HNB Final Report Template

Alessandro Becci, Luca Tonelli

Abstract

The project aims to build a peer-to-peer Android game using the Bluetooth communication protocol. The objective is to implement a reaction-based game where the players have to react faster than each other to a specific random signal, miming a “high noon dual”. Any node of the network (a phone) will be able to create or participate in a match.

At the start of the application, users can choose to become either a server or a client. The server has the ability to create a lobby, while the client can join an existing one. Once all players are ready, the server will be able to initiate the match. During game, the players will see a button appear on their screens, and they must press it as quickly as possible. The first player to press the button wins the match. The button will appear at random times and positions on the screen, adding an element of surprise and challenge.

After all the players have finished the game, they will be able to see the leaderboard with reaction times. Once the game is over, the players can return to the lobby.

Goal/Requirements

The project aims to develop a game that utilizes Bluetooth communication protocol. A key initial step is to evaluate whether Bluetooth possesses the necessary capabilities to support the game's requirements. To assess this need, developing a prototype for assessing it is essential.

A significant technical challenge involves also designing a strategy that can accurately measure players reaction times accounting for Bluetooth latency.

Additionally, the project must incorporate robust fault tolerance mechanisms to enhance reliability, specifically addressing scenarios such as connection interruptions or players exiting mid-game.

We expect that the application will offer various features, related to lobby and game phases.

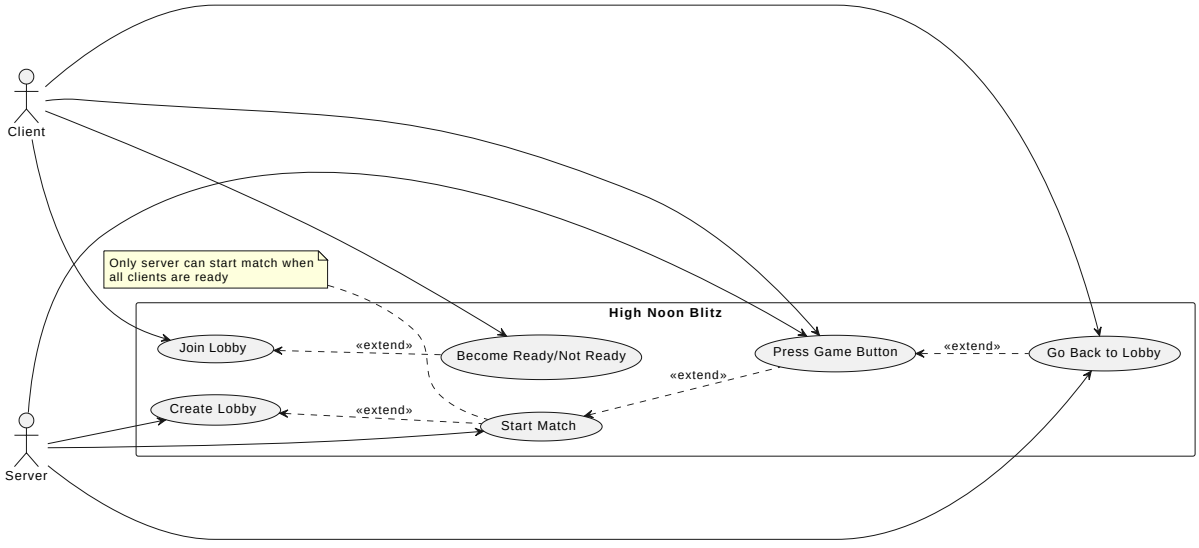


Figure 1. Use Cases Diagram showing the main use cases of the system.

Scenarios

Users interact with the application when they want to play a round of the game. Upon opening the application, they can choose between two roles:

Server Role: The user can make their phone discoverable via Bluetooth connection, creating a lobby that others can join.

Client Role: The user can scan for discoverable phones in the area and connect to them to join an existing lobby. While clients typically connect to servers, they can also connect to other clients, in which case server information is shared across all connected phones.

Once connected, clients can indicate their readiness to the network. The server can initiate the match only when all connected clients have marked themselves as ready.

During gameplay, all users must react to the HNB Button that appears in a random location on their screens and at random times. After all players have clicked the button, a leaderboard displays the results.

Following a match, players can return to the lobby for another game or disconnect to join different players.

The system accounts for potential connection loss between phones during any phase of the game, ensuring appropriate handling of these scenarios.

Informal description of the ways users are expected to interact with your project. It should describe *how* and *why* a user should use / interact with the system.

Self-assessment policy

The quality of the produced software will be assessed through a multi-dimensional evaluation approach:

1. **User Acceptance Testing**: We will conduct structured testing sessions with two distinct user groups:

- Participants familiar with the game mechanics to evaluate advanced functionality
- First-time users to assess intuitiveness and learning curve

2. **Cross-device Compatibility**: The application will be tested across multiple Android devices spanning different:

- Screen sizes and resolutions
- Hardware specifications
- Android OS versions (from 12.0 to latest)

3. **Network Resilience Testing**: We will simulate various connection scenarios to evaluate robustness:

- Intentional disconnections at critical points in the gameplay loop
- Testing failure scenarios for both server and client roles
- Observing effects on gameplay experience

4. **Performance Evaluation**: While exact measurements are not feasible, these key performance aspects will be qualitatively evaluated:

- Application responsiveness

- Bluetooth connection stability
- Effectiveness of reaction time calculation approach

Requirements Analysis

Implicit Requirements

1. **Synchronization Mechanism**: There's an implicit need for time synchronization between devices to ensure fair gameplay and accurate reaction time comparison.
2. **User Interface Adaptability**: The application must adapt to different screen sizes and orientations since the button appears at random positions.
3. **State Management**: The system needs robust state management to track game phases (lobby, gameplay, results) across all connected devices.
4. **Network Topology Management**: As clients can connect to other clients with server information being shared, there's an implicit requirement for managing a fully connected network topology.

Implicit Hypotheses

1. **Device Compatibility**: There's an assumption that Android devices with Android 12+ will have compatible Bluetooth implementations that work with the application.
2. **User Behavior**: The design assumes users will naturally understand how to navigate between server/client roles and gameplay phases.

Non-functional Requirements

1. **Responsiveness**: The application must respond immediately to user input, especially during gameplay.
2. **Reliability**: The system must handle connection interruptions and player exits gracefully.
3. **Usability**: The interface must be intuitive for both first-time and experienced users.
4. **Scalability**: The system should support multiple connected players in a single game session.

Best Suited Model/Paradigm/Technology

1. **Communication Model**: A peer-to-peer network with a designated coordinator (server) is most appropriate, allowing for direct communication between devices while maintaining a central authority for game state.
2. **Programming Paradigm**: Object-oriented programming for the overall structure, with event-driven programming for handling UI interactions and network events.
3. **Technology Stack**:
    - Android Bluetooth API for communication
    - Android's UI Jetpack Compose for the interface

Abstraction Gap

1. **Bluetooth API vs. Game Requirements**: Android's Bluetooth API is general-purpose and doesn't specifically address the timing precision needed for reaction games. This gap requires custom implementation of timing mechanisms.
2. **From Client-Server to Fully Connected Topology**: While the Android bluetooth design adopts a conventional client-server structure, the network needs to be a fully connected topology where each node can establish direct communication with any other node. This evolution necessitates the

implementation of additional abstraction layers to manage dynamic peer-to-peer interactions and ensure consistent state across the network.
3. **Fault Tolerance**: Standard Bluetooth connection handling doesn't account for the specific game states and recovery mechanisms needed, requiring custom reconnection and state recovery logic.
4. **Timing Precision**: There's a significant gap between the millisecond-level precision needed for reaction time comparison and the variable latency inherent in Bluetooth communication. This will require developing a compensantion algorithm.

Design

Bluetooth latency solution

To address the critical timing challenges in our reaction-based game, we first conducted fundamental latency research by developing a test application called Connectivity Radar. This diagnostic tool measured Bluetooth message transmission times between devices, revealing latencies consistently in the hundreds of milliseconds range.

Given that the average human reaction time is approximately 200ms, this latency presented a significant design challenge. Direct server-to-client coordination would create an unfair advantage for devices physically closer to the server, as they would receive the "start game" signal earlier than more distant devices.

To ensure fair gameplay regardless of network topology, we implemented a decentralized timing mechanism:

1. Local Timing Architecture: When a device receives the game start message, it doesn't immediately display the reaction button.

2. Synchronized Random Delays: Instead, each device launches a coroutine with a device-specific random delay interval.

3. Timestamp Recording: At the conclusion of this delay, the system:

    - Records the precise local timestamp (T_1)
    - Displays the reaction button to the player

4. Reaction Measurement: When the player taps the button:

    - A second timestamp (T_2) is recorded
    - The device calculates the reaction time as (T_2 - T_1)
    - Only this time difference is broadcasted

Given these premises we go forward in explaining the structure of our project.

