

Showdown

Final Report for the Distributed Systems Course [A.Y. 2025/2026]

Andrea Baldazzi

`andrea.baldazzi7@studio.unibo.it`

April 5, 2026

The project presents a simple distributed real-time game based on the showdown mode of Supercell's Brawl Stars. The players are forced to combat inside an arena and the last one standing is considered the winner. The only way to survive is to eliminate all the other players by shooting at them. The game features a distributed architecture that uses a peer-to-peer communication model between players and client-server model to enable the presence of a lobby server. To overcome unstable networks, the game implements client-side prediction mechanisms.

Disclaimer

During the preparation of this report, the author used LLMs to refine syntax and correct linguistic errors. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of the final report.

1 Concept

This project mainly consists of a desktop application with a graphics user interface (GUI), accompanied by a CLI server.

- **Primary Users:** The main target of this application are casual gamers, who want to have fun together in a simple online shooter game.
- **Use Case:** Before playing, a user can decide whether to host a lobby, allowing others join, or to participate in an already open one hosted on a friend's PC. Sessions are usually short, with each game lasting not more than a few minutes. This encourages quick rematches after a loss and gives players the chance to take revenge on their friends. For simplicity reasons, authentication and authorization mechanisms are not provided.
- **Interactions:** The users interact through a GUI, controlling their player's movements with the keyboard and shooting with mouse clicks. Since this is a real-time game, interactions between a player and the host of a lobby are very frequent and happens multiple times per second through the network.
- **Roles:** The game features two roles: the peer and the server. Multiple peers interact with each others during a game, while the server is responsible for keeping a list of active lobbies, enabling new players to search for open matches.
- **Data:** The system temporarily maintains data related to the game, such as players, bullets and walls which are required to ensure proper functionality of the system. No persistent storage is needed between sessions.
- **Geographical Distribution:** The system is designed to operate within a local area network (LAN), such as a home Wi-Fi environment. The system does not support NAT traversal and multiplayer functionality is limited to local networks.

2 Requirements Elicitation and Analysis

This section provides functional, non-functional and implementation requirements for the analysed system.

Functional Requirements

1. Players must be able to join a open lobby by specifying the address and the port of the host.
2. The players can host a lobby and allow others to join it.
3. At the beginning of a session all inputs should be disabled until the host decides to start the game. From this moment nobody should be able to join the lobby.
4. The system must support multiple players in the same lobby. **Acceptance criterion:** support at least 4 players, including the host.
5. During a game players must be able to interact with the game by moving their character and shooting bullets.
6. Interactions should happen in real-time, given the nature of the game.
7. The game should feature collision-checking mechanisms, restricting player and bullets inside the map and checking bullets hits.
8. Each player should have a finite amount of life and being hit by another player's bullet should decrease it.
9. When a player's life reaches 0, the player should be considered dead and the user controlling it is defeated. Instead, the last one remaining is considered the winner of the game.
10. Users must be able to connect to a lobby server, which facilitates the discovery of available games. Players should be presented with a list of open lobbies and be able to join them easily. When a user hosts a game, they should be able to make its existence public. **Acceptance criterion:** the lobby server must be reached without knowing its location.

Non-Functional Requirements

1. Availability: Player inputs should always be responsive. **Acceptance criterion:** the system must always react to user inputs; that is, a visible response should occur on the screen during gameplay whenever an action is performed.
2. State Consistency: The game state must be (eventually) synchronized across every player. **Acceptance criterion:** the game state should reach eventual consistency.

3. Performance: Gameplay should always be smooth. **Acceptance criterion:** maintain at least a frame rate of 30 FPS.
4. Network Partition: The system should continue to work even in presence of network partitions. **Acceptance criterion:** gameplay continue as normal even in presence of packet drops.
5. Usability: The application should be easy to use.

Implementation Requirements

1. Architecture: The system adopts a hybrid architecture that combines peer-to-peer communication between players—featuring authoritative game state—with a client-server model for interactions between peers and the lobby server. This approach ensures consistent gameplay while reducing overall system complexity.
2. Programming Language: The project is developed using Kotlin, a language known for its scalability, strong type system, and compatibility with a wide range of Java libraries. Additionally, at the time of writing, it is the language with which I'm most proficient, enabling more efficient development.
3. Software Tools: The peer application will be developed using the KTX (Kotlin wrapper for LibGDX) framework, a cross-platform game development framework designed for building high-performance games. KTX provides a comprehensive set of tools for graphics rendering, input handling, audio management, accelerating the development of games.

2.1 Relevant Distributed System Features

Below are listed the relevant distributed systems features of this project:

- **Transparency:** The system should abstract most networking details from the user, including data representation formats and communication protocols. However, complete transparency is not required. Users may still be exposed to certain networking information, such as the IP addresses and ports of hosts.
- **Fault tolerance and dependability:** Since game state messages are passed over the network, their content can be lost during communication. The system can continue to work even after a packet drop has occurred. This is possible because of client-side prediction mechanisms, which prioritize availability over consistency. Eventual consistency will be reached once a game state message arrives from the authoritative host.
- **Scalability:** The system should be capable of supporting multiple players connected to the same host. Although the architecture allows for a large number of concurrent users, the expected user base for this system is relatively small.

- **Security and Trust:** For simplicity reasons, the system does not feature any authentication or authorization mechanics. Since no sensitive data is handled by the application, that does not represent a problem.
- **Resource sharing:** During a game, there must be a shared game state managed by an authoritative host, eliminating the need for additional synchronization mechanisms. Furthermore, any data provided by the lobby server must be consistent and identical for all users who request it.
- **Heterogeneity of Components:** Although both the GUI application and the lobby server are implemented using the same technology, replacing one of them with a different technology should not pose a problem, provided that the interaction mechanisms and communication protocols are preserved.
- **Evolvability and maintainability:** The system is designed to be easily extendable, allowing new game features to be added without requiring significant changes to the underlying platform.
- **Performance and Concurrency:** The system must support multiple simultaneous communications. Because this is a real-time game, high performance is critical to minimizing latency and ensuring a smooth player experience.
- **Economy and costs:** The project is developed using existing, already-owned computers, without relying on external hosting platforms. As a result, there are no associated costs.

3 Design

This chapter explains the strategies used to meet the requirements identified during the analysis.

3.1 Architecture

The system employs a hybrid architecture that integrates both peer-to-peer and client-server communication architectures.

- **Peer-to-Peer:** During gameplay, peers communicate directly, exchanging messages essential to the gaming phase. Within each lobby, one peer serves as the authoritative host, to which all other peers connect to receive the latest game state. This allows players to interact without requiring an external server. Once a game ends, peers can switch roles depending on whether they want to host a new session or join as a client.
- **Client-Server:** The lobby server functions as the central hub for managing active lobbies. Each peer can communicate with the server to request a list of open lobbies or, in the case of a host, to register a new lobby. This approach greatly

simplifies the process of joining games, removing the need for users to manually enter IP addresses and ports.

