Distributed LazIR Tag

Zihao Ding, Varun Desai, Pino Cholsaipant, and Olayinka Adekola
School of Enginnering
Stanford University
Email: {zihding, vdesai, pschol, oadekola}@stanford.edu

***Abstract*—We implemented a distributed laser tag game system using Raspberry Pi Zero W single-board computers. Each Raspberry Pi acts as both a server and a client, facilitating communication among nodes via the gRPC protocol. Players use IR remotes to send signals to IR receivers on the boards, earning and losing points accordingly. We monitor signal reception, identify users, maintain a distributed log of actions (such as who shot whom), and ensure consensus across all nodes in a decentralized manner using the Raft consensus protocol. The decentralized nature of the system allows the majority of players to continue playing even if a few nodes fail. Raft's inherent heartbeat monitoring helps detect node failures, and disconnected nodes can reintegrate into the game with minimal effort. Additionally, we analyze how increasing the number of players (and thus the number of nodes requiring consensus) affects the latency of committing various game events to the distributed log via a simulation. Our project demonstrates a smooth and engaging multiplayer distributed game of laser tag, leveraging a robust consensus protocol on cost-effective embedded hardware.**

1. Introduction

Traditional centralized gaming servers encounter significant limitations, including single points of failure and scalability challenges. In response, we are developing a multiplayer game leveraging cost-effective embedded hardware, specifically the Raspberry Pi Zero W, to create a decentralized and robust game system. Our project aims to overcome the traditional limitations of centralized gaming servers by implementing a distributed laser tag game system using Raspberry Pi Zero W single-board computers.

In our system, each node functions as both a server and a client, facilitating peer-to-peer communication and collaboration. By employing the gRPC protocol, we enable efficient and effective communication between nodes. This decentralized approach ensures that no single point of failure exists, enhancing the system's robustness and scalability. The use of Raspberry Pi Zero W devices also ensures cost-effectiveness without compromising performance.

By leveraging the capabilities of modern embedded and hardware and decentralized concensus protocols, we aim to create a more resilient and scalable multiplayer gaming experience, addressing the challenges that have long plagued centralized gaming infrastructures.

Players interact with the game using IR remotes, sending signals to IR receivers on the boards to earn or lose points. A critical aspect of our system is the maintenance of a distributed log that records actions, such as which player shot whom, and ensures consensus across all nodes. We employ the Raft consensus protocol to achieve this, as it is well-suited for the game's requirements of high availability and consistency. The decentralized architecture of our game ensures that gameplay can continue uninterrupted even if nodes fail, thanks to Raft's heartbeat monitoring and efficient node reintegration capabilities.

Analyzing the impact of an increasing number of players on the latency of event commitments provides insights into the scalability and performance of our system. This project not only showcases the feasibility of a distributed multiplayer game on embedded hardware but also highlights the potential for robust and engaging decentralized (low-cost) gaming experiences.

2. Design

System Architecture - The core architecture of our distributed laser tag game system is based on a network of Raspberry Pi Zero W single-board computers. Each Raspberry Pi serves a dual role as both a server and a client, facilitating peer-to-peer communication and distributed processing.

Communication Protocol - We utilize the gRPC protocol for communication between the nodes. gRPC is chosen for its efficiency, scalability, and support for various programming languages, making it ideal for real-time game communication.

Consensus Mechanism - To ensure consistent state across all nodes, we employ the Raft consensus protocol. Raft is selected due to its simplicity and effectiveness in maintaining distributed logs and achieving consensus. It is particularly suitable for our requirements, providing robust failure detection and recovery through heartbeat monitoring and leader election processes.

Distributed Log - A critical component of our system is the distributed log that records game events such as hits and score updates. This log is maintained across all nodes to ensure consistency and reliability. Each event is committed

to the log only after achieving consensus among the majority of nodes.

Fault Tolerance and Recovery - Our design emphasizes fault tolerance, allowing the game to continue even if some nodes fail. Raft's heartbeat mechanism detects node failures promptly, and the system can seamlessly reintegrate disconnected nodes once they are back online. This ensures minimal disruption to gameplay.