Structure

Controller Structure

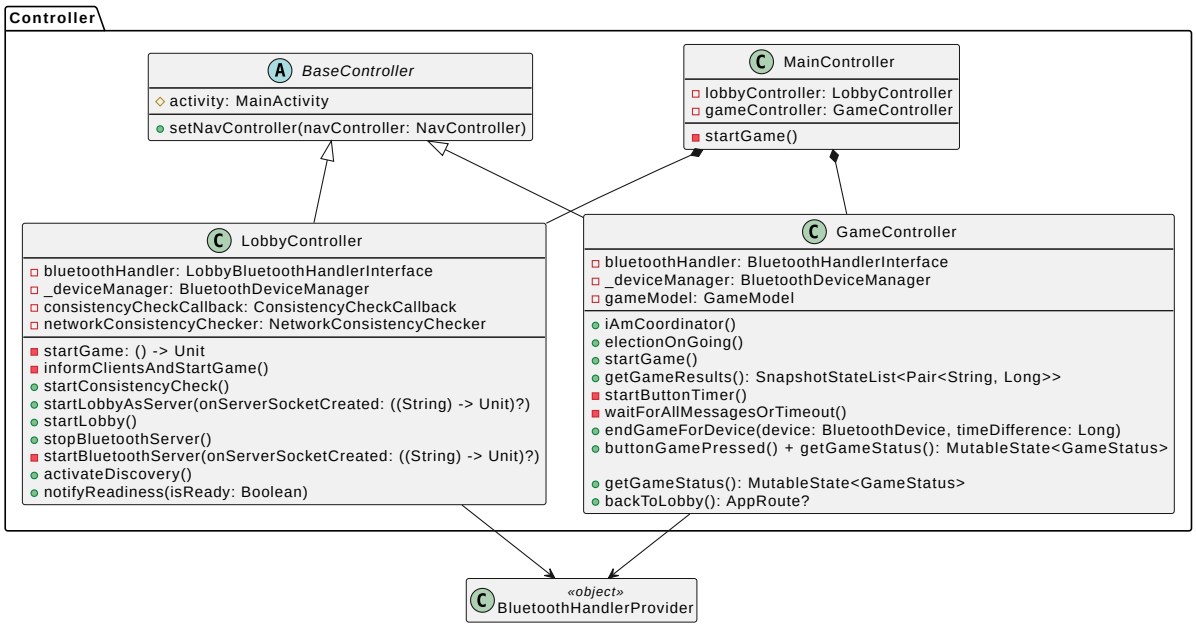


Figure 2. UML Controller Schema showing the High-Level Controllers of the Application.

The Controller diagram illustrates the core control components of the application, responsible for managing game state and user interactions. This layer follows a hierarchical structure with well-defined responsibilities.

Key Components:

- **BaseController**
  - An abstract class that forms the foundation for specialized controllers.
  - Maintains a reference to the main activity and provides navigation control.
- **MainController**
  - Acts as the central orchestrator by composing both the `LobbyController` and `GameController`.
  - Coordinates transitions between lobby and game states through the `startGame()` method, bridging the pre-game setup with gameplay.
- **LobbyController**
  - Extends `BaseController` and manages the pre-game lobby where players connect via Bluetooth.
  - Responsibilities include:
    - Managing Bluetooth connections for the lobby phase.
    - Handling player readiness states.
    - Conducting network consistency checks.
    - Coordinating transitions to the game phase.

  Key Methods:

  - `startLobbyAsServer`
  - `activateDiscovery`
  - `notifyReadiness`

- **GameController**
  - Extends `BaseController` and manages active gameplay.
  - Responsibilities include:
    - Handling in-game Bluetooth communications.
    - Maintaining the game model and state.
    - Tracking game timing and player interactions.
    - Managing game completion and results.
  -

  Key Methods:

  - `buttonGamePressed`
  - `endGameForDevice`
  - `getGameResults`

- **BluetoothHandlerProvider**
  - A singleton object that acts as a factory for Bluetooth handlers.
  - Supplies the appropriate handler instances to both the `LobbyController` and `GameController`, ensuring consistent Bluetooth communication throughout the application lifecycle.

Message System

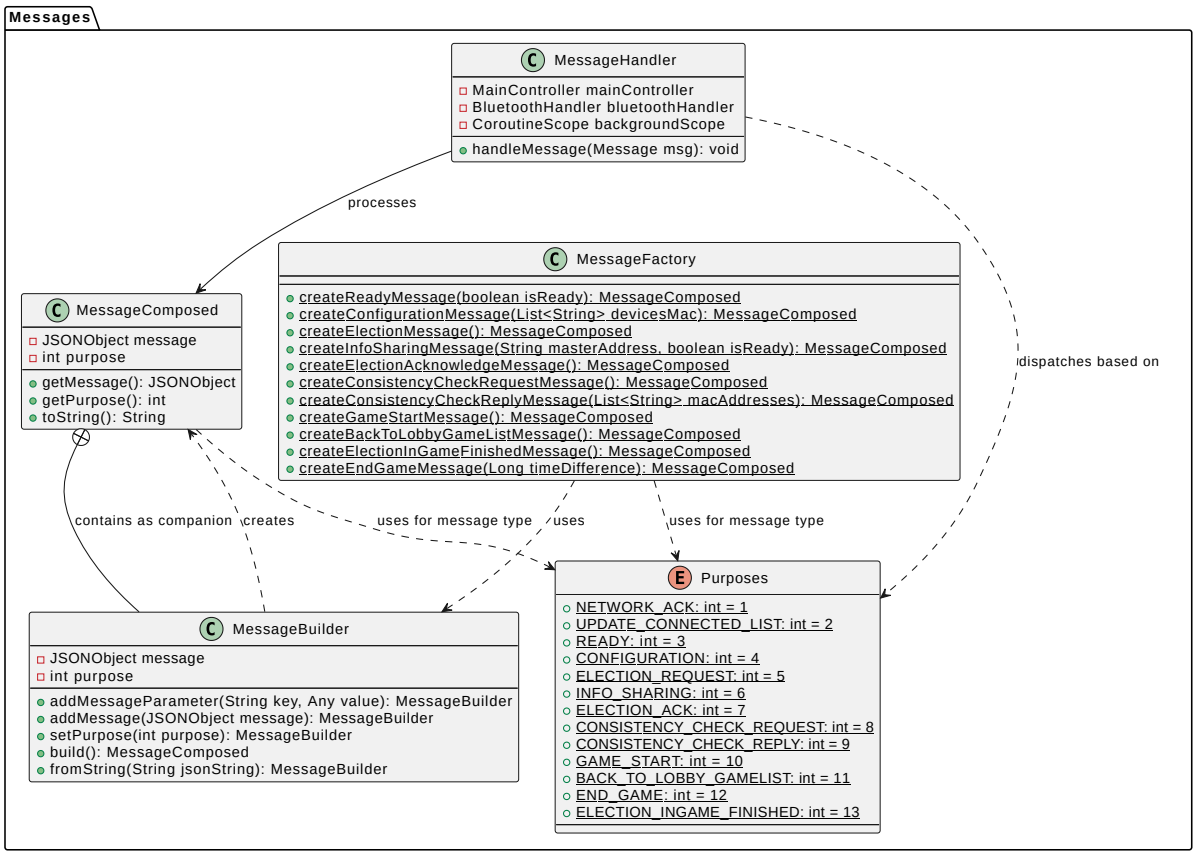


Figure 3. UML Messages Schema showing the messages infrastructure.

The Messages diagram shows the messaging infrastructure that enables communication between devices in the application. This system provides a structured approach to creating, sending, and handling messages using a builder pattern and factory approach.

Key Components:

- **MessageComposed**
  - Represents a fully formed message ready for transmission.
  - Encapsulates:
    - A `JSONObject` containing the message content.
    - An integer purpose identifier specifying the message type.
    - Methods to access the message content and purpose.
- **MessageBuilder**
  - Follows the builder pattern for step-by-step message construction.
  - Features:
    - Methods to add parameters or entire JSON objects.
    - A purpose setter to define the message type.
    - A `build` method producing a completed `MessageComposed` object.
    - A `fromString` method to reconstruct messages from a JSON string.
- **MessageFactory**
  - A static factory class providing convenience methods for creating standard message types.
  - Includes:
    - Ready messages indicating player readiness.
    - Configuration messages containing the connected devices.
    - Election messages for coordinator selection.
    - Messages to check the network consistency.
    - Game control messages for starting, ending, or transitioning between game states.
- **MessageHandler**
  - Processes incoming messages by:
    - Receiving raw message objects.
    - Interpreting them based on their purpose.
    - Dispatching appropriate actions to the `MainController`.
    - Managing background processing through coroutines.
- **Purposes Enum**
  - Defines standard message types with assigned integer codes, including:
    - Network acknowledgments (`1`).
    - Connected list updates (`2`).
    - Ready status changes (`3`).
    - Election requests and responses (`5-7-13`).
    - Information sharing (`6`).
    - Consistency check request and response (`8-9`).

- Game control signals (`10-12`).