3.2 Infrastructure

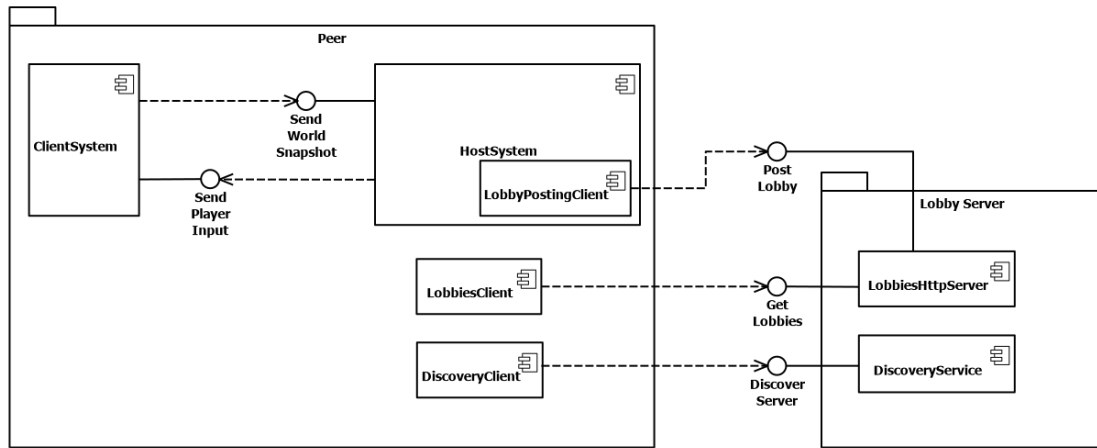


Figure 1: Component diagram of the system.

Infrastructural Components

The system is composed of only two components, the peer and the lobby server:

- **Peers:** The peer represents the graphical application that the player have access to. It can function as both a simple client and a lobby host, depending on user configuration.
- **Lobby Server:** The lobby server's role is to facilitate peers in finding available lobbies to join. If deployed, it remains in a listing state for everyone who wants to get access to the content it can offer.

Network Distribution

The components are distributed over a local area network (LAN), eventually on different machines. For simplicity, the system is not capable of NAT traversal, thus requiring the nodes to be on the same local network.

Components Discovery

- **Lobby Server Discovery:** Since the lobby server's location is initially unknown to peers, they discover it by sending a UDP broadcast message over the network.

If the server receives the message, it responds with its IP address and the port of its internal HTTP server.

- **Peer Discovery:** Peers can discover other players by interacting with the lobby server, which provides the IP addresses and ports of hosts that are currently accepting new connections.

3.3 Modelling

3.3.1 Lobby Server

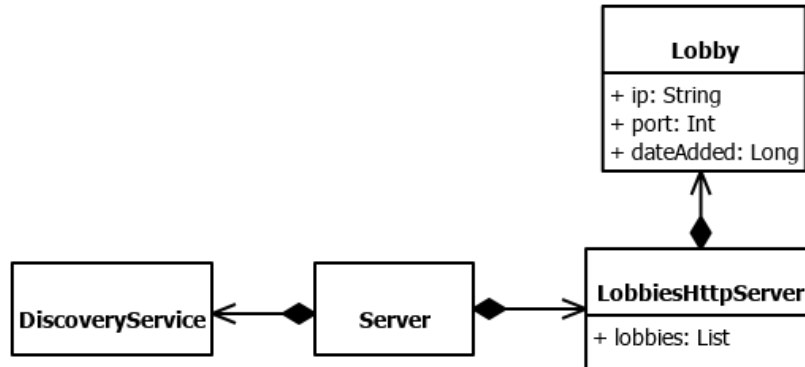


Figure 2: Lobby Server class diagram.

The Lobby Server is composed of two services, created at the beginning of its lifecycle:

- **UDP DiscoveryService:** It responds to discovery messages coming from peers.
- **LobbiesHttpServer:** The server listens for HTTP GET and POST requests from peers. It maintains an internal collection of available lobbies (identified by IP address and port), each associated with a timestamp of its last update, which is used to invalidate stale entries after a certain amount of time.

3.3.2 Peer

The peer application utilizes the Entity Component System (ECS) pattern provided by the KTX framework, introduced during the requirements phase. Because these components are pre-existing, certain classes are labelled (KTX) to distinguish them from the custom logic developed for this project, although this should not be part of class diagrams. Given that the ECS pattern can be difficult to understand, this section places extra emphasis on class diagrams to clarify the architecture, even though it's not part of distributed systems modelling (i.e. not required for the project).

Main structure

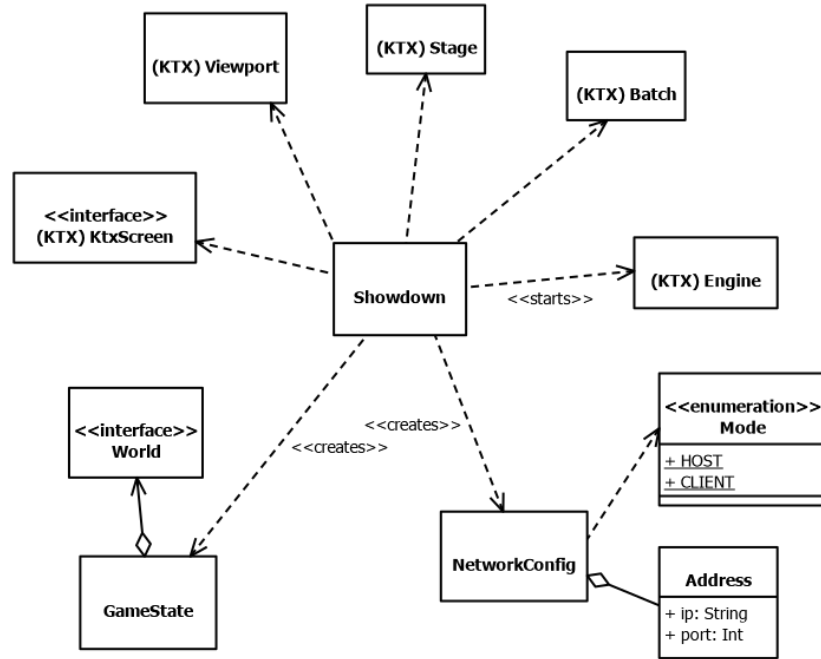


Figure 3: Peer main class diagram.

The main structure of the Peer application is composed by:

- **Showdown:** The main class, acting as an entry-point. At creation, it configures other items required by others, such the GameState, representing the current static configuration of a game and containing a passive world which just keeps record of map size and spawn points locations, NetworkConfig, used by Network managers, the Engine and graphics related entities.
- **KtxScreen, Viewport, Stage, Batch:** Graphics related classes provided by the framework.
- **Engine:** The core of the application. In a ECS system it's responsible for activating Systems every tick. Systems are described in the following diagram.

In an ECS (Entity Component System), the application consists of multiple **Entity**, each associated with various **Components**. Each component is a data-oriented structure storing the minimal state necessary (e.g., position, velocity, graphics, or health). A set of **Systems**, invoked every tick by the engine, processes specific entities by transforming the data within their components. For the sake of brevity, a comprehensive diagram of all components has been omitted. The systems designed for this application are represented in figure 4.

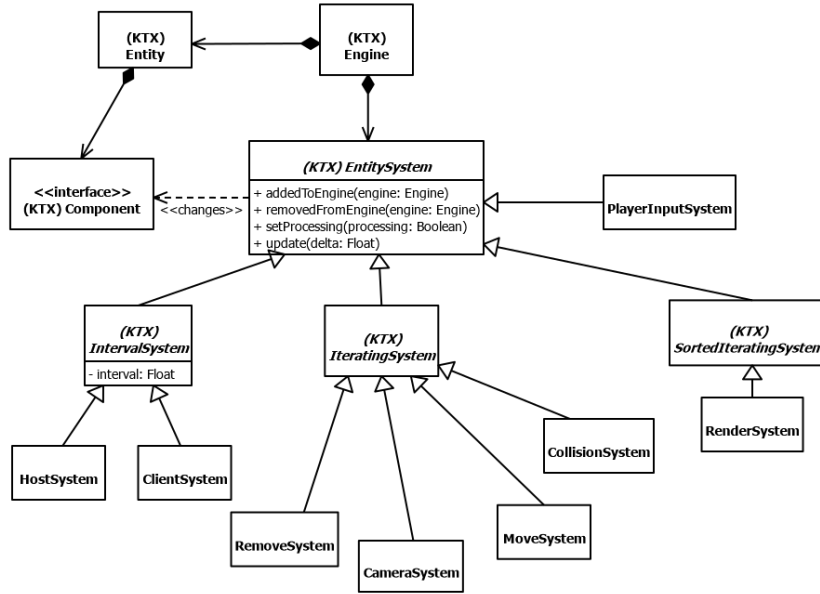


Figure 4: Systems class diagram.

