the game using whatever programming language desired
3) The project should be usable in an educational setting to teach programming concepts

This paper will discuss our project implementation and how we accomplished our goals. In Section II, we discuss the design and implementation of our project components. In Section III, we perform experimental analysis on our implemented clients. Section IV will discuss the impact of this project and future extensions to the code. Finally, Section V will critique the project given the knowledge learned throughout CSCI 53700 Distributed Computing class.

II. Design and Implementation

To avoid ambiguity when discussing our implementation, we define the following terms:

- **Player** – Physical human being trying to play the game by implementing a client.
- **Client** – Program running on a computer that connects to the game server to play the game. Clients communicate with the game server to receive game state updates and move their agent in the game world. Multiple client instances may be running the same program and algorithm.
- **Agent** – Entity inside the game world that is controlled by the client. It performs in-game actions such as moving, attacking, and picking up items.
- **Viewer** – Similar to a client program, but it does not control an agent. It receives the same game state updates as a client.

In order to fulfill our previously discussed goals, we made the following design decisions when building our project:

- *Central game server* – Although it has less redundancy than a decentralized model, a central game server works better in a classroom setting. The instructor only has to run one program instead of trying to set up a distributed network of servers.
- *Simple yet flexible game engine* – We wanted the game logic to be simple enough that clients can be implemented using common computer science algorithms. However, we also wanted the flexibility to add more game elements for writing more advanced clients and algorithms, if desired.
- *WebSocket with JSON Messages* – We chose to use these technologies to allow for greater heterogeneity in the system. WebSockets [3] and JSON [4] are standards supported by most programming languages and are simple to work with for beginners. User authentication is handled using JSON Web Tokens [5], which we discuss later in the paper.

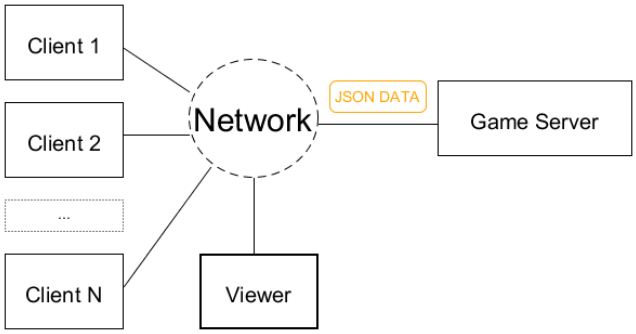

Fig. 1. High-level architecture of our multiplayer game environment

The overall architecture of our multiplayer game environment is shown in Figure 1. Clients connect to the central game server over a network connection and send JSON messages for communication. The system also supports viewers, which display the game state graphically so people can watch the autonomous agents play the game. Unlike clients, viewers are not allowed to control a player in the game.

In the following sub-sections, we discuss specific implementation details about different part of the project:

A. Gameplay

Since our focus was on the distributed computing aspect of the project, not the game design aspect, we decided our game should be a simple top-down battle arena. As previously mentioned, we wanted the game engine to be simple enough that clients could be implemented using common computer science algorithms. At the same time, we wanted the flexibility to add gameplay elements later, if needed.