Bluetooth Handler And Management

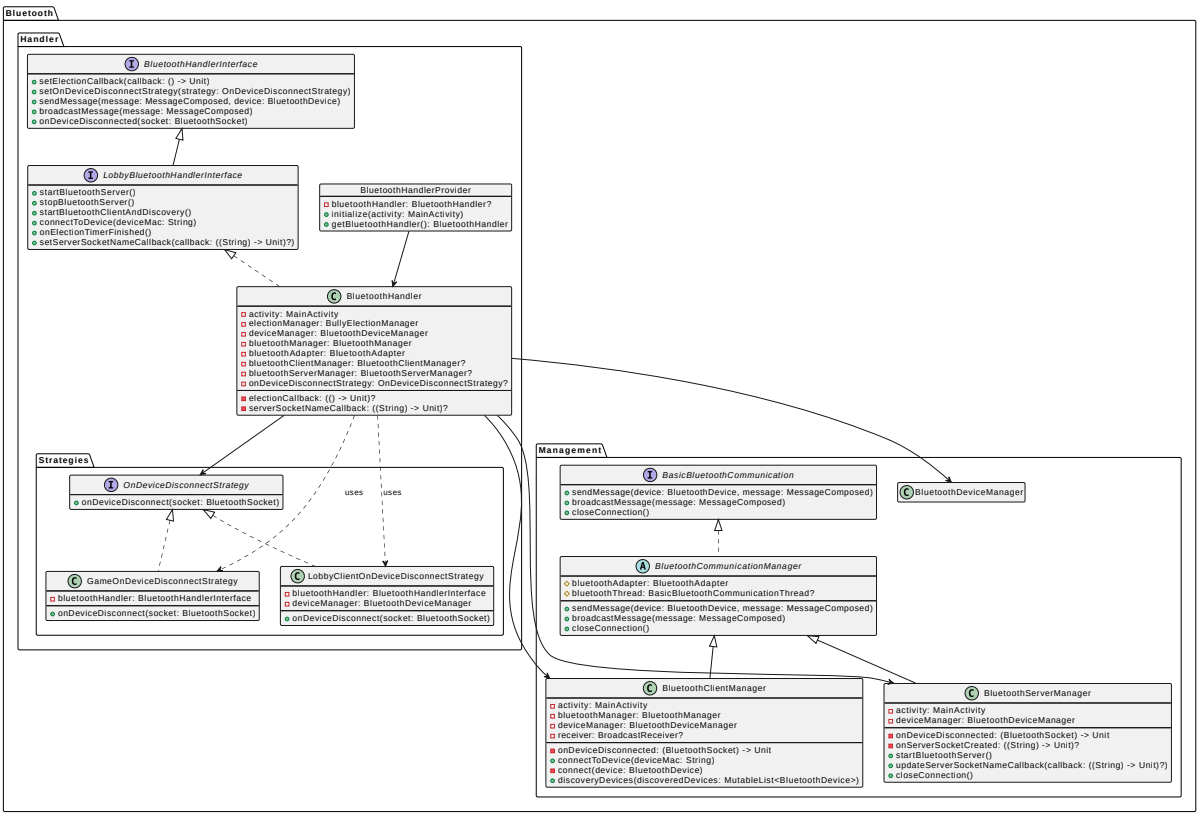


Figure 4. UML Bluetooth Handler and Management Schema.

This diagram integrates the Bluetooth Handler layer with management components, showing how higher-level control interfaces with lower-level Bluetooth operations.

Key Components:

- **BluetoothHandlerInterface**
  - Defines core functionality for Bluetooth communication, including:
    - Setting election callbacks for coordinator selection.
    - Defining disconnection handling strategies.
    - Sending targeted and broadcast messages.
    - Responding to device disconnection events.
- **LobbyBluetoothHandlerInterface**
  - Extends `BluetoothHandlerInterface` with lobby-specific operations:
    - Starting and stopping the Bluetooth server.
    - Initiating client connections and device discovery.
    - Connecting to specific devices.
    - Managing election timeouts.
    - Setting callbacks for server socket creation.
- **BluetoothHandler**

    - Implements `LobbyBluetoothHandlerInterface`, bridging the controller layer with the Bluetooth infrastructure.
    - Manages:
        - References to the main activity and device manager.
        - The Bluetooth adapter and managers.
        - Disconnection strategies.
        - Callbacks for elections and server events.

- **BluetoothHandlerProvider**
    - A singleton object that serves as a factory and access point for `BluetoothHandler`.
    - Responsibilities:
        - Initializes the handler with the main activity.
        - Provides global access to the handler instance.

- **OnDeviceDisconnectStrategy Interface**
    - A strategy pattern interface defining different responses to device disconnection events.
    - Provides a method for handling disconnection events with context-specific behavior.

- **GameOnDeviceDisconnectStrategy & LobbyClientOnDeviceDisconnectStrategy**
    - Concrete implementations of `OnDeviceDisconnectStrategy`.
    - Handle device disconnections in different contexts:
        - `GameOnDeviceDisconnectStrategy` → Disconnections during active gameplay.
        - `LobbyClientOnDeviceDisconnectStrategy` → Disconnections during lobby setup.

- **BluetoothCommunicationManager & Extensions**
    - Contains communication managers:
        - `BluetoothCommunicationManager` → Abstract base for communication management.
        - `BluetoothClientManager` → Manages outgoing connections and device discovery.
        - `BluetoothServerManager` → Handles server socket and incoming connections.

- **BluetoothDeviceManager**
    - Tracks connected devices and their states.
    - Provides a consistent view of the network topology.

Bluetooth Communication

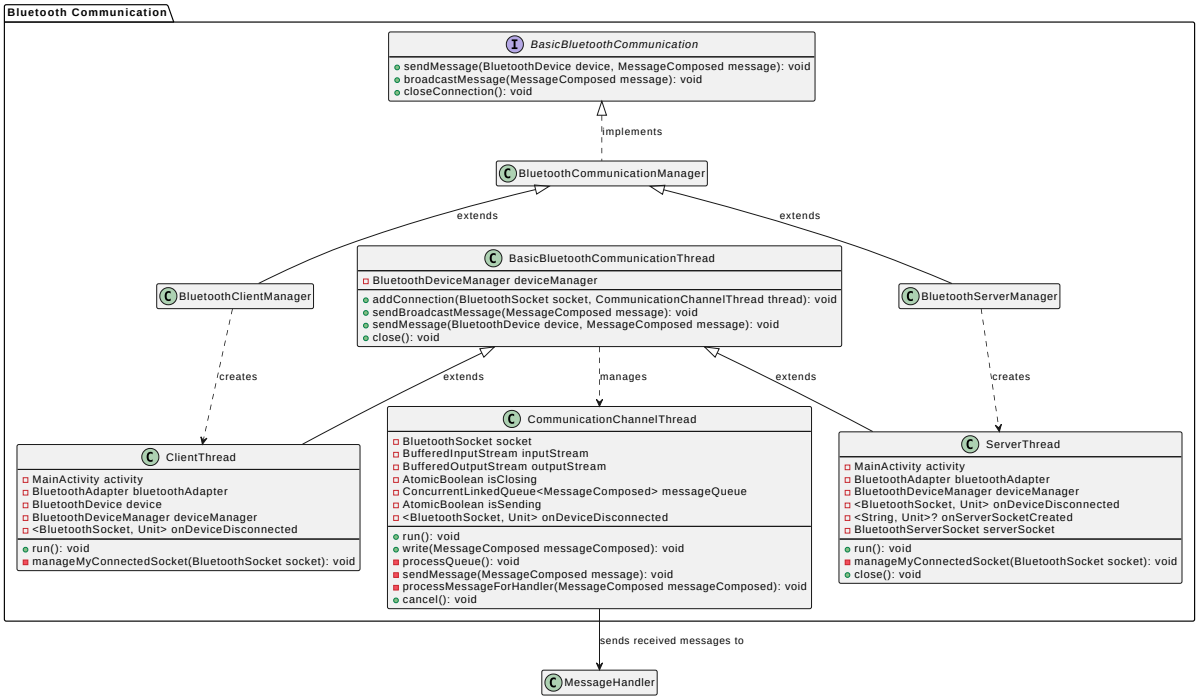


Figure 5. UML Bluetooth Communication Schema.

This diagram details the internal Bluetooth communication flow, illustrating how messages are exchanged between devices. It highlights the management of connections, message queuing, and thread-based operations to ensure efficient data transmission.

Key Components:

- **BasicBluetoothCommunication**
  - Defines core Bluetooth messaging operations, including:
    - `sendMessage(BluetoothDevice device, MessageComposed message)`: Sends a message to a specific device.
    - `broadcastMessage(MessageComposed message)`: Sends a message to all connected devices.
    - `closeConnection()`: Closes active Bluetooth connections.
- **BluetoothCommunicationManager**
  - Implements `BasicBluetoothCommunication` and serves as the base class for managing Bluetooth communication.
  - Defines shared behaviors for client and server implementations.
- **BluetoothClientManager**
  - Extends `BluetoothCommunicationManager`.
  - Handles client-side Bluetooth operations, including connecting to a server.
  - Creates and manages `ClientThread` instances.
- **BluetoothServerManager**
  - Extends `BluetoothCommunicationManager`.
  - Manages server-side Bluetooth operations, including handling incoming client connections.
  - Creates and manages `ServerThread` instances.

- **CommunicationChannelThread**
  - Handles low-level Bluetooth socket communication.
  - Manages:
    - A `BluetoothSocket` for device communication.
    - Input and output streams for message transmission.
    - A concurrent queue for message processing.
  - Key methods:
    - `run()`: Listens for incoming messages and processes them.
    - `write(MessageComposed message)`: Queues and sends outgoing messages.
    - `cancel()`: Closes the communication channel.

- **BasicBluetoothCommunicationThread**
  - Manages multiple Bluetooth connections.
  - Maintains a `BluetoothDeviceManager` for tracking connected devices.
  - Key methods:
    - `addConnection(BluetoothSocket socket, CommunicationChannelThread thread)`: Registers a new connection.
    - `sendBroadcastMessage(MessageComposed message)`: Sends a message to all connected devices.
    - `sendMessage(BluetoothDevice device, MessageComposed message)`: Sends a message to a specific device.
    - `close()`: Closes all active connections.

- **ClientThread**
  - Extends `BasicBluetoothCommunicationThread`.
  - Handles client-side Bluetooth connections and communication.
  - Key responsibilities:
    - Connecting to a Bluetooth server.
    - Managing established connections via `manageMyConnectedSocket(BluetoothSocket socket)`.
    - Retries Connection x3 attempts if the initial connection fails.

- **ServerThread**
  - Extends `BasicBluetoothCommunicationThread`.
  - Manages incoming client connections on the server side.
  - Key responsibilities:
    - Listening for and accepting incoming connections.
    - Managing server sockets through `manageMyConnectedSocket(BluetoothSocket socket)`.
    - Handling connection closures via `close()`.