The existence of an **EntityFactory** facilitate entities creation (and their related components).

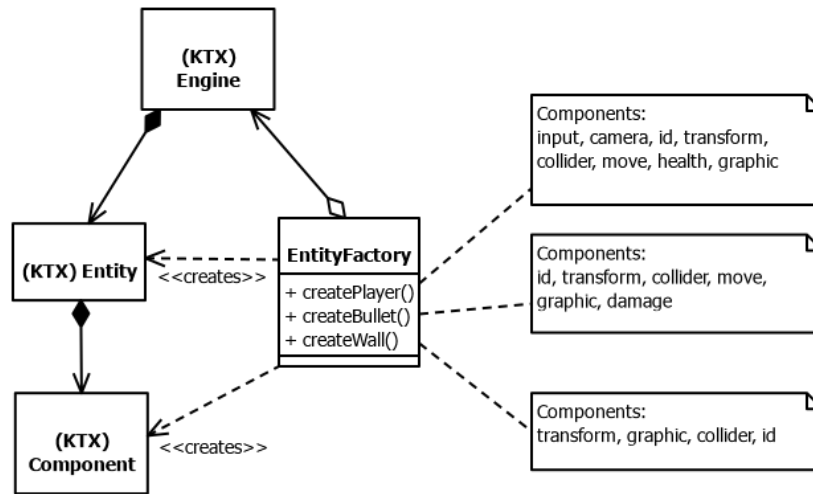


Figure 5: Entity factory class diagram.

Within the KTX framework, screens function as the logical controllers for different views of the application. A **Screen** acts as a container for logic rather than a direct graphical holder. Specifically, the HomeScreen architecture is composed of:

- **LobbiesHttpClient:** An HTTP client responsible for fetches lobbies from the

lobby server.

- **ServerDiscoveryClient:** Tries to establish communication with the lobby server by sending broadcast UDP messages in the local network.

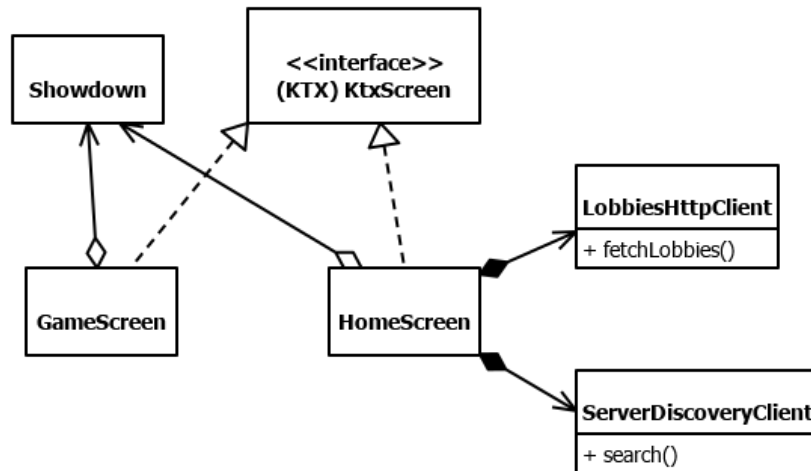


Figure 6: Screens class diagram.

Host System

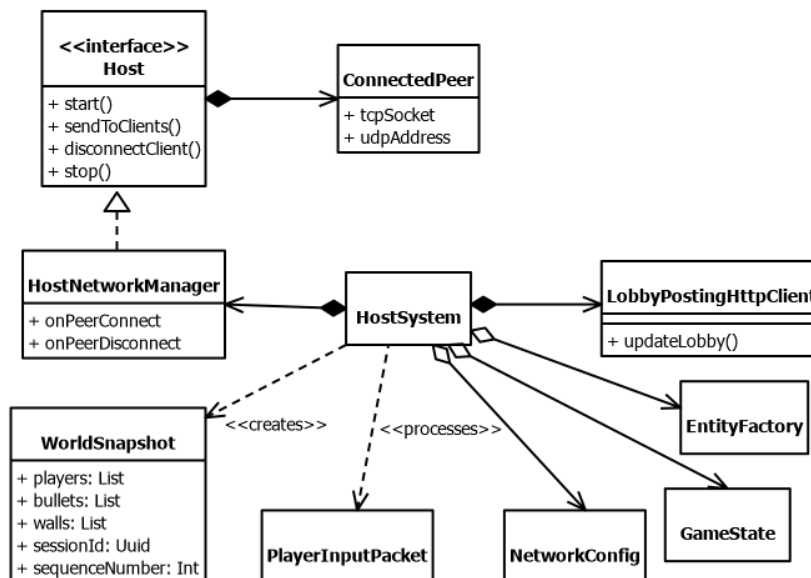


Figure 7: Host System class diagram.

The **HostSystem**, enabled only when the application is in host mode, is responsible for handling incoming inputs from peers and sending them game snapshots (authoritatively). It is composed by:

- **HostNetworkManager:** The manager responsibilities are:
 - Maintaining TCP connection with peers, detecting eventual disconnections.
 - Keeping a list of connected peers.
 - Sending/receiving UDP messages to/from peers
- **LobbyPostingHttpClient:** An HTTP client needed to interact with the Lobby Server to make the current lobby available for players to join.

The presence of an EntityFactory enables the creation of player entities following a connection of a new client.

Client System

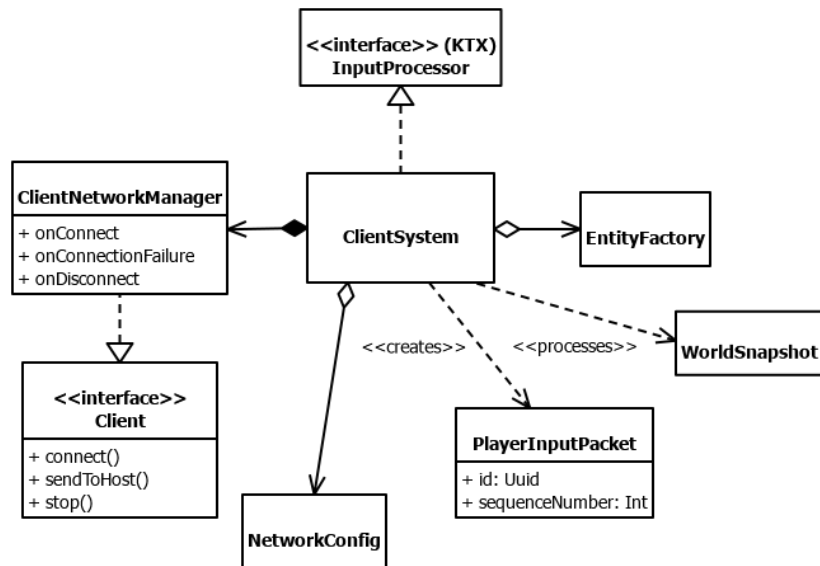


Figure 8: Client System class diagram.

The **ClientSystem**, enabled only when the application is in client mode, is responsible for sending inputs to the host and receive game snapshots from it. It is composed by:

- **ClientNetworkManager:** Similar to the host manager, it's responsible for handling the TCP connection with the host the client is connected to and send/receive UDP messages to/from it.