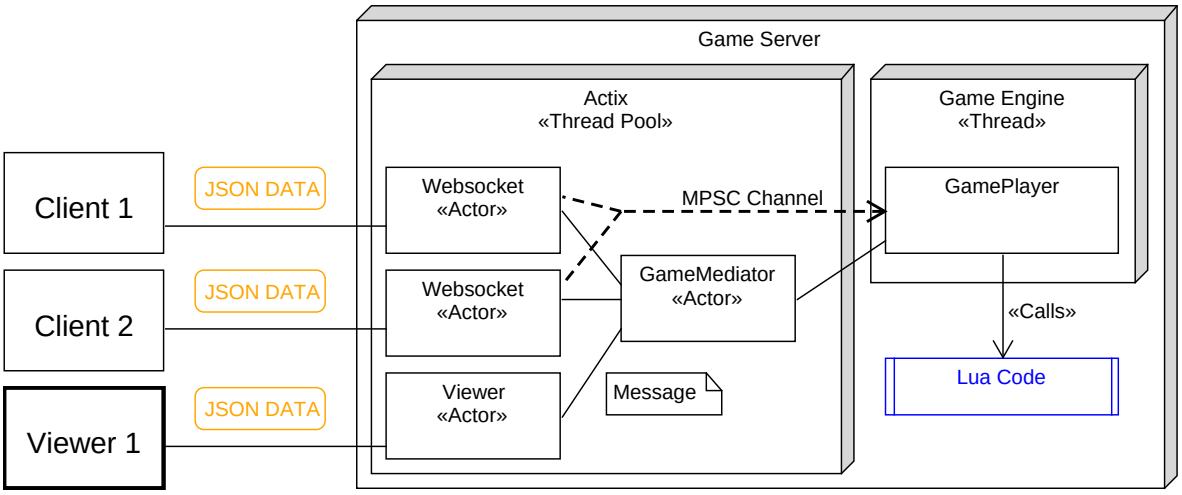

Fig. 2. Game server architecture

In Lines of Battle, the agents are "robots" that move around the square grid arena. The arena has walls which block player movement. Robots can move orthogonally, attack orthogonally, pick up weapons, or drop their currently held weapon. Robots can always attack if adjacent to each other. However, using a laser gun will attack from any distance away, assuming there is not a wall blocking the shot. A laser gun only has one round of ammo, meaning the robot will need to find a new weapon after using it. Attacking without a weapon deals 1 hit of damage, and attacking with a laser gun deals 2 hits of damage. Each robot has 3 hit points (HP), and if a robot has 0 HP remaining, they are eliminated from the game. The last robot left in the arena is the winner.

To avoid having infinitely-long games, the game server enforces a time limit on the maximum game length. If more than one robot is remaining after the time limit expires, all robots are declared winners and the game ends in a tie. It is also possible (albeit unlikely) that all players are eliminated from the arena, meaning there is no winner.

B. Game Server

The game server is responsible for implementing all of the logic required for handling WebSocket communication and running the game engine. In many ways, it functions as a "mini" distributed computing system. We chose to implement the game server using the Rust programming language [6], which performance similar to C or C++ but provides better abstractions for multithreaded applications. Rust uses a concept called *lifetimes* to track the scope of references (pointers), meaning that unsound multithreaded code becomes a compile-time error instead of a run-time bug. We used the Actix library [7] for managing WebSocket connections on a shared thread pool. Finally, we used the Lua programming language [8] for implementing our game engine logic. Lua is a fast, embeddable, interpreted language that is well-suited for game development and is used in many existing multiplayer game systems, such as Roblox [9]. The complete architecture of the game server is shown in Figure 2.

The Actix library is a Rust implementation of active objects, which we learned about in class in the paper by Bal et al [10]. In Actix, an active object is called an "actor", and actors communicate with one another by passing messages. Actors are run in a thread pool to handle asynchronous message communication. All WebSocket connections are managed by separate actors, which handle the low-level details of JSON message parsing and transmission. As a special case, viewer connections have a Viewer actor, which is functionally equivalent to a WebSocket actor but does not control an agent in the game. All WebSocket actors communicate with the GameMediator actor, who maintains the game server state, unique WebSocket connections, and registered players. The GameMediator will kick a connection if a client attempts to connect twice using the same JSON Web Token. The GameMediator also communicates with the game engine thread to start the next round of the game. When the game is running, each WebSocket actor is given a multiple-producers single-consumer (MPSC) channel to asynchronously push user actions to the game

engine thread. Any message broadcasts from the game engine go to the GameMediator, who sends the message to all connected actors to transmit on the WebSocket connection.