IR Remote and Receiver Setup - Each player is equipped with an IR remote that sends signals to IR receivers attached to the Raspberry Pi boards. These signals are interpreted to determine hits, points scored, and points lost.

The overall design of the system is also shown in Fig 1.

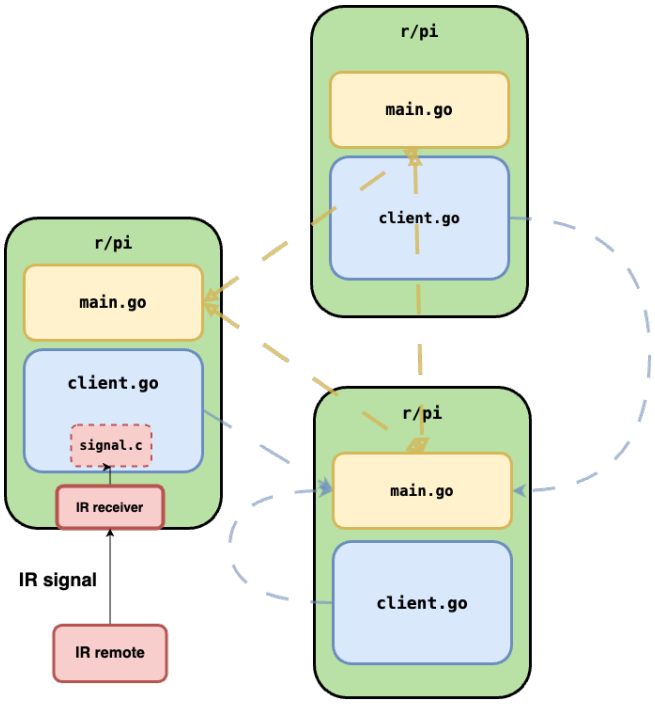

Figure 1: Overall infrastructure of the system. The r/pi nodes are shown in green, each of which is connected to an IR receiver which is programmer to recive signals from an IR remote with known key signatures and maps each signal received to a player in the game. Each r/pi nodes runs a server (`main.go`) and a client (`client.go`) process, both of which are crucial to the proper functioning of the system. The server processes run a consensus algorithm (`raft` in our case) while the client process is responsible for letting the cluster know of the events in the game in real time.

3. Implementation

3.1. Raft

Due to the lack of a centralized gaming server to keep records and interact with clients, a consensus algorithm is required to maintain a distributed and synchronized log of game events across all the nodes. Among all choices, Raft is chosen, mainly because it uses a stronger form of leadership than other consensus algorithms [1]. In other words, log entries only flow from the leader to other servers, which simplifies the management of replicated logs. In addition, Raft is equivalent to Paxos in fault tolerance and performance [2]. Its mechanism for leader election and log replication ensures that each node in the network maintains an up-to-date and synchronized record. The heartbeat mechanism enables nodes to monitor the status of their peers, swiftly detecting failures and initiating necessary corrective actions. Moreover, Raft's efficient node reintegration capabilities prove indispensable in the event of transient failures or network partitions, allowing a node to seamlessly recover after re-connection without disrupting the game-play continuity.

In this project, we use the implementation of Raft from HashiCorp [3]. Its biggest advantage is the support for customizable FSM application. As long as the application has implementations of corresponding functions defined in the interface, developers have great flexibility in making use of the replicated log. Essentially what we develop is a key-value store to record players' life points.

3.2. gRPC

In our implementation, the integration of gRPC plays a pivotal role in enabling efficient peer-to-peer communication. gRPC is a high-performance RPC (Remote Procedure Call) framework developed by Google. Because both the client and the Raft node processes are running in parallel on each Pi node, and Raft allows log entries to only flow from the leader to other servers, instead of just communicating with the Raft node process on the same Pi node, the client process needs to rapidly and accurately reach out to the current leader node to submit the log appending request. As a lightweight and efficient communication protocol, gRPC facilitates this kind of real-time updates and synchronization of game states. The implementation can be divided into several steps: 1. Implement a Transport for Raft over gRPC. One benefit of this is that gRPC is easy to multiplex over a single port. 2. Based on client-side service health checks, implement the feature of connecting to all Raft nodes, but only send RPCs to the leader. 3. Implement other auxiliary features, such as a frontend game monitor over gRPC, since multiple services can be registered to a single gRPC server, and multiple clients can trigger RPC calls in parallel.