The presence of an EntityFactory enables the creation of entities as needed, such as during synchronization with host-provided world snapshots. This ensures the local client accurately reflects the authoritative game state.

3.4 Interaction

This section describes how the main components of the system interact with each other by analysing every possible interaction. Most of the communication is asynchronous, making the system resilient to network latency.

3.4.1 Game Connection

When a player wants to join a lobby, they do so by specifying the host's IP address and port (either manually or through the lobby server). This action triggers an asynchronous connection request from the ClientNetworkManager to the HostNetworkManager. If the request is successful, the HostNetworkManager responds by requesting the client's UDP port, which will later be used for message exchange during gameplay. At this stage, the user is assigned a random ID, and a corresponding player entity is created by the HostSystem.

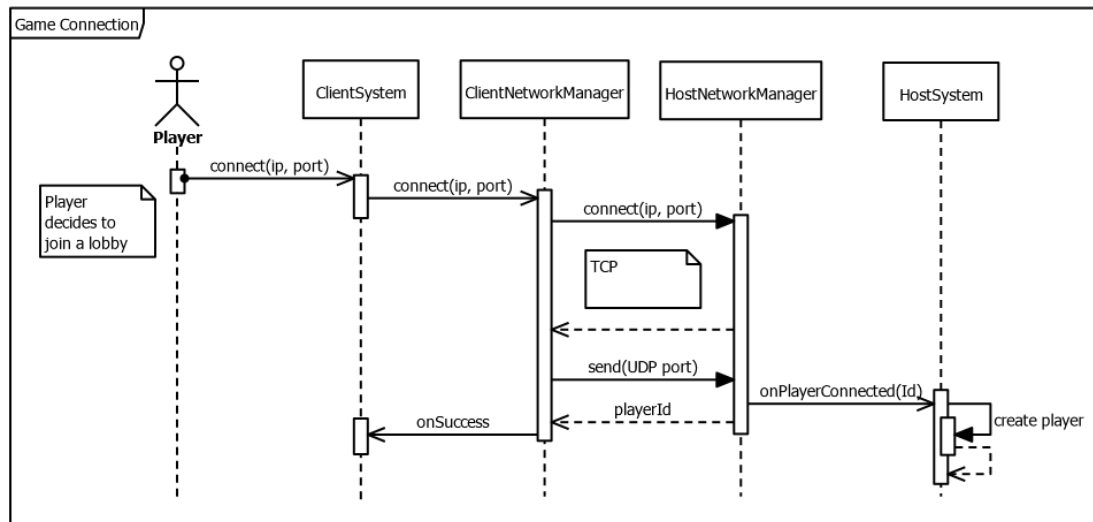


Figure 9: Game connection sequence diagram.

3.4.2 Game Disconnection

The system is capable of handling disconnection from both sides:

- **Client Disconnects:** When a client chooses to disconnect from a lobby, the ClientSystem sends an asynchronous command to the ClientNetworkManager, which then closes the connection with the host. Upon disconnection, the HostNetworkManager notifies the HostSystem, which removes the corresponding player entity from the game.

- **Host Disconnects:** If the host player disconnects while a game is in progress, the HostSystem sends an asynchronous command to the HostNetworkManager, which closes all active connections with the ClientNetworkManagers.

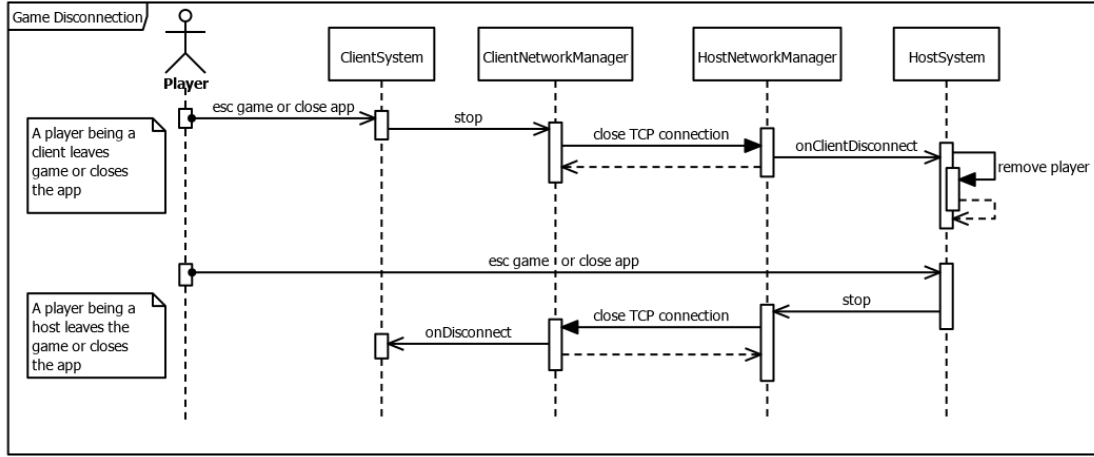


Figure 10: Game disconnection sequence diagram.

3.4.3 Gameplay

Gameplay consists of peers exchanging messages with each other based on their roles. The communication is bidirectional:

- **Client → Host:** When a player triggers an input (e.g., a button press or mouse click), it is first captured by the InputSystem, which updates the local game state. This local simulation occurs on both the host and the clients to enable client-side prediction. If the peer is a client, the ClientSystem also encapsulates this input into a payload packet and transmits it to the host using the ClientNetworkManager. Upon arrival, the HostNetworkManager passes the packet asynchronously to the HostSystem, which processes the input and updates the authoritative global game state.
- **Host → Client:** The process of transmitting the world snapshot from the host to the clients mirrors the one described above; however, this communication is triggered periodically according to a pre-determined update rate. This process is also asynchronous: the HostNetworkManager broadcasts the authoritative state, which the ClientNetworkManager receives and passes to the local ClientSystem for state reconciliation.

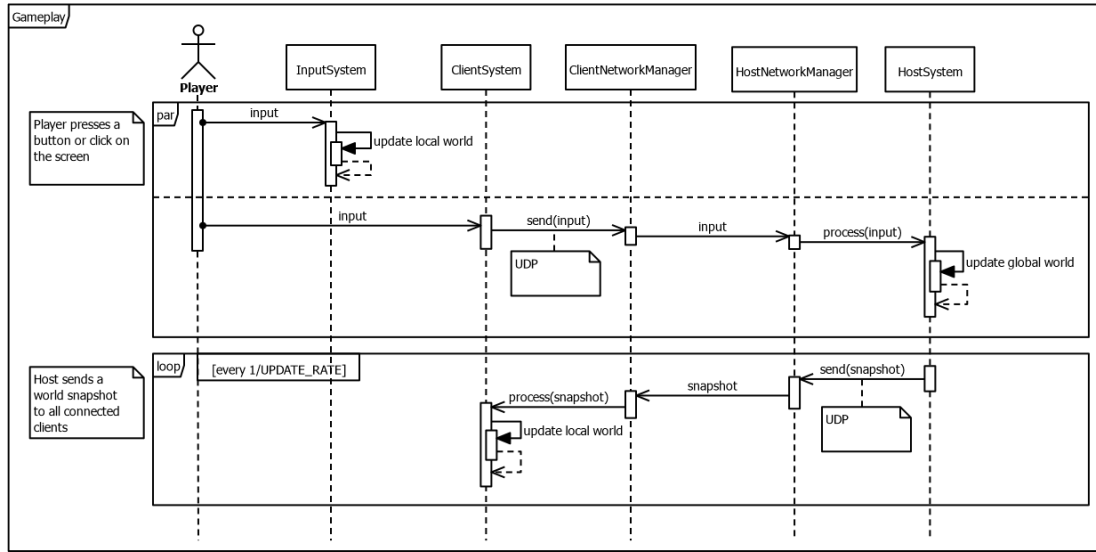


Figure 11: Gameplay sequence diagram.