The game server also has a separate thread for running the game engine, which contains the actual logic of the game code. The GamePlayer object is responsible for initializing and calling the Lua code to run the game logic. It handles the low-level details of inner-process communication between the WebSocket actors and GameMediator.

The Lua code allows the game logic to be modified without having to recompile the game server. A valid Lua code file must export two global functions: *Init* and *Update*. The *Init* function is called once to start a new game and is given the player order for resolving actions. The *Update* function is called once per game "tick" and is given a map from player ID to their chosen action. This function performs the actions in the pre-determined player order and updates the game state accordingly. Both *Init* and *Update* return the game state to the caller. The game engine passes a Context object to both methods that are used for Lua communicating back, such as notifying the server that a player has been killed.

C. Game States

The game server has four main game states for modeling interaction with the clients. The transition diagram of all states is shown in Figure 3 and explained in more detail below.

1) Registration

When the game server program is first run, it starts in the *Registration* state. Clients can connect to the server and send a registration JSON message indicating they wish to play in the next game round. Alternatively, an already registered client can send a message to unregister from the game. Once the server receives enough registrations, it begins a countdown process before actually starting the game. New clients can still register during this time, but when the timer reaches 0 the server enters the *Initializing* state. If enough clients unregister during this time, the clock will stop and reset back to its default value. The server administrator can configure the minimum number of players required and countdown time before starting the game.

Fig. 3. Four main states in the game server

2) Initializing

Once the game server begins initialization, new clients are not allowed to register for the game. The current list of clients is now the official list playing in the game. The game server generates a new game world and spawns agents into the world. It also generates a random movement order for clients in the game. Once the game is ready to begin, the server broadcasts out a message to all clients with the initial state of the game world.

3) Game Running

When the game is running, clients need to send one action to the server every game "tick" to move their agent in the game. If a client does not send an action before the time-window expires, then their agent does not move that round. After the tick time window has passed, the game server will grab the current list of actions and update the game state using the generated player order. By default, a game tick occurs every real-world second, but this can be changed by the game server administrator.

After a player is eliminated from the game, they are no longer allowed to take any actions. However, they can stay connected to the server and receive broadcast update messages. When the game ends, the player can register for the next round.

4) Fatal Error

Since Lua is an interpreted programming language, it may run into run-time bugs or crashes with the code. If this happens, the code will be re-run up to five times, but after this the game server will enter a fatal error state. The game server will need to be manually restarted and the Lua code fixed if it enters this error state. We include a utility to test Lua code modifications to help prevent fatal errors with the server.

D. JSON Web Tokens

Our game server implementation uses JSON Web Tokens (JWT) [5] to handle authentication when connecting to the server. A JSON Web Token is an encoded string that consists of three dot-separated parts: the header, payload, and signature. The header is a base64-encoded [11] JSON object that specifies the signature algorithm. The payload is a base64-encoded JSON object

that contains the token claims, which are used for identity and authorization. Finally, the signature is a base64-encoded value output from the signing algorithm. In our implementation, JWTs are signed using the HS256 algorithm [5], which uses *HMAC_SHA256* [12] to compute the signature using a secret value known only to the server. An example JWT is shown in Figure 4.

When connecting to the game server, clients and viewers must include the JWT string as part of the WebSocket protocols header. The JWT includes claims for when the token is valid (start and end Unix timestamp), the unique player UUID, and the player alias in the game. Our implementation provides a utility program for the game server administrator to generate JWTs for players and viewers.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Fig. 4. Example JSON Web Token signed using the HS256 algorithm [5]. The three dot-separated colors represent the different base64-encoded parts of the token (header, payload, signature).

E. Visual Interface

Lines of Battle comes with a visual interface built using HTML, CSS, and JavaScript. This interface adds an easy way for students to take part in the action and watch their clients battle. It provides information on the status of each player, what actions each player has taken, and displays the entire arena with each turn. Figure 5 shows an example of the visual interface during an active game.