3.3. IR Transmitter and Receiver

In this section, we detail the implementation of the IR transmitter and receiver system for our distributed laser tag game. The code leverages the WiringPi library for GPIO control on the Raspberry Pi, handling the reception and decoding of IR signals from remotes. The implementation involves reading signals from an IR receiver connected to a Raspberry Pi, processing the signal to extract meaningful

data, and converting these signals to identify specific actions (e.g., which player was hit).

The IR transmitter and receiver system is crucial for capturing and processing player actions. By leveraging the WiringPi library for GPIO control on the Raspberry Pi we detect and decoding of IR signals sent by players' remotes.

IR remotes transmit data by encoding bits as high signals of varying durations, with 0 bits and 1 bits represented by different lengths of the high signal. These bits are separated by fixed low signals. A unique integer corresponding to each button press on the remote is formed by concatenating these bits, akin to the UART protocol. Each command is prefixed with a distinctive header signal, significantly longer than a single bit, and concludes with an extended low signal to mark the end of the transmission.

To decode these signals, we reverse engineered the values for each button by following a structured approach. The process begins by spinning in a loop until the IR receiver detects a low signal, indicating the start of a transmission. If a header is detected (characterized by a long low signal followed by a long high signal), the system proceeds to read the subsequent bits until a timeout occurs, set longer than any valid signal duration (e.g., 20,000 microseconds).

The `read_while_eq` function monitors the IR receiver pin, recording the duration for which the pin's value remains constant. This timing data is essential for interpreting the incoming signal. The `get_readings` function collects these timings, storing them in an array of structs. Each struct captures the duration in microseconds `usec` and the value `v`, alternating between high and low states.

To ensure that the captured signal is valid, the `is_header` function checks if the initial readings match the expected header signal, which typically marks the start of a legitimate transmission. Once validated, the `convert`function processes the timings, converting them into a single value representing the IR signal. This function skips the header and decodes the signal bits based on the duration of each high and low state. Specifically, a 0 bit is interpreted as a high signal lasting 600 microseconds followed by a low signal of the same duration, while a 1 bit corresponds to a high signal of 1600 microseconds followed by a 600-microsecond low signal. Then the `key_to_str` function maps the decoded signal to a specific player identifier, such as "pi1" or "pi2," facilitating the identification of which player sent the signal.

The main function initializes the WiringPi library, sets up the GPIO pin for input with a pull-up resistor, and enters a loop to continuously read and process IR signals. It ensures real-time interaction by converting the signals and checking for known player identifiers. A debounce delay is included to prevent processing the same signal multiple times, maintaining the accuracy and responsiveness of the game.

3.4. Network

To ensure reliable communication between the Raspberry Pis in our distributed laser tag game, we connect them to a dedicated router that assigns static IP addresses. This setup addresses the limitations of using the Pis on Stanford internet (no browser interface, dynamic IP addressing), which is unreliable for maintaining connections while the Pis are in different campus buildings. Our Pis operate on the 2.4GHz band, which provides adequate range and penetration for indoor use.

Router Configuration - We use a standard wireless router to create a local network for the Raspberry Pis. This router is configured to assign static IP addresses to each Pi, ensuring consistent and predictable network communication. By turning off the Wi-Fi repeater function, we prevent the router from trying to connect to external networks, which is unnecessary for our purposes and could introduce instability. This configuration also enhances the portability of the system, allowing us to set up and operate the game in various locations without relying on external internet access.