3.5 Peer-Lobby Server

The process of getting available lobbies from the Lobby Server is divided into two phases:

- Server Discovery:** Since the lobby server is initially unknown to the peers, a discovery mechanism must take place first. Specifically, when initiated by the user, the ServerDiscoveryClient broadcasts a UDP message requesting the server's location. If a lobby server with an active ServerDiscoveryService is listening on the local network, it responds with its IP address and the TCP port required to access its lobby services. Once this handshake is complete, the peer stores the server's location for later communications.
- Lobbies Fetching:** Once a peer has acquired the server's location, it can query the server to retrieve the list of available lobbies. This asynchronous process involves the LobbiesHttpClient, which issues a GET request to the LobbiesHttpServer. Upon receiving the request, the server first removes any expired lobbies (based on their timestamps) and then returns the list of the remaining active ones.

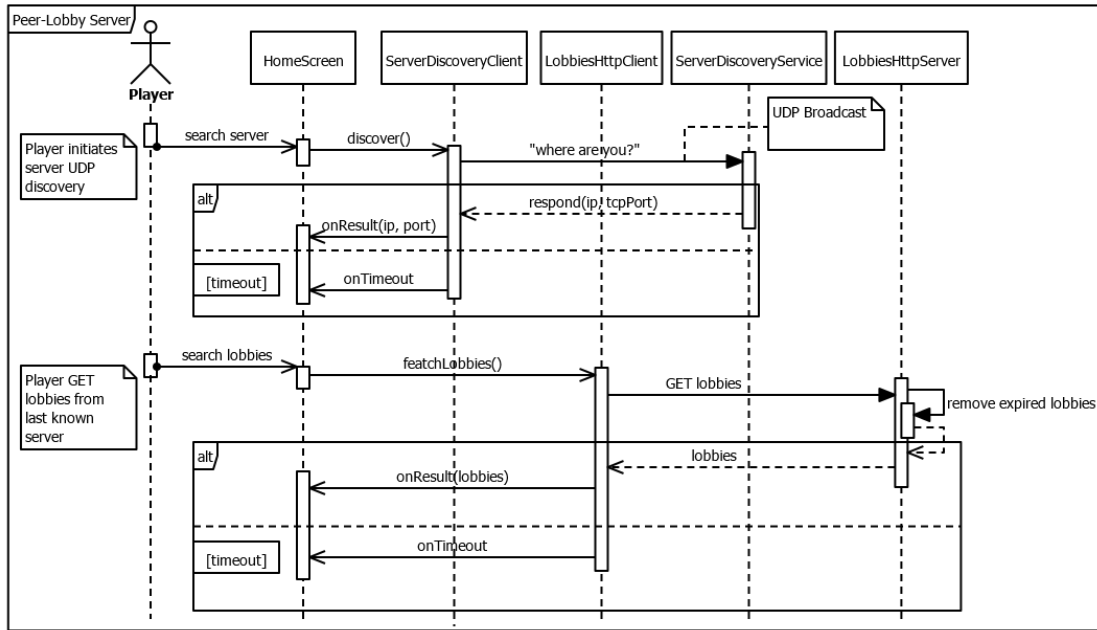


Figure 12: Peer-Lobby Server sequence diagram.

3.5.1 Host-Lobby Server

If the peer is acting as a lobby host and has acquired the lobby server's location, it periodically notifies the server of its active lobby while the game has not started. This message serves as a *heartbeat*, indicating that the lobby is still active.

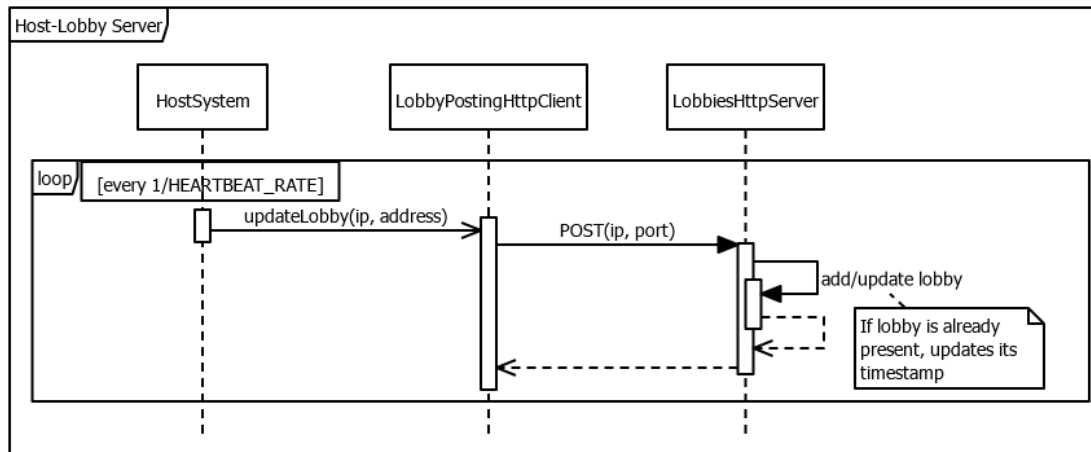


Figure 13: Host-Lobby Server sequence diagram.

3.6 Behaviour

In the current configuration, most components of the system are stateful. The *Host* maintains the authoritative global game state, while each *Client* holds a local version of the state. This local state may temporarily diverge from the authoritative one to support client-side prediction and ensure responsive gameplay. State transitions occur exclusively through message exchanges between components: these updates propagate changes in the game world and are also used to determine game progression.

The lobby server, by contrast, can be considered stateless. Although it maintains a collection of active lobbies, its behaviour remains consistent over time: it passively listens for requests, updates its internal list based on incoming messages (e.g., heartbeats), and responds to queries without maintaining complex session-specific state.

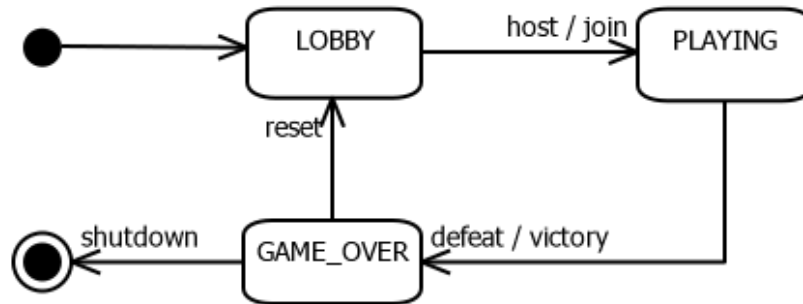


Figure 14: Peer state diagram.

3.7 Data and Consistency Issues

3.7.1 Data Storage Strategy

Due to the session-based nature of the game, there is no requirement for persistent data storage. All relevant data exists only for the duration of a game session and is discarded afterward. Consequently, all state information is maintained in RAM memory on the peers and the lobby server, making standard Kotlin data structures sufficient for managing and accessing the data efficiently.

3.7.2 Consistency Models

The game follows an **eventual consistency** model. To support client-side prediction, each client continuously simulates its own local version of the game state, even in the absence of immediate updates from the authoritative host. Periodically, the host sends state snapshots to all clients; upon receiving them, clients reconcile their local state with the authoritative one, correcting any differences.

3.8 Fault-Tolerance

3.8.1 Heart-beating

The only heartbeat mechanism in the system operates between the game host and the lobby server. To indicate that a lobby is still active, the host periodically sends heartbeat messages containing the lobby's information to the server. Upon receiving these messages, the server updates the corresponding entry's timestamp. If no heartbeat is received within a predefined interval, the lobby is considered inactive and is then removed from the list of available lobbies.