Bluetooth Model

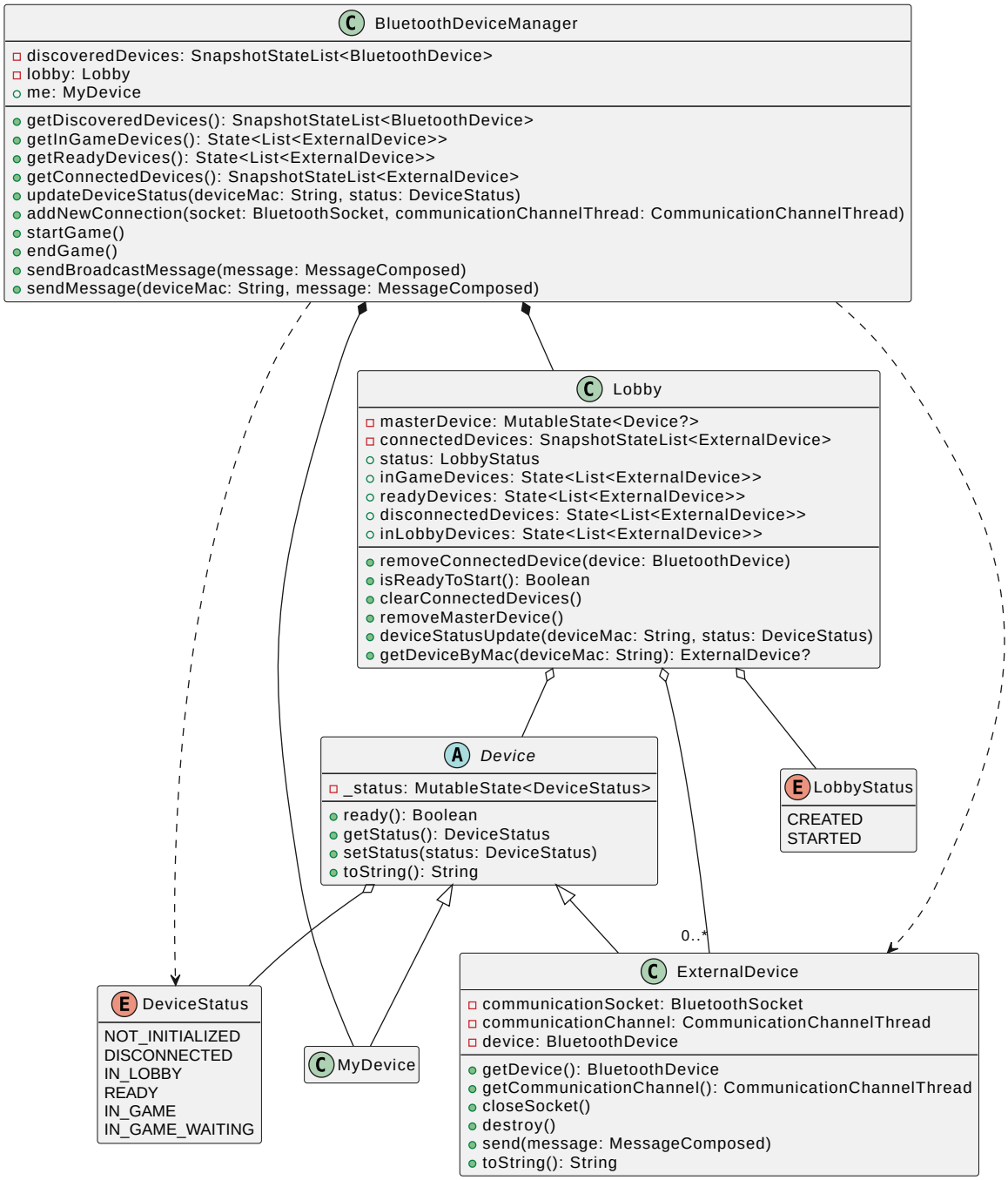


Figure 6. Bluetooth Model Schema.

The Bluetooth model architecture represents the device management and communication structures within the Bluetooth system, tracking the status and relationships between connected devices.

Key Components

Device Statuses

The system uses two enum classes to track the state of devices and lobbies:

- **DeviceStatus**: Defines the possible states of a device in the system:
  - `NOT_INITIALIZED`: Device has not yet been initialized
  - `DISCONNECTED`: Device is disconnected from the session
  - `IN_LOBBY`: Device is connected and present in the lobby
  - `READY`: Device has signaled it's ready to start
  - `IN_GAME`: Device is actively participating in a game

    - `IN_GAME_WAITING`: Device is in a game but in a waiting state
- **LobbyStatus**: Represents the current state of a lobby:
    - `CREATED`: Lobby has been created but isn't fully operational
    - `STARTED`: Lobby is active and operational

Base Classes

- **Device**: An abstract class representing any Bluetooth device in the system
    - Tracks device status using a `MutableState<DeviceStatus>`
    - Provides methods to check if a device is ready and to manage its status

Device Types

- **MyDevice**: Represents the local device (extends `Device`)
    - Provides functionality specific to the user's own device
- **ExternalDevice**: Represents a remote connected device (extends `Device`)
    - Maintains references to:
        - `BluetoothSocket` for communication
        - `CommunicationChannelThread` for message handling
        - `BluetoothDevice` for the actual device information
    - Provides methods for:
        - Accessing device properties
        - Managing the communication channel
        - Sending messages
        - Closing connections and freeing resources

Management Classes

- **Lobby**: Manages the collection of devices participating in a session
    - Tracks:
        - Master device reference
        - List of connected external devices
        - Current lobby status
    - Provides filtered views of devices based on status:
        - `inGameDevices`
        - `readyDevices`
        - `disconnectedDevices`
        - `inLobbyDevices`
    - Handles device management operations:
        - Adding and removing devices
        - Checking if the session is ready to start
        - Updating device statuses
- **BluetoothDeviceManager**: Central management class for all Bluetooth operations
    - Maintains:
        - List of discovered devices
        - Reference to the local device (`me`)
        - Lobby instance
    - Provides methods for:

- Retrieving different device collections
- Managing device statuses
- Establishing new connections
- Starting and ending game sessions
- Broadcasting messages to all devices
- Sending messages to specific devices

Relationships

- `Device` is the parent class for both `MyDevice` and `ExternalDevice`
- `BluetoothDeviceManager` contains a `Lobby` instance
- `BluetoothDeviceManager` manages the local `MyDevice`
- `Lobby` tracks a collection of `ExternalDevice` instances
- `Lobby` references a master `Device`

This model provides a comprehensive structure for managing Bluetooth connections, device statuses, and communication within a multi-device application, particularly one with lobby and game functionality.

Behaviour

Device and Game State Transitions

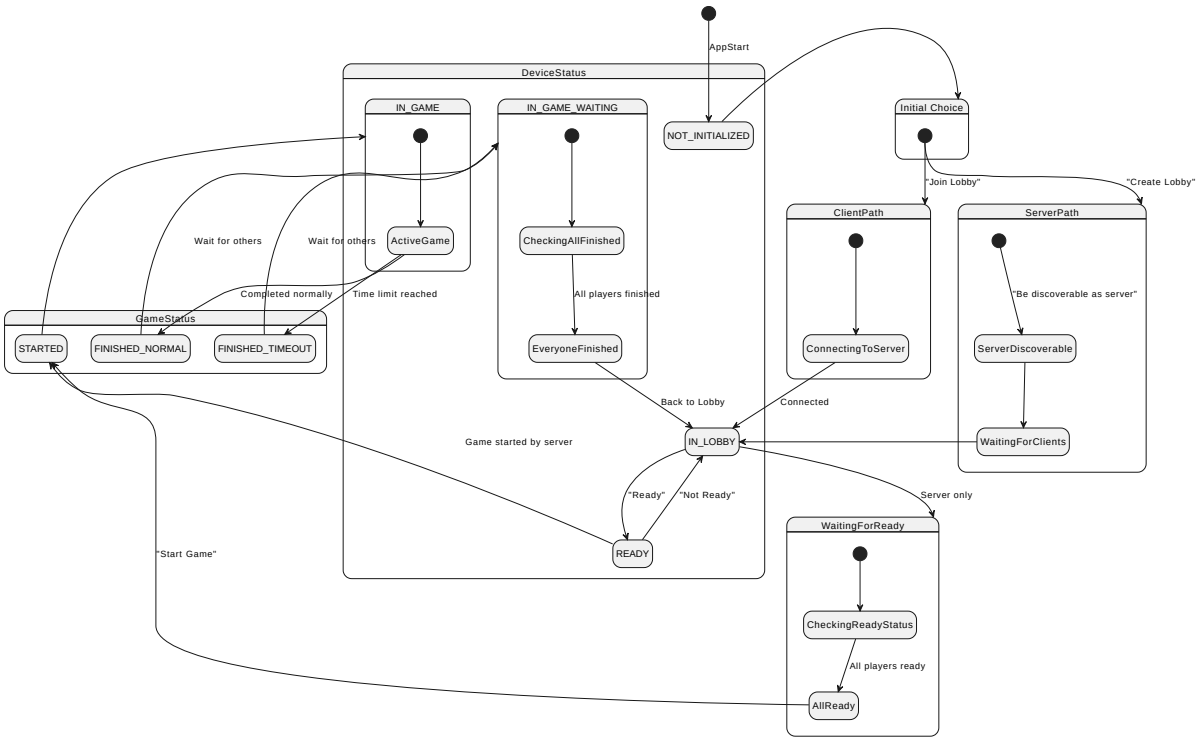


Figure 7. State Diagram showing transitions between device and game states.