3.5. Integration

The integration of Raft, gRPC, and Raspberry Pi Zero W boards was crucial for the functionality of our distributed laser tag game system. To streamline the startup process, we created a script that initializes the Raft nodes and starts the client processes on each Raspberry Pi. This script sets up the Raft consensus protocol, initializes the gRPC servers and clients, and launches the game processes, ensuring that all nodes are synchronized and ready to handle player interactions. This automation is important for maintaining and re-implementing the game environment, allowing players to start and enjoy the game with minimal setup time.

3.6. Frontend

In order to facilitate easy monitoring of the game, we also implement a simple front-end which displays the current status of the game in a human-friendly manner. The script simply repeatedly makes requests to the raft cluster about the current state of the k-v store, and the displays the result on the GUI, which can be seen in Fig 2.

3.7. Obstacles

Implementing the game posed several challenges. One significant obstacle was the need to cross-compile the Raft and gRPC code for the Pi Zero W's Broadcom BCM2835 SoC, which features a single-core 32-bit ARM11 CPU clocked at 1 GHz. This required us to use a specific Go distribution that supports the ARMv6 architecture. Ensuring compatibility between Go and C was another challenge, as we had to call C functions from Go to handle the low-level operations.

Reverse engineering the IR signals was particularly complex. Each IR remote button sends a unique integer value formed by concatenating bits, with 0 and 1 bits distinguished by different durations of the IR signal. We needed to capture these signals, identify the header, and decode the bits to

Figure 2: The front-end for the game, that facilitates probing the current status of all the players.

reconstruct the original command. Getting the direct timing for these reading proved to be the most difficult part on the hardware side.

4. Evaluation

We evaluate our system's performance and scalability by measuring the latency of an RPC call from a client to a cluster of servers running Raft in various setups, as shown in Fig 3 and 4. In particular, we launch a number of client processes submitting overlapping key/value reads and writes to the cluster concurrently, and then record the time it takes for the system to return to each process the correctly updated state. We adjust the total number of nodes in the Raft cluster and the number of players (clients) and record the average latency over 20 iterations of each experiment. Additionally, we repeat the procedure with a non-zero probability of node failure to understand how Raft's fault tolerance procedures impact performance.

We run the simulation on a machine with an Apple M1 ARM CPU with 8 cores (4 performance, 4 efficiency) running at 3.2 GHz, with 16 GB LPDDR4X unified memory [4].

To simulate node failures, we run a script that, with probability $p = 0.1$, kills and restarts processes within the Raft cluster every $t_0 = 0.1$ seconds. Since the average latency measured without node failures ranges from 20 to 40 ms, we expect this node failure script to trigger several times for each node across 20 iterations of the simulation, as this script is ran for each node separately and independently. We ensure that more than half of the nodes in the cluster are always operational, thereby recreating scenarios where nodes fail, triggering re-elections, but where the entire system does not collapse. This approach allows us to test our system under stressful conditions.

We see that the latency generally increases as the number of Raft nodes increases. This is not surprising: as there