While it is entirely possible to play Lines of Battle without the visual interface, we feel there are three main reasons it is important to include one.

1) A visual interface is approachable.

At its core, Lines of Battle is a tool made to help teach programming. It provides a way for students to learn about and interact with programming concepts they may not yet be familiar with. This visual interface makes the learning process easier to approach, and lets students spend less time trying to understand what is happening during a game. By removing any obstacles to understanding the flow and state of a game, players can focus on the important task of designing and testing the best possible algorithm. Utilizing an official GUI means students can be confident that all other players in the game are seeing the same results.

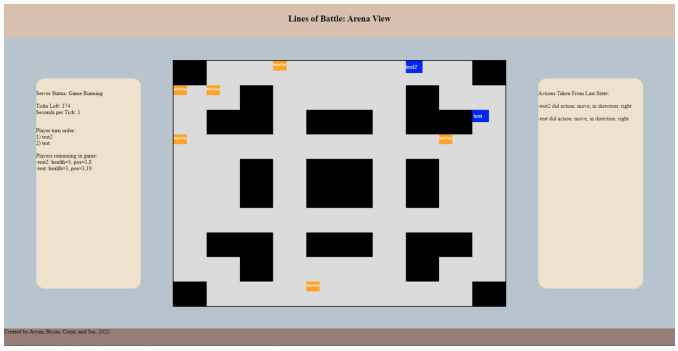

Fig. 5. Screenshot of the visual interface

2) Visuals provide fast feedback.

Individual games in Lines of Battle do not take a long time to complete. This short game time means it is likely that players may want to make changes to their clients and client algorithms between rounds. Watching a visual representation of the battle allows players to quickly discern where changes need to be made, and where their client may be making the wrong strategic decision. This helps players to develop a sound understanding of the game and agent actions compared to the traditional Print statement debugging that is common with on the fly programming.

3) The visual interface can grow with the game server.

The visual interface code provides another starting point for students that want to customize or expand Lines of Battle. A set of simple JavaScript function calls are responsible for handling the server connection and rendering the game arena every turn by utilizing an HTML canvas object. This means there is no need for students to source or resize custom graphics for anything they wish to add, and viewer functionality can easily be modified to support custom weapons and items, player actions, or the addition of non-player characters (NPCs).

F. Client Algorithms

We implemented four different client algorithms to serve as examples for interacting with the game server. To show the inherent heterogeneity of our system, we used different programming languages to implement our algorithms. Algorithms 1 and 2 were both implemented in TypeScript [13]. Algorithm 3 was originally implemented in Python [14] then ported to TypeScript later. Finally, Algorithm 4 was implemented in Java [15]. Our four algorithms are detailed in the sub-sections below:

1) Algorithm 1: Random Movement

This is a simple, naïve algorithm that randomly picks either move, attack, or drop weapon. For the move and attack actions, the algorithm picks a random direction for the action. This algorithm is meant to serve as a control when comparing the other algorithms in Section III.

2) Algorithm 2: A with Avoidance*

This algorithm is significantly smarter than random movement by attempting to find weapons and using them to attack robots. It also implements logic to dodge robots that have a weapon if they are in the line-of-sight to fire. The pseudocode for this algorithm is shown below:

Algorithm 2 A* with Avoidance

```
if directly next to another robot then
    attack that robot
end if
if robot with a weapon in line-of-sight then
    move orthogonally out of the way
end if
if robot does not have a weapon then
    find nearest weapon using A* algorithm
    move towards that weapon
else
    find nearest robot using A* algorithm
    move towards that robot
end if
```

3) Algorithm 3: A No Melee Attack*

This is essentially the same algorithm as the *A* with Avoidance*, but when the robot is directly next to another robot and does not posses a weapon, it will try to move out of the way instead of attacking the other robot. What this means is that this robot will not be spending time to do damage by Melee attacks but instead prioritizes finding a weapon before attacking other robots.