Initial States

All devices begin in the `DeviceStatus.NOT_INITIALIZED` state upon application launch. From here, users must make a fundamental choice:

- **Create Lobby** → Transitions to the server path
- **Join Lobby** → Transitions to the client path

Server Path

When following the server path, the device:

1. Becomes discoverable via Bluetooth (`ServerDiscoverable`)
2. Waits for client connections (`WaitingForClients`)
3. Once clients connect, transitions to `DeviceStatus.IN_LOBBY`

The server device is unique in that it simultaneously:

- Acts as the coordinator handling lobby management
- Participates as a player in the game
- Monitors readiness states of all connected devices (`WaitingForReady`)

Client Path

Client devices follow a simpler path:

1. Initiate connection to a discoverable server (`ConnectingToServer`)
2. Upon successful connection, transition to `DeviceStatus.IN_LOBBY`

Lobby States

Once in the lobby (`DeviceStatus.IN_LOBBY`), all devices—including the server—share the same state transitions:

- `DeviceStatus.IN_LOBBY` → `DeviceStatus.READY` (User presses "Ready")
- `DeviceStatus.READY` → `DeviceStatus.IN_LOBBY` (User cancels ready status)

The server monitors these state changes across all connected devices. When all devices reach the `DeviceStatus.READY` state, the server can initiate gameplay.

Game Initiation

Game initiation follows these steps:

1. Server detects all players are ready (`WaitingForReady.AllReady`)
2. Server transitions to `GameStatus.STARTED` (Start Game button pressed)
3. All devices, including the server itself, receive notification and transition to `DeviceStatus.IN_GAME`

Active Gameplay

During `DeviceStatus.IN_GAME`, the following transitions may occur:

- `ActiveGame` → `GameStatus.FINISHED_NORMAL` (Player completes the game normally)
- `ActiveGame` → `GameStatus.FINISHED_TIMEOUT` (Player fails to act within time limit)

Both paths lead to `DeviceStatus.IN_GAME_WAITING`, where devices wait for all other players to finish.

Game Conclusion

Once in `DeviceStatus.IN_GAME_WAITING`:

1. Each device checks if all other players have finished (`CheckingAllFinished`)
2. When all players have completed their games (`EveryoneFinished`), devices return to `DeviceStatus.IN_LOBBY`
3. From here, a new game cycle can begin

Interaction

Message Protocol

Communication between devices in High Noon Blitz follows a structured message protocol where each message has a specific purpose and carries relevant data. This protocol forms the foundation for all device interactions throughout the application lifecycle.

The protocol defines the following standardized message types, each with a unique purpose code:

- **Networking Messages**
  - `NETWORK_ACK (1)`: Basic network acknowledgment with no additional data.
  - `CONFIGURATION (4)`: Contains the complete list of client MAC addresses (`devices`) connected to the sender.
  - `INFO_SHARING (6)`: Shares essential connection information including:
    - `masterAddress`: MAC address of the current coordinator
    - `UUID`: Unique identifier of the sending device
    - `status`: Current readiness state of the sending device
- **Lobby Control Messages**
  - `READY (3)`: Signals player readiness status via a boolean (`ready`).
  - `CONSISTENCY_CHECK_REQUEST (8)`: Initiates network topology verification (no additional data).
  - `CONSISTENCY_CHECK_REPLY (9)`: Reports a device's connections via a list of MAC addresses (`macAddresses`).
- **Election Protocol Messages**
  - `ELECTION_REQUEST (5)`: Initiates leader election, carrying the sender's unique identifier (`ID`).
  - `ELECTION_ACK (7)`: Acknowledges an election request with a simple acknowledgment (`ACK`).
  - `ELECTION_INGAME_FINISHED (13)`: Signals the completion of an in-game election process (no additional data).
- **Game Control Messages**
  - `GAME_START (10)`: Signals the beginning of gameplay (no additional data).
  - `END_GAME (12)`: Reports game completion, carrying the player's reaction time (`timeDifference`).
  - `BACK_TO_LOBBY_GAMELIST (11)`: Requests return to the lobby after game completion (no additional data).

This message protocol ensures consistent communication across all network interactions, from initial connection through gameplay and recovery from failures.

Client-Server Connection Pattern

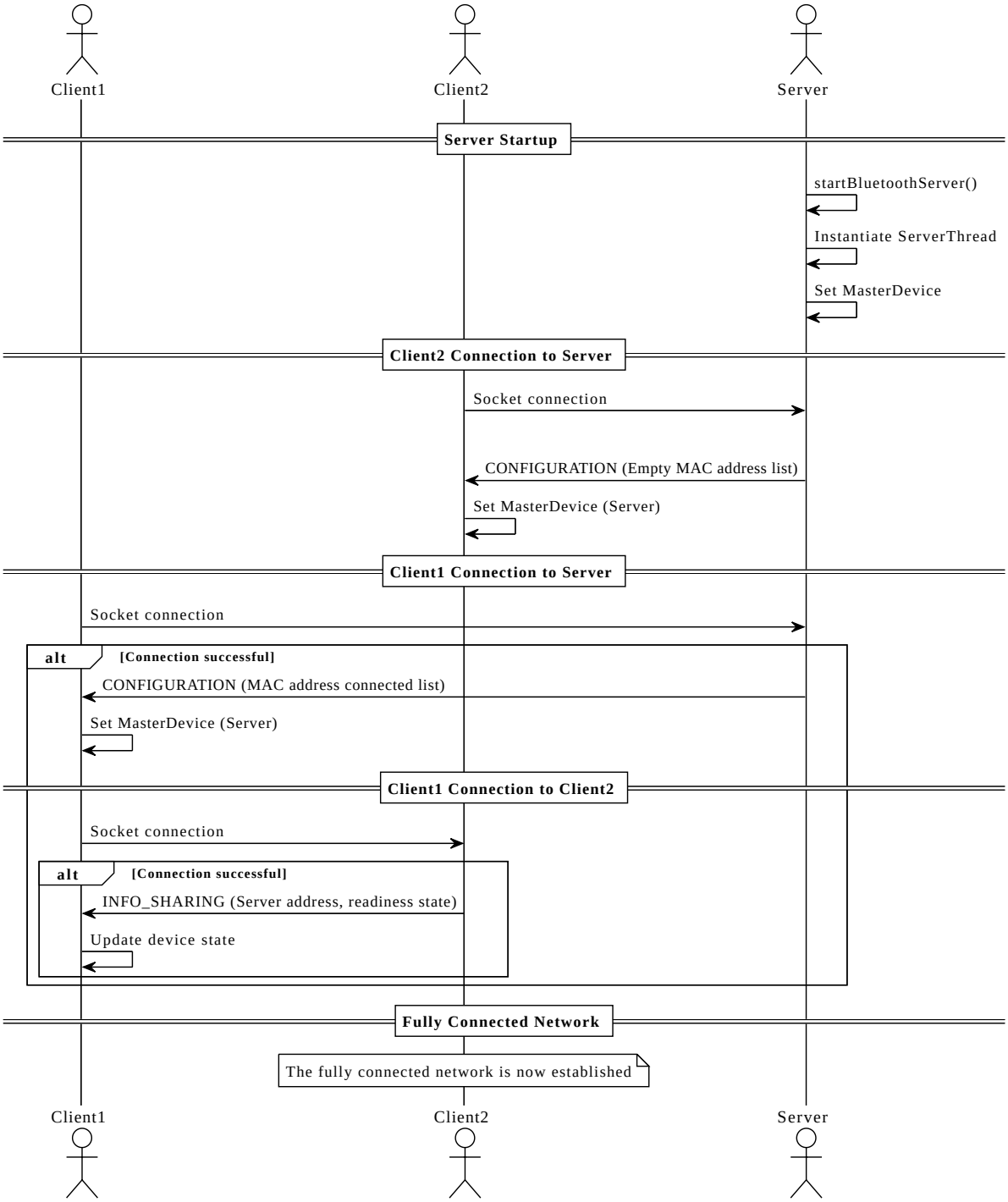


Figure 8. Client-Server Connection Sequence showing how clients connect through the server.

The client-server connection pattern follows these steps:

1. **Server Initialization**:
    - The server calls `startBluetoothServer()`
    - Creates a `ServerThread` to listen for incoming connections
    - Sets itself as the `MasterDevice` in the network

2. **First Client Connection** (Client2):
    - Client2 establishes a direct socket connection to the server
    - Server sends a `CONFIGURATION` message with an empty MAC address list (since Client2 is the first client)
    - Client2 sets the server as its `MasterDevice`
3. **Second Client Connection** (Client1):
    - Client1 establishes a direct socket connection to the server
    - Server sends a `CONFIGURATION` message with the current connected MAC address list (including Client2)
    - Client1 sets the server as its `MasterDevice`
    - Client1 then connects to Client2 directly using the MAC address from the list
    - Client2 shares server information and readiness state with Client1 via `INFO_SHARING` message
4. **Network Completion**:
    - A fully connected network is established where all devices are connected to each other

This approach ensures a robust network where:

- The server maintains the authoritative list of all connected devices
- Each client is aware of all other clients in the network
- Direct connections exist between all pairs of devices