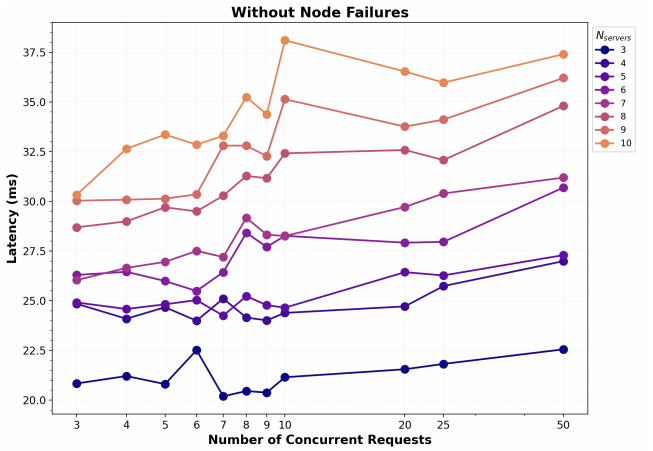

Figure 3: Without node failures

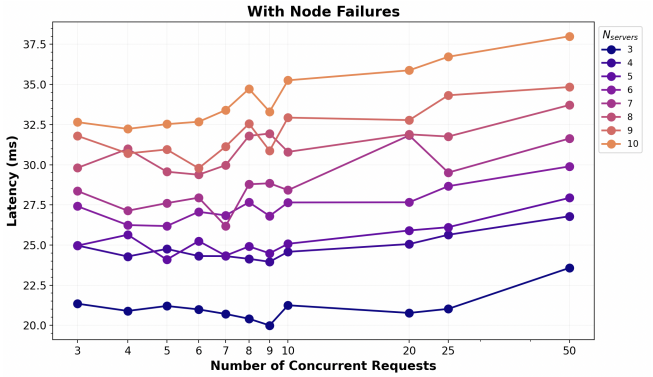

Figure 4: With node failures

are more nodes, there are more overhead in both inter-node communication, as well as overhead in making sure the correct logs are written to every server. As we increase the number of players, the latency also generally increases, which also makes sense since we are increasing the number of overlapping key/value reads and writes. Furthermore, we note that (at least in the parameter space we tested) latency scales roughly linearly with the number of Raft nodes and scales sub-linearly with the number of players. There are no "explosion" points where the latency starts increasing exponentially, showing that the system is well-equipped to support a considerably large number of players.

Somewhat surprisingly, when we add node failure events to our simulations, we find that latency values do not change very much at all. The occasional server failures affect latency in two different ways. On the one hand, latency suffers from the fact there are additional overheads triggered in the event that the leader goes down, as re-election has to happen. On the other, the latency benefits from the fact that during the time it takes for failed nodes to reconnect, there are less nodes for the system to communicate with and manage. The presence of these two opposing effects could foreseeably cancel each other out, explaining why we do not really see a difference in the two plots. It could also very well be the

case that overhead related to dealing with node failures is not part of the performance bottleneck (unless they are failing at an extremely high frequency). Given more time, we would ideally run this simulation with node failures over a larger number of iterations and get a better idea and estimates of not only the average latency but of the latency's tail behaviours.

5. Future Work

We would like to experiment with other forms of logging, for example with an algorithm like Lamport Clock [5] timestamps; this is a relatively lightweight technique that allows a distributed system to keep track of certain event orderings between multiple processes. While there are no insurances for fault tolerance and recovery, it would be interesting to measure the latency of a laser tag system built with Lamport Clocks and gRPC; this might allow us to get a sense of how much the latency we recorded from our Raft-gRPC system is due to the distributed log and fault tolerance overhead from Raft.

Furthermore, we would like to handle an edge case that came up in testing. Because transmitting the IR signals and receiving the IR signals are separated when a player/node is removed from the game either (because the node went down or the player no longer has any points), the player can still use their IR transmitter to interact with the other nodes.

We would also like for the game to be more seamlessly terminated. While we have a startup-script to handle starting the game in the future we would also like to have a shut down process that remove the data from the previous game automatically and terminates the raft + gRPC functionality. Currently this is a manual process.

6. Conclusion

We successfully demonstrate the feasibility of implementing a distributed laser tag game using Raspberry Pi Zero W single-board computers, the gRPC protocol, and the Raft consensus algorithm. By leveraging these technologies, we implement a resilient and scalable multiplayer gaming system that addresses the limitations of traditional centralized servers. The decentralized architecture ensures robust fault tolerance and seamless recovery from node failures, maintaining uninterrupted gameplay; which can handle an increasing number of players with minimal impact on latency, proving its scalability while maintaining fault tolerance.

Thus, we deliver a robust, scalable and *fun* multiplayer gaming experience.

Check out the video for a working demo of the system here.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[2] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the raft consensus protocol," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 154–165. [Online]. Available: https://doi.org/10.1145/2854065.2854081

[3] hashicorp, "Golang implementation of the raft consensus protocol," https://github.com/hashicorp/raft/releases, 2024.

[4] [Online]. Available: https://support.apple.com/en-us/111883

[5] 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