3.8.2 Timeouts

Timeouts mechanisms are provided in the following cases:

- **UDP Discovery:** When a peer needs to determine the location of the lobby server, it sends a UDP broadcast message over the network. If no response is received within a specified time interval, the attempt is considered to have timed out and the discovery process is aborted.
- **Lobbies Fetching:** The same workflow described above applies to retrieving the list of lobbies from the lobby server. The HTTP request may time out, in which case the operation is aborted.

3.8.3 Error handling

When an error occurs, the application handles it gracefully; however, for the sake of simplicity, no recovery mechanisms are implemented. Depending on which component fails, the following situations may occur:

- **Client:** If a client fails, it's simply removed from the current lobby.
- **Host:** Whenever a host fails, every connected peer is simply disconnected and returned to the home screen.
- **Lobby Server:** If the lobby server encounters a failure or becomes unavailable, peers will no longer be able to use its services. However, they can continue to operate normally; in this case, users must join lobbies manually by specifying the host's IP address and port.

3.9 Availability

3.9.1 Caching

While a game is in progress, each peer maintains the current game state in memory. If the connection to the host becomes temporarily unavailable, clients continue operating normally and proceed with local simulation, thus enabling client-side prediction.

3.9.2 Network Partitioning Behaviour

Under standard operating conditions, the system maintains both **Consistency** and **Availability**. However, in the event of a network partition, the architecture prioritizes Availability (AP). To prevent gameplay interruptions, the system features client-side prediction, allowing the local state to evolve independently of host updates. Once a snapshot arrives from the host, the system achieves eventual consistency by synchronizing the client state with the snapshot received.

3.10 Security

To simplify the project and focus on distributed systems principles, the system does not include any authentication or authorization mechanisms. In a real-world project, such security features would be essential to prevent unauthorized access and protect against potential misuse or malicious activity. However, the risk is limited, as no sensitive data is processed, thus leaving gameplay cheating as the only real concern.

4 Implementation

This section describes the implementation choices made during implementation of the project discussed so far.

4.1 Technological details

The implementation of the project is entirely done with the Kotlin language, alongside specific frameworks and libraries:

- *KTX*: the core of the peer application, it's a wrapper to Java LibGDX, a framework designed specifically for game development, offering an extensive collection of modules and tools.
- *Kotlin Serialization*: used for implementing automatic serialization and deserialization of data structures.
- *Ktor*: the most famous Kotlin library for network programming. It provides native support for sockets, as well as built-in HTTP server and client capabilities, and it's considered the standard in Kotlin applications.
- *JUnit*: considered the standard framework for unit testing in Java and Kotlin, it provides a lot of features to support extensive and structured testing of application components.
- *Gradle*: the most famous and open source build system for Java and Kotlin applications.

4.2 Network Protocols

Depending on the components involved during a communication, the following network protocols are used:

- **Host-Client:** The connection between peers—specifically between one acting as the host and another as a client—uses a combination of **TCP + UDP**. The TCP connection is used to exchange initial data, such as the peer ID and UDP port, and to maintain a persistent connection, used to detect disconnections, which would be difficult to handle using UDP alone. Game-related data, such as player input packets and world snapshot packets, is transmitted over UDP, which is connectionless and has lower latency. Since UDP does not guarantee packet order, world snapshots include a sequence number to ensure that outdated snapshots are discarded before being processed. This hybrid approach leverages TCP for reliable error detection while using UDP to minimize latency and maximize throughput during gameplay.
- **Lobby Server Discovery:** The discovery of the lobby server over the network leverages UDP to transmit broadcast messages. This would not be possible using TCP, since it requires creating a connection, impossible without knowing the destination address.
- **Peer-Lobby Server:** Communication between peers and the lobby server uses HTTP, a TCP-based protocol widely adopted in web development for its robustness and simplicity, guaranteed by the extensive library support in all programming languages.

4.3 Data Representation

Most of the data exchanged during communication is represented using JSON payloads. This well-known data format ensures consistent representation of data when converting between in-memory objects and text. The processes of serialization and deserialisation are handled automatically by the Kotlin Serialization Library, enabling faster development and more structured, maintainable code.

5 Validation

5.1 Automatic Testing

As mentioned in the implementation section, testing was done using the *JUnit* testing framework. Automated tests were conducted to test the system's behaviour both during normal operation and corner cases, such as errors. The following parts of the system are automatically test:

- **Game logics:** Game-related classes such as entity creation, components and systems behaviour, including `ClientSystem` and `HostSystem`.

- **Game data structures:** Game data structures, such as vector implementation, snapshots and configuration classes were tested for correctness.
- **Network managers:** Both the ClientNetworkManager and HostNetworkManager were tested both in isolation and in communication with each other to verify correct behaviour.
- **HTTP Lobby Server:** The lobby server was tested extensively, ensuring every possible interaction through HTTP methods lead to expected results.

The peer application was tested using a headless version of LibGDX, which simulates the real environment without requiring the full GUI infrastructure. The HTTP server was tested using Ktor’s extensive testing support, which provides classes and utilities for simulating server-client interactions.

Running the automated tests is possible from command line using `gradlew test`, which relies on the Gradle build tools.

5.2 Acceptance test

While automated testing validates core business logic, the inherent complexity of distributed systems necessitates manual verification within a real-world operating environment. I conducted manual tests across multiple personal computers on a local home network to validate end-to-end system behaviour. This process focused on scenarios that automated testing missed, specifically UDP service discovery, client synchronization in relation to the host, and edge cases involving unpredictable network latency.

The following criteria, defined in the requirements section, were successfully verified:

1. Support for at least 4 players, including the host.
2. Lobby server can be reached without knowing its location, thanks to UDP discovery mechanism.
3. Low latency client response, thanks to client-side prediction
4. Eventual consistency is always reached once a world snapshot is received from clients connected to an host.
5. A minimum of 30 FPS was consistently maintained, and in most cases, performance exceeded this threshold.
6. Gameplay is not interrupted in the event of a network partition. Specifically, a debug configuration was implemented in the peer application to simulate packet loss.

6 Deployment

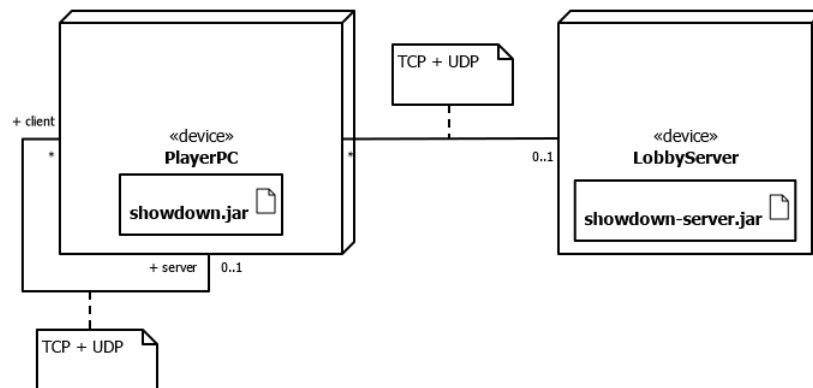


Figure 15: System deployment diagram.

The system is designed as a native Java application, with the peer functioning as a GUI application and the lobby server as a simple Java command-line program. Therefore, the only requirement for running the project is having **Java 25** or higher installed on the machine.

6.1 Windows Only: Firewall

On some Windows machines, the Windows Firewall may block UDP messages before they reach the application. This issue was observed during the testing phase and can prevent proper communication between peers and the lobby server. It can be resolved by adding the Java runtime environment to the firewall's whitelist, specifically, adding **java.exe** and **javaw.exe** to the exclusion list.