4) Algorithm 4: Hide in Corner

This algorithm attempts to utilize the map layout to gain a defensive advantage. It locates the nearest corner of walls using a breadth-first search algorithm, then moves into the corner. The robot stays in this location for the duration of the match and attacks any robots that move next to it.

III. Results and Discussion

We performed experimental analysis of our four client algorithms to show the ability of our multiplayer server to run the game simulation. We simulated 150 game rounds using the IUPUI RRPC computers and ran groups of six simulations in parallel (one for each RRPC computer) to increase the overall throughput. We ran one instance of each algorithm for every simulation, meaning that all simulations have four agents in the game world. The client instances were run outside the IUPUI network to show that our system correctly functions as a networked distributed environment.

In our simulations, we used a 10×10 grid with a fixed layout (the layout is shown in Figure 5). We spawned the four agents into random positions on the grid and also spawned 12 random laser guns in the grid. We ran each simulation with a maximum duration of 60 game ticks. We decided to use a delay of 1 second for each game tick (the default) since we did not run into any problems with network delay causing player actions to arrive late. Finally, we used a custom viewer program to aggregate the results from the groups of parallel simulations. The results of our simulations are shown in Figure 6, and a histogram of game lengths is shown in Figure 7.

Algorithm	Wins	Eliminated	Ties
Random Movement	2	131	17
A* with Avoidance	106	9	35
A* No Melee Attack	1	120	29
Hide in Corner	0	119	31

Fig. 6. Results of the algorithms

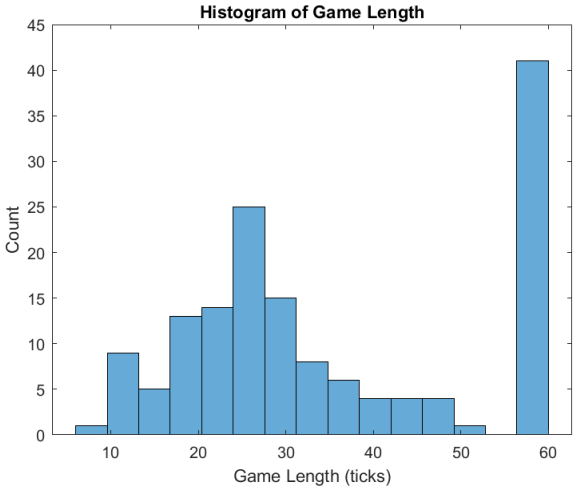

Fig. 7. Histogram of Game Length

Looking at our results, it is evident that the *A* with Avoidance* algorithm had the best overall approach for winning the game. Intuitively, this makes sense when looking at the algorithm code. This algorithm prioritizes attacking other robots above everything else, including finding a weapon. In a 10×10 grid, this algorithm has

a high chance of attacking the other robots before they have a chance to get a more powerful weapon. However, such an approach may not work well in a larger grid where other robots may find a more powerful weapon first.

Both the *A* No Melee Attack* and *Hide in Corner* algorithms performed poorly for the opposite reason that the *A* with Avoidance* performed well: they prioritize defense over offense. On a smaller grid like our 10×10 grid, the robots are in close proximity to one another, so they have nowhere to hide and wait to attack others. This is especially true with the *Hide in Corner* algorithm. Although it attacks adjacent squares, it cannot defend against laser guns which shoot from any distance away.

Interestingly enough, the *Random Movement* algorithm did end up winning 2 games. However, this is due to pure blind luck and not a superior strategy of the algorithm. It was eliminated in the overwhelming majority of games even when compared to the *Hide in Corner* algorithm. As previously mentioned, we intended this algorithm to serve as a control instead of a strong winning strategy.