Client-Client Connection Pattern

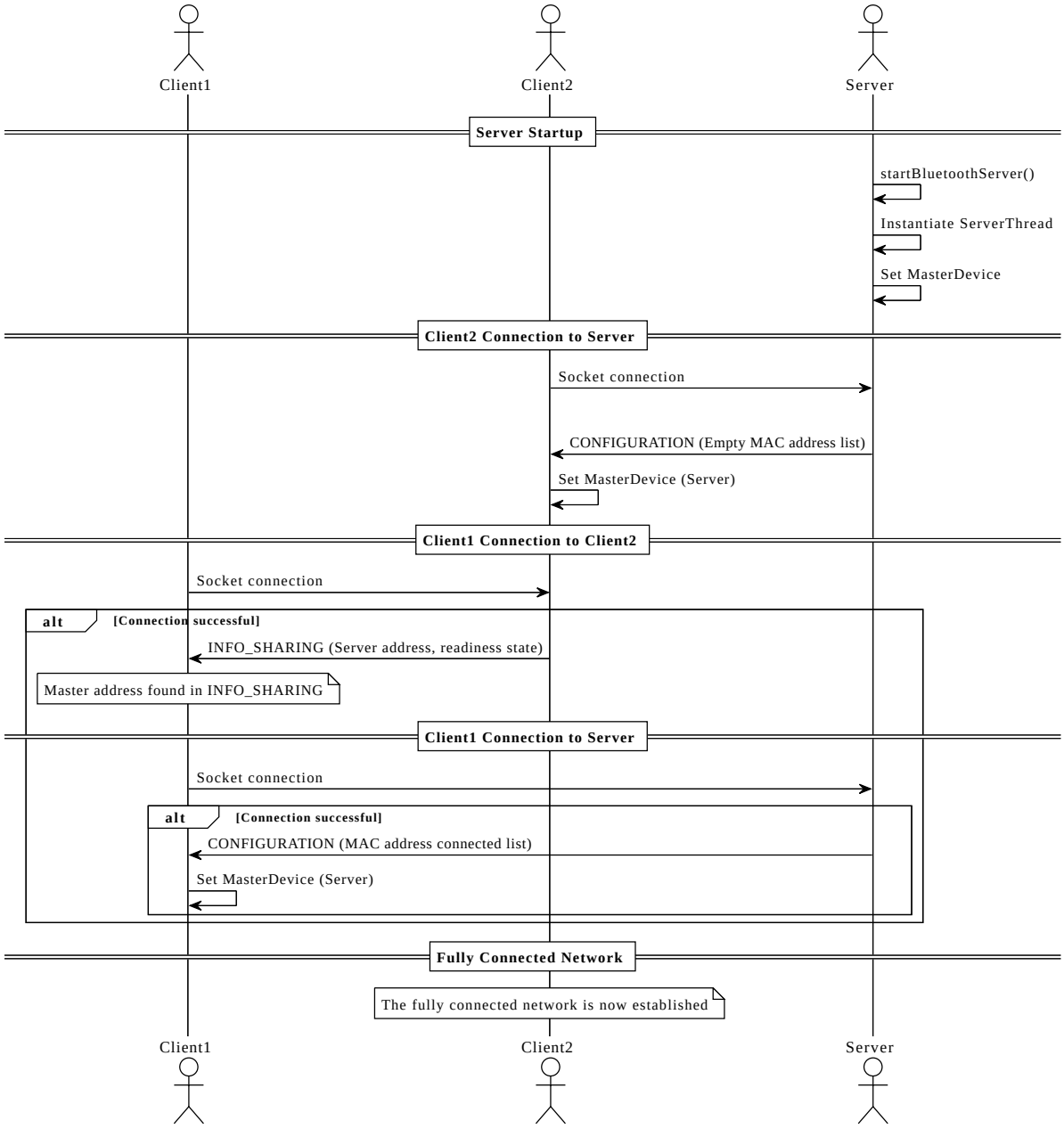


Figure 9. Client-Client Connection Sequence showing alternative connection establishment.

The client-client connection pattern provides an alternative path for network formation:

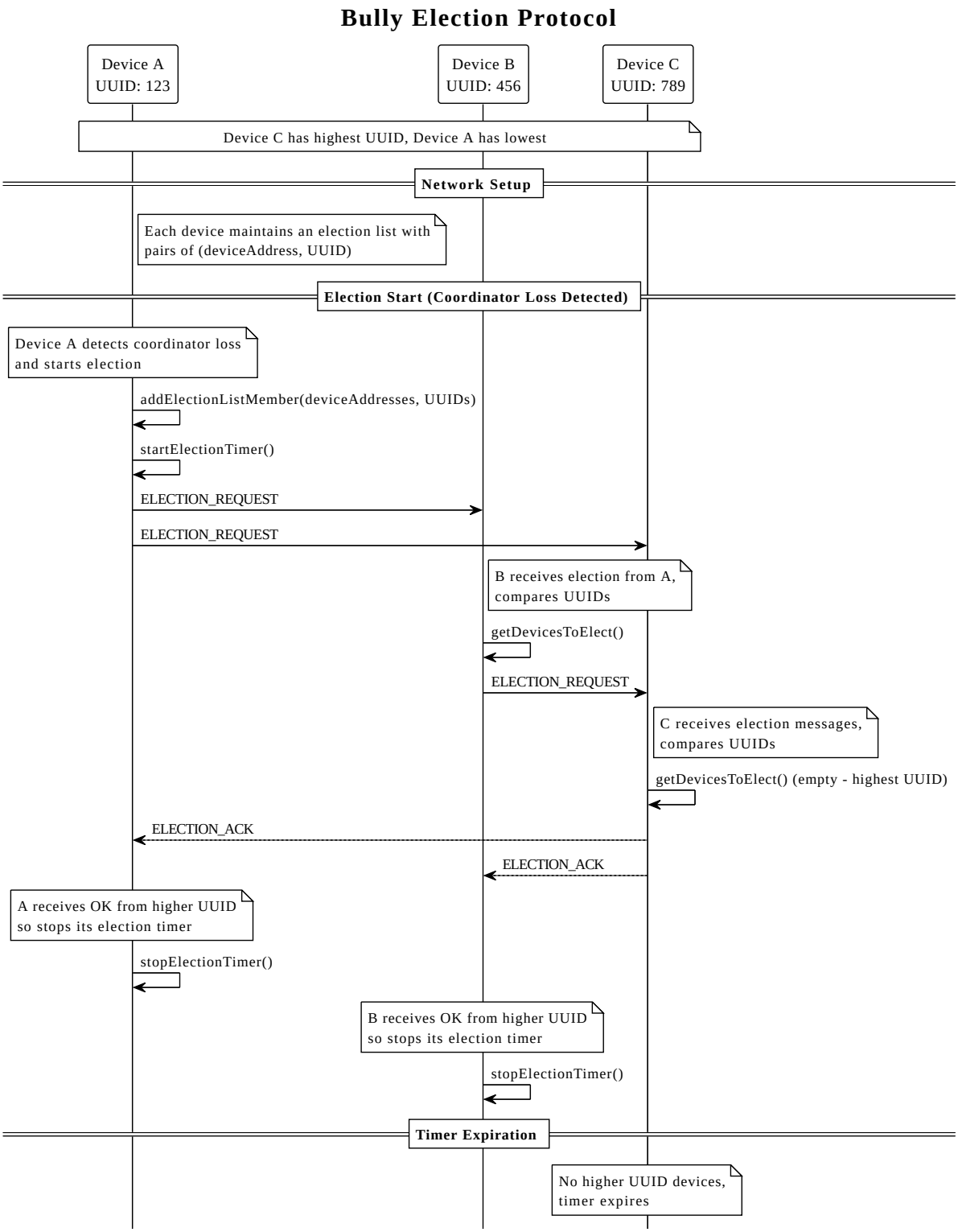
1. **Server Initialization**: Same as in the client-server pattern
2. **First Client Connection** (Client2):
   - Client2 establishes a direct socket connection to the server
   - Server sends a `CONFIGURATION` message with an empty MAC address list
   - Client2 sets the server as its `MasterDevice`
3. **Client-to-Client Discovery** (Client1 to Client2):
   - Client1 discovers and connects directly to Client2 first (rather than the server)
   - Client2 sends an `INFO_SHARING` message containing the server address and readiness state
   - Client1 learns about the server from this information
4. **Delayed Server Connection**:

- Client1 uses the information from the `INFO_SHARING` message to connect to the server
- Server sends a `CONFIGURATION` message with the current MAC address list
- Client1 sets the server as its `MasterDevice`

5. **Network Completion**:
   - The fully connected network is established through this alternative path

This pattern demonstrates the system's flexibility in handling different discovery and connection sequences, ensuring network integrity regardless of the order in which devices connect.