7 User Guide

The application is designed to be simple to use. Below a user guide is provided:

1. **Start the peer application:** double-click on the JAR file (only on Windows) or run the following command on a terminal:

```
java -jar showdown-1.1.0.jar
```

2. **Start the lobby server:** open a terminal and executes the following command:

```
java -jar showdown-server-1.1.0.jar
```

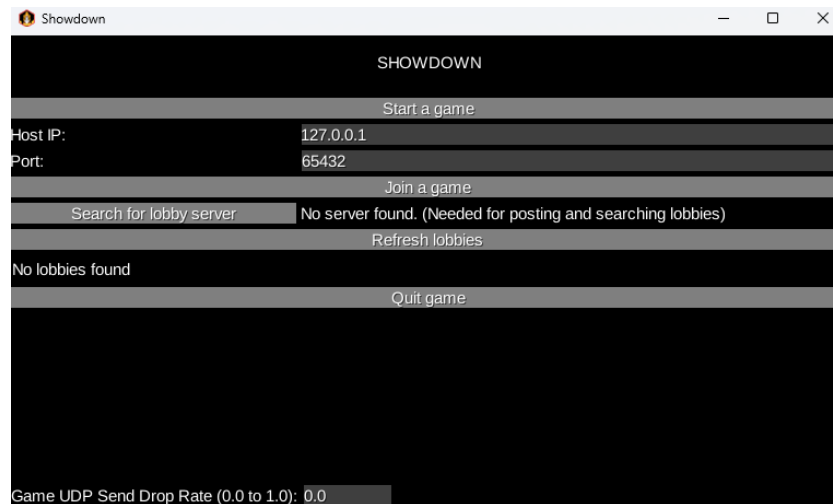


Figure 16: The Home Screen.

3. **Server discovery:** in order to discover the lobby server, click on "Search for lobby server". If the server is able to respond, its location will be displayed on the right of the button.

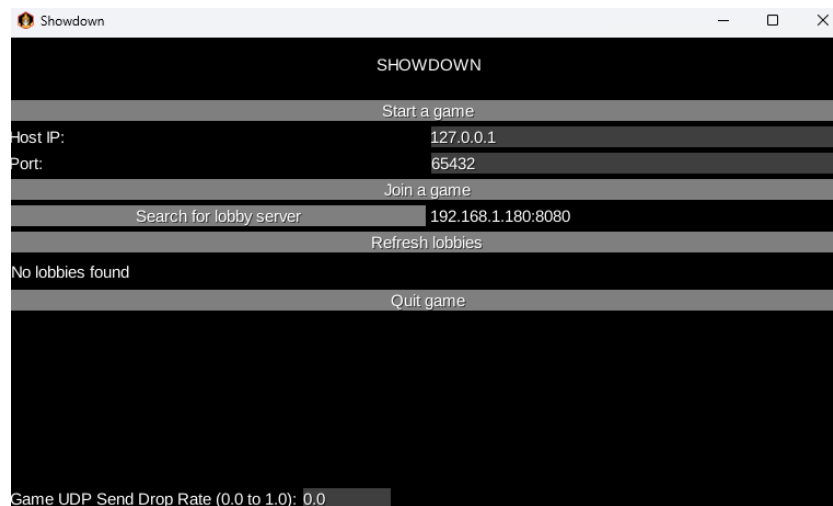


Figure 17: Discovering the Lobby Server.

4. **Host a lobby:** to host a lobby, click on "Start a game". A new game will begin with input disabled at first. In order to start the game, click on the dedicated button. Note that every other peer will be able to join the lobby only while the input is disabled and only if the lobby server has been located before starting the game.

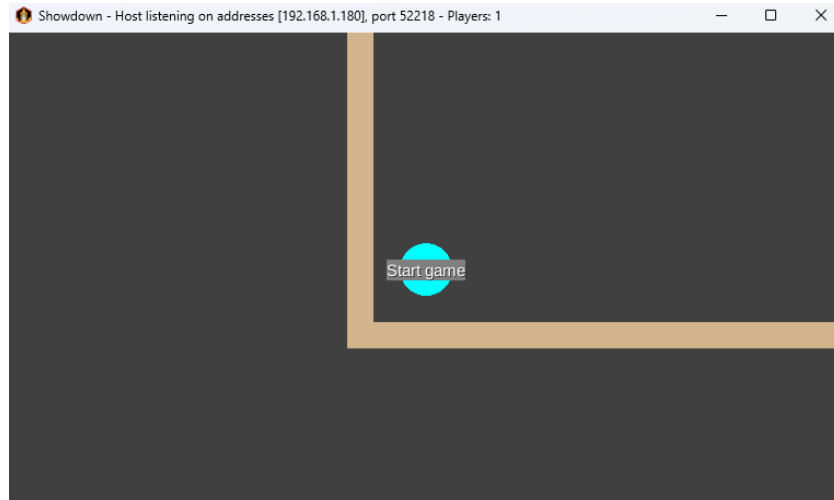


Figure 18: Host starts a new game.

5. **Join a lobby:** to join a lobby, a client can either manually enter the host's IP address and port or use the lobby server for convenience. By clicking on "Refresh Lobbies," the client can retrieve the list of active lobbies; if any are available, they will be displayed below and can be selected to join.

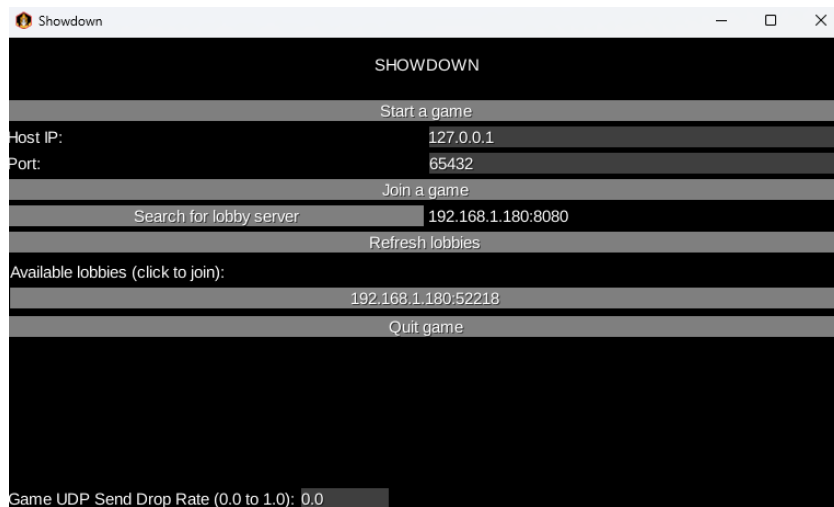


Figure 19: Fetched lobbies are displayed.

6. **Gameplay:** once the game are started, inputs are enabled. To move the character you can use the WASD keys or the arrow keys on the keyboard. To fire a bullet, just click on the screen in the direction you want to shoot at.

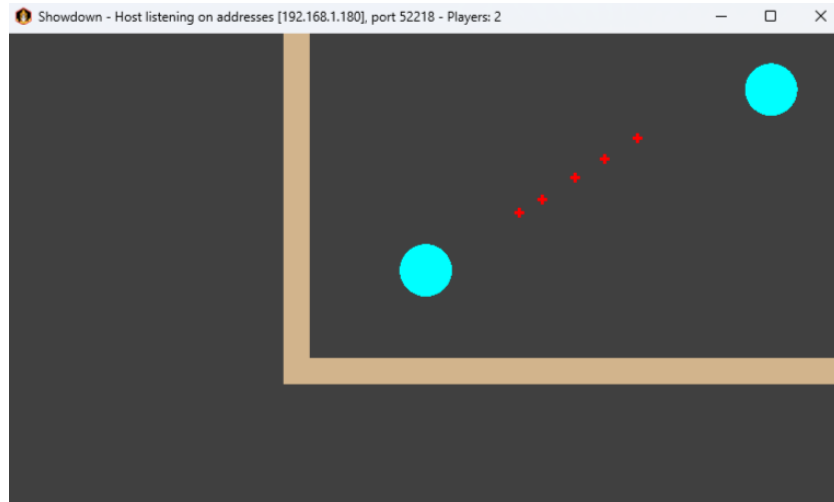


Figure 20: Gameplay showing two players and some bullets moving in the arena.