When looking at the individual match results, the *A* with Avoidance* and *A* No Melee Attack* often tied with one another. Since both algorithms use the same pathfinding implementation, the algorithms would often be in conflict with one another when trying to attack each other using a laser gun. They would both move into each others' path to attack, then move out of the way to avoid being hit by the laser gun. This caused an infinite loop of moving back and forth, broken only by the game time expiring. This effect can be seen in the histogram of game lengths, where a significant number of games end with a tie after 60 game ticks. A smarter algorithm might break the infinite loop based on the player turn order, but we did not consider this approach in our implementations.

Apart from the performance of the individual algorithms, our results show that our multiplayer environment allows for many different client algorithms and implementations. Although one algorithm may perform best in certain circumstances, our implementation can flexibly handle any variety of client implementations across heterogeneous systems. Therefore, the use of a standard public API allows for variety when using our multiplayer game environment in a classroom setting.

IV. Conclusion and Future Work

In this paper, we have discussed our implementation of a distributed multiplayer programming game environment. Our design and implementation is optimized for use in the classroom. The programming languages and technologies we selected ensures our implementation is performant and scalable. We also designed the game logic to be flexible by using Lua code, which can easily be modified to support additional gameplay elements. The visual interface and provided example clients lowers the entry bar for introducing our implementation into a classroom setting.

We see this project as a good starting point for future work in distributed multiplayer programming games. Some ways to extend the project include:

- *Making the game server itself distributed* – There are many techniques learned in distributed computing class that would simplify this problem.
- *Supporting different multiplayer games* – The Lua game logic is very decoupled from the game server, meaning we could easily support different genres of multiplayer games. However, in the current implementation, the Rust code must define the JSON actions that players can take in the game. A future extension could allow custom data types to be defined using configuration files without having to modify the game server code.
- *User accounts for authentication* – This would allow players to register an account and get the JSON Web Token without having the server administrator manually generate the tokens for all players.
- *Selecting an available server* – It may be desirable to run multiple server instances, then have a mechanism for players to select any available server.

Even without these features, we see this project as a strong contribution to teaching programming in the classroom through the use of multiplayer gaming.

V. Project Critique

In this section, we critique our project based on what we learned in CSCI 53700 Distributed Computing class. Overall, we feel that we accomplish our project goals as we outlined at the beginning of the semester.

A. What We Would Keep the Same

First, when it comes to our mechanism for remote message passing, we would still choose to use WebSockets with JSON despite the other techniques learned in class. We wanted our project to be beginner-friendly so it could be used to teach programming in a classroom setting, but also allow for students to use whatever programming language desired. This naturally excludes Java RMI. Although Java RMI is beginner friendly, it requires

students to implement their logic in Java. We also would not have used rpcgen. Although it supports multiple programming languages, the technology is outdated and would not be beginner-friendly. Even writing the simple client and server in assignment 3 had many pitfalls when managing memory for relatively simple abstract data types. Raw sockets are another option, but they are not beginner-friendly in every language. They were easy for us to work with in Java, but not every language has the same abstractions for working with sockets. Additionally, some execution environments, such as JavaScript code in a web browser, do not support TCP/UDP socket connections for security reasons. Finally, we would not have used shared variables since this would be challenging for interoperability with different programming languages. Overall, WebSockets with JSON are still the best option for a beginner-friendly two-way communication with the server. The fact that JSON is a text-based format makes it easy to debug and explain in a classroom setting.

Next, we would still use threads and active objects for our parallelism in the game server. In class, we discussed many different units for parallelism, from processes to individual expressions. Using active objects is a good compromise between complexity and fine-grained control. The logic in the game server is too complicated for just statement-based or expression-based parallelism. Also, the ability for active objects to pass messages greatly simplified the WebSocket implementation for handling multiple clients.

We would also not have changed our message passing techniques for our two-way communication. We chose to implement asynchronous communication so the algorithms can run in the background while the server is processing. Using synchronous messaging would have greatly limited our parallelism for both the server and client implementations. Although we discussed how synchronous error handling may be easier, we feel that this would not improve our implementation.