Fault Tolerance: Bully Election Protocol



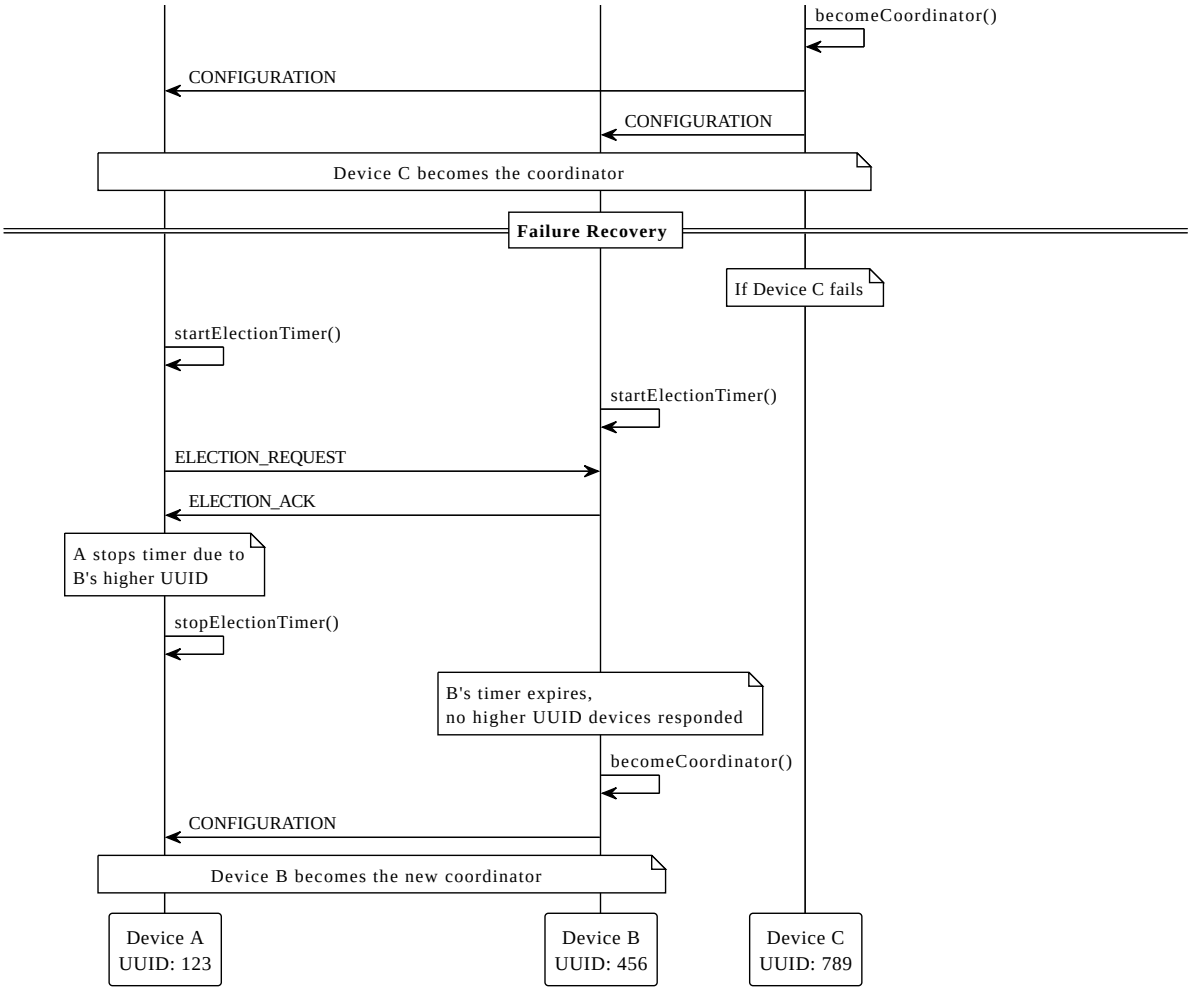


Figure 10. Bully Election Protocol

Our system implements an adapted version of the Bully Election Protocol optimized for Bluetooth networks. Unlike traditional implementations that rely on heartbeat mechanisms, we detect coordinator failures through Bluetooth connection exceptions.

When any device joins, it creates and maintains its own election list containing the MAC addresses and UUIDs of all devices it connects to. As every device connects directly to every other device in our fully connected topology, each device independently builds and updates its election list when new connections are established or existing ones are terminated.

When the coordinator (server) fails, the system employs the Bully Election Protocol:

1. **Election Initiation**:
    - A device detects coordinator loss through a Bluetooth connection exception
    - It starts an election by consulting its election list
    - It sends `ELECTION_REQUEST` messages to all other devices with higher UUIDs
2. **Election Processing**:
    - Devices receiving election requests compare UUIDs
    - Devices with higher UUIDs respond with `ELECTION_ACK`
    - Devices receiving acknowledgments from higher-UUID devices stop their election timers
3. **Coordinator Selection**:

- The device with the highest UUID (whose timer expires without receiving any higher-UUID response) becomes the new coordinator
- The new coordinator sends `CONFIGURATION` messages to all devices

4. **Recovery Example**:

- If the highest-UUID device (Device C) fails, the election process repeats
- The next highest UUID device (Device B) becomes the new coordinator

This robust election protocol ensures that even if the original server/coordinator fails, the network can automatically reorganize itself and continue functioning under a new coordinator.

Network consistency checker

The Network Consistency Checker ensures that your Bluetooth network is fully connected before allowing gameplay to begin. It verifies that every device in the network has established direct connections with every other device, preventing scenarios where some devices may be unreachable to others.

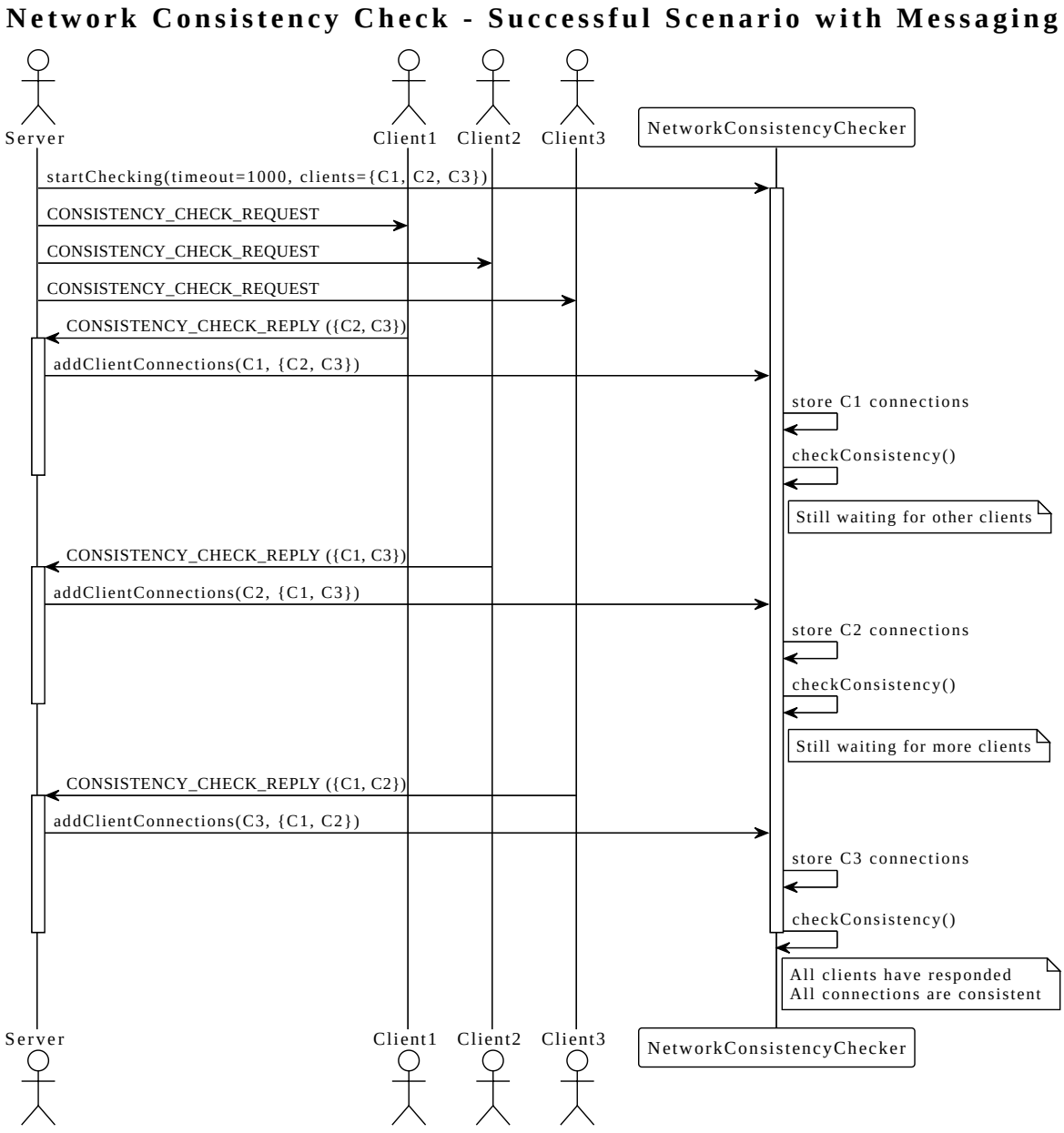


Figure 11. Network consistency checker

The network consistency check follows these steps:

1. **Initiation**:
    - The server (coordinator) calls startChecking() with:
        - A timeout duration (typically 1000ms)
        - The set of all client MAC addresses
    - Single-client networks automatically pass the check
2. **Request Phase**:
    - The server broadcasts a CONSISTENCY_CHECK_REQUEST message to all clients
    - Each client receives this request and prepares to respond
3. **Response Collection**:
    - Each client sends a CONSISTENCY_CHECK_REPLY with its list of connected devices
    - The server processes each reply
        - These connections are stored in the clientConnections map
4. **Validation Logic**:
    - After each response, checkConsistency() is called to validate the current state
    - The check verifies that:
        - All clients have reported their connections
        - For each client, its reported connections match all other clients in the network
        - There are no asymmetric connections (if A connects to B, B must connect to A)
5. **Result Determination**:
    - **Success Case**: All clients have responded and all connections are symmetric
    - **Failure Cases**:
        - Timeout occurs before all clients respond
        - Connection asymmetry is detected
        - Missing connections between clients
6. **Completion**:
    - The LobbyController receives the result through the callback
    - On success, game start proceeds with informClientsAndStartGame()
    - On failure, an error message is displayed to users

Implementation Details

In this project the documentation has been generated within a GitHub environment and is then delivered via GitHub Pages.

The process is integrated into a continuous integration pipeline that uses Dokka to produce the documentation. You can view the complete output at the URL.

Self-assessment

The quality of the produced software was evaluated through a qualitative assessment framework designed to verify compliance with the specified requirements. Our evaluation approach focused on four key dimensions:

1. User Acceptance Testing

We conducted informal testing sessions with two distinct user groups:

- **Experienced Players:** Familiar with the game mechanics, these users evaluated advanced features.
- **First-Time Users:** These participants focused on the intuitiveness and learning curve.

Feedback was collected through observation and discussion, emphasizing user satisfaction and the identification of pain points.

2. Cross-Device Compatibility

Manual testing was performed across a diverse range of Android devices, including:

- 2× Redmi Note 7
- OnePlus 10 Pro 5G
- OnePlus 5T
- Moto G 2015
- Moto G200 5G

Custom ROMs were installed as needed to ensure that all devices were running Android 12 or newer. The testing focused on visual rendering, touch responsiveness, and overall performance across different screen sizes and hardware capabilities.