7. **Pause menu:** once the game has started, it can be paused by pressing the ESC key, allowing the user to leave the lobby. Note that this does not stop the game simulation, so players should be careful not to die while the menu is open.

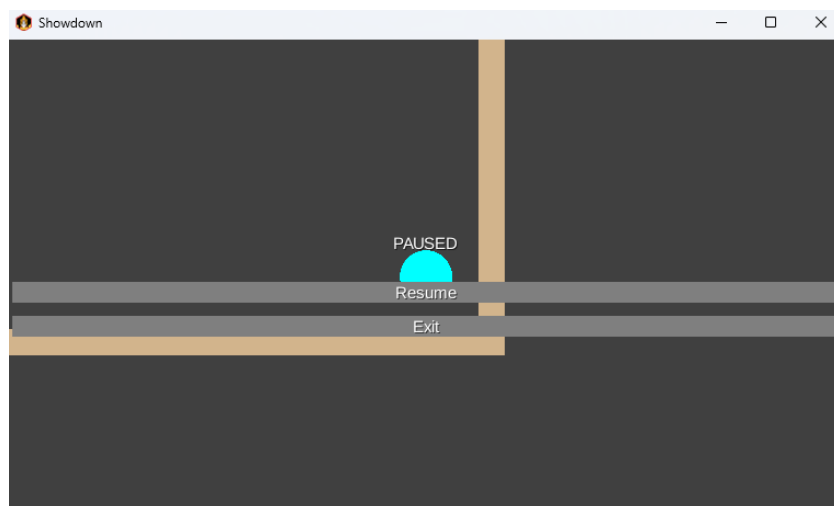


Figure 21: Pause menu.

8. **Game end:** the game ends when a player is eliminated (i.e., runs out of health) or when they are the last player remaining. These conditions correspond to defeat and victory, respectively.



Figure 22: Victory Screen.

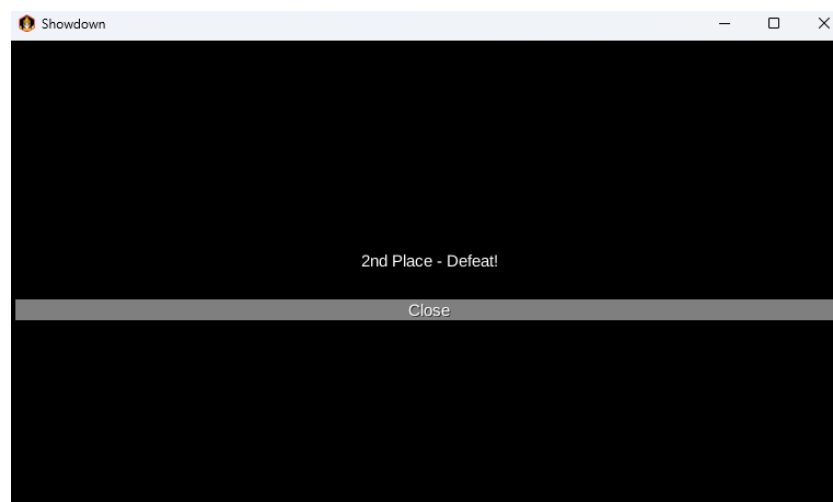


Figure 23: Defeat Screen.

9. **Game disconnection:** if the host exits the lobby while a game is in progress, all connected peers are disconnected. A message is displayed on the screen to inform the users.

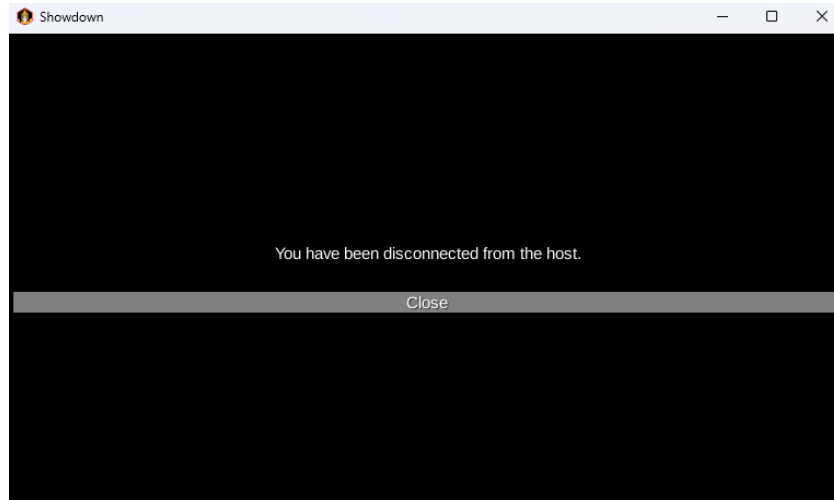


Figure 24: Disconnection message.

8 Self-evaluation

This project was designed and developed entirely by me as a single-person effort. Despite already being familiar with the MVC pattern, I chose to explore the ECS pattern through the KTX framework. While this aspect is not directly related to distributed systems, it provided a valuable learning opportunity.

During the design and implementation of interactions between peers, I encountered a significant challenge: ensuring smooth and low-latency gameplay, required due to the real-time nature of this game. This requirement naturally led me to consider UDP. However, I soon faced the difficulty of reliably detecting connection establishment and disconnections, which UDP alone cannot handle effectively. To address this, I adopted a hybrid approach combining TCP and UDP, leveraging the strengths of both protocols.

This solution, however, does not account for NAT traversal, as implementing such functionality was beyond the intended scope of a single-person project. After completing the peer application, I addressed the problem of discovering the lobby server's location. The solution I implemented, UDP-based discovery, proved to be simple and effective for the intended use case, despite its limitations.

One aspect of the project I am particularly satisfied with is the implementation of client-side prediction. This was made relatively straightforward by the system design, as the host and the client differ only by a single system: either the `HostSystem` or the `ClientSystem` can be active at the same time.

On the other hand, I think the GUI is quite minimal and, at times, lacks visual refinement. However, as outlined in the project proposal, GUI design was outside the primary focus on distributed systems and would have required more development time.

9 Future works

Although the current implementation fully meets the project requirements, the system could receive further improvements:

- **Fault tolerance:** With the current design, whenever a host fails, all the connected clients are disconnected and the lobby is cancelled. It think it would be a good idea to implements host failure detection and automatic host re-election, ensuring transparency and continuity for the clients.
- **NAT traversal support:** I think it would be nice if the system supported NAT traversal, enabling players on different networks to play together. I've done some research and I believe that WebRTC could do the job. However, I decided not to implement it, as it would have added significant complexity to the system.
- **Security and Encryption:** Currently, the system does not include any authentication or authorization mechanisms. In a production environment, these would be necessary to prevent misuse. Additionally, JSON payloads are transmitted in clear text over the network. Although no sensitive data is currently exchanged, a proper encryption method would be ideal to prevent interception.
- **Gameplay improvements:** Right now, the gameplay is relatively basic, allowing players only to move and shoot. In the future, it would be beneficial to introduce additional gameplay mechanics to enrich the experience. Such enhancements would not require a major redesign of the system and would better demonstrate its scalability.