Finally, we would use the same techniques for inner-process communication (IPC) between the actors and the game engine on the game server. As previously mentioned, we chose to use a multiple-producers single-consumer (MPSC) channel for pushing player actions to the game engine. We also used a bi-directional channel for the GameMediator to communicate with the game engine. In class, we discussed other techniques for IPC, such as atomic variables and sockets. However, our use of channels provides reliable delivery and ordering of messages, which still works the best for our game server implementation.

B. What We Would Have Changed

There are three main things we would have changed with our project implementation. First, when it comes to the game server, we would have preferred to designed a distributed environment with the option to run multiple game server instances. We mainly decided to use a single central server to simplify use in the classroom, but we were also unsure of our ability to implement a distributed system. However, after taking this class, we discussed a variety of techniques for synchronizing a shared state across multiple machines. This includes things such as using Lamport's Logical Clock [16] for determining the event order, various inner-process synchronization primitives, and message passing techniques. Using this knowledge, we could have implemented a system for ensuring event ordering across devices based on multicast communication between the servers. The server would still work with only one instance running to simplify classroom use, but could also scale to synchronize across multiple instances. The use of multiple distributed servers is something we listed under future work for the project.

Second, we would have focused on better techniques for dealing with network latency. In our lectures on networking technologies, we talked about ensuring a good quality-of-experience with remote communication. WebSockets have mechanisms for sending a heartbeat (ping-pong), which we could have used for measuring the communication round-trip time (RTT). Then, we could have automatically set the seconds per game tick based on the needs of the connections instead of the server administrator picking an arbitrary time (like the default time of 1 second per tick). This would simplify deployment in remote classrooms where students may be located in geographically different parts of the world.

Finally, we would have implemented a better security mechanism for the game server. We used JSON Web Tokens for their simplicity: server administrators (class instructor) can generate the token once and players are responsible for keeping the token secure. However, from Scott Orr's lectures, we learned different approaches for handling user authentication and authorization. A better approach might be for the server to have some form of login credentials, or use an external server for user authentication. We would like to have explored these options further based on what we learned in class.

REFERENCES

[1] "Screeps: MMO sandbox game for programmers," https://screeps.com/, accessed in December 2022.

[2] “Core Wars,” https://www.corewars.org/, accessed in December 2022.
[3] T. Fette, “The websocket protocol,” Internet Requests for Comments, RFC Editor, RFC 6455, 12 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6455.txt
[4] T. Bray, “The javascript object notation (json) data interchange format,” Internet Requests for Comments, RFC Editor, RFC 8259, 12 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8259.txt
[5] N. Sakimura, “Json web token (jwt),” Internet Requests for Comments, RFC Editor, RFC 7519, 5 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7519.txt
[6] N. D. Matsakis and F. S. Klock II, “The Rust language,” in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.
[7] “Actix,” https://actix.rs/, accessed in December 2022.
[8] R. Ierusalimschy, *Programming in Lua*. Roberto Ierusalimschy, 2006.
[9] “Roblox,” https://www.roblox.com/, accessed in December 2022.
[10] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, “Programming languages for distributed computing systems,” *ACM Comput. Surv.*, vol. 21, no. 3, p. 261–322, sep 1989. [Online]. Available: https://doi.org/10.1145/72551.72552
[11] S. Josefsson, “The base16, base32, and base64 data encodings,” Internet Requests for Comments, RFC Editor, RFC 3548, 7 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3548.txt
[12] U. D. of Commerce, N. I. of Standards, and Technology, *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2012.
[13] G. Bierman, M. Abadi, and M. Torgersen, “Understanding typescript,” in *European Conference on Object-Oriented Programming*. Springer, 2014, pp. 257–281.
[14] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.
[15] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.
[16] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, p. 558–565, jul 1978. [Online]. Available: https://doi.org/10.1145/359545.359563