3. Network Resilience Testing

Real-world testing of network behavior was conducted by:

- Deliberately disconnecting devices during gameplay.
- Observing reconnection behavior for both server and client roles.
- Noting any game state inconsistencies after connection restoration.

4. Performance Evaluation

Rather than using precise measurements, performance was qualitatively assessed by examining:

- Overall application responsiveness during gameplay.
- Bluetooth connection stability among various device combinations.
- The effectiveness of the reaction time calculation approach.

Test Suite Overview

The following tests validate the core logic and functionalities of our application:

Network Consistency Checking

- **NetworkConsistencyCheckerTest**
  Verifies that all clients in the network are properly interconnected. This test evaluates both successful consistency scenarios and various failure modes, including timeouts.

Leader Election Algorithm

- **BullyElectionManagerTest**
  Tests the implementation of the Bully Algorithm for electing a leader device. It ensures the correct management of the election candidate list and the overall election process.

Lobby Management

- **LobbyTest**
  Covers the game lobby functionality, including device management, game state transitions, and messaging. It checks the proper assignment of the master device, the management of connected devices, and the initiation of gameplay.

Game Model

- **InGameModelTest**
  Validates the core game state management by verifying proper state transitions and the tracking of game results.

Device Management

- **ExternalDeviceTest**
  Ensures proper handling of external Bluetooth devices, focusing on communication and resource management.

- **DeviceTest**
  Tests the basic device state management functionality.

Deployment Instructions

To install and launch the produced software artifacts on a completely virgin environment, follow these steps:

1. **Environment Setup:**

- Ensure that the latest version of Android Studio is installed.
- Install the Android SDK, and verify that your system meets the minimum requirements for running Android Studio.
- Confirm that Java (JDK 11 or higher) is available on your system.

2. **Project Configuration:**

- The project is managed with Gradle. Open the project in Android Studio, which will automatically import the Gradle configuration.
- Check the `build.gradle` file to find the following configuration:
  - `minSdkVersion 24`
  - `targetSdkVersion 34`

- Review the AndroidManifest.xml to ensure all necessary permissions and components are defined as per project requirements.

3. **Building and Running:**

- In Android Studio, click on the “Build” menu and select “Make Project” to compile the source code.
- Once the build is successful, connect an Android device or start an emulator.
- Run the project using the “Run” menu. Android Studio will install the APK onto the selected device/emulator and launch the application.

We also have an APK file generated, downloadable on URL.

Usage Examples

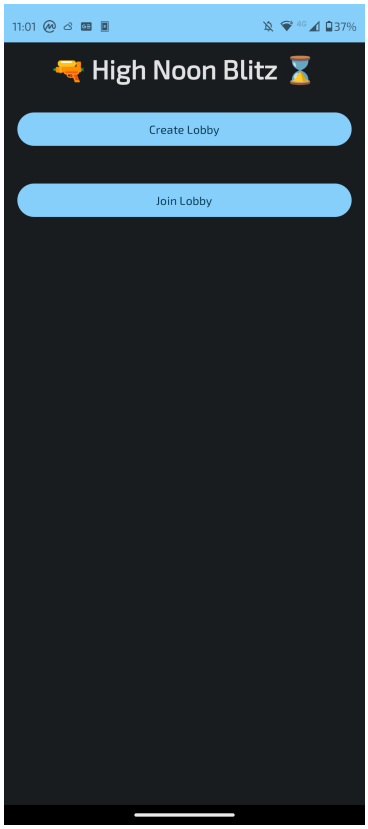


Figure 12: Start Role Choice

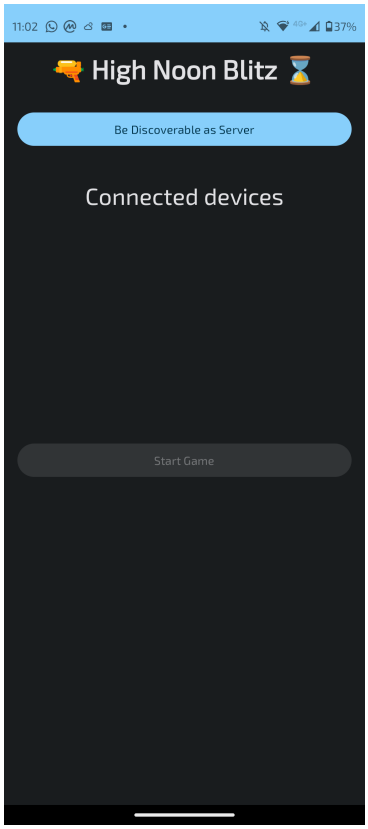


Figure 13: Start of Server

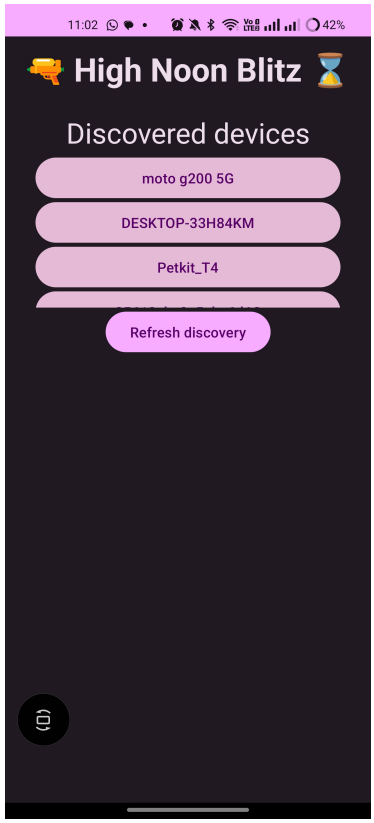


Figure 14: Discovery made by client

Figure 15: Lobby without Ready

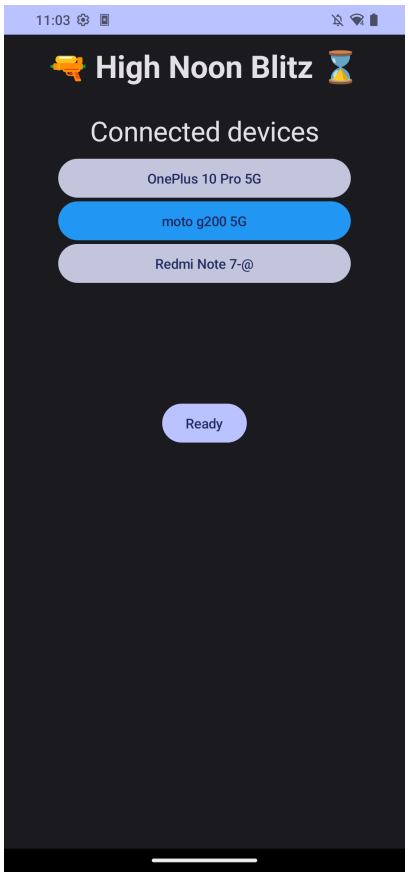


Figure 16: Lobby without Ready

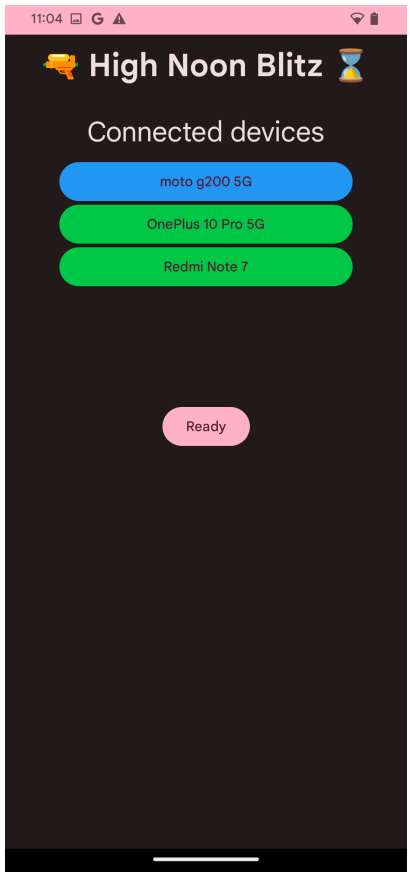


Figure 17: Lobby with Ready

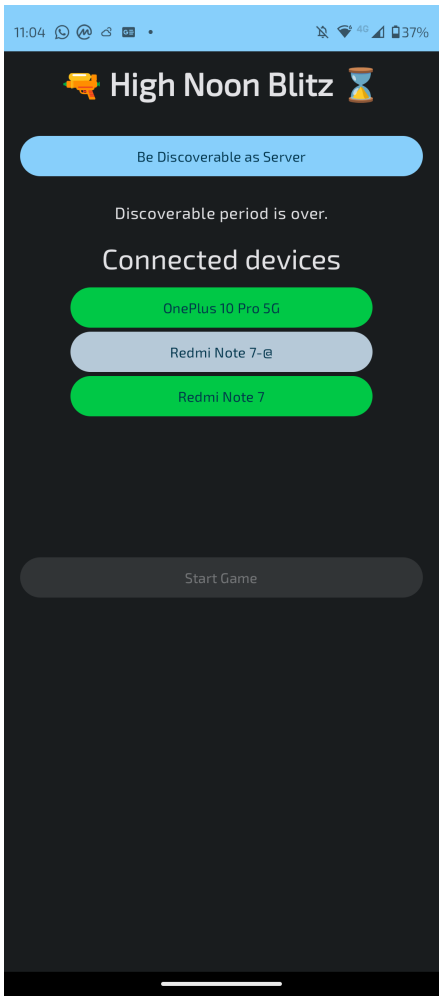


Figure 18: Lobby from Server

Figure 19: Election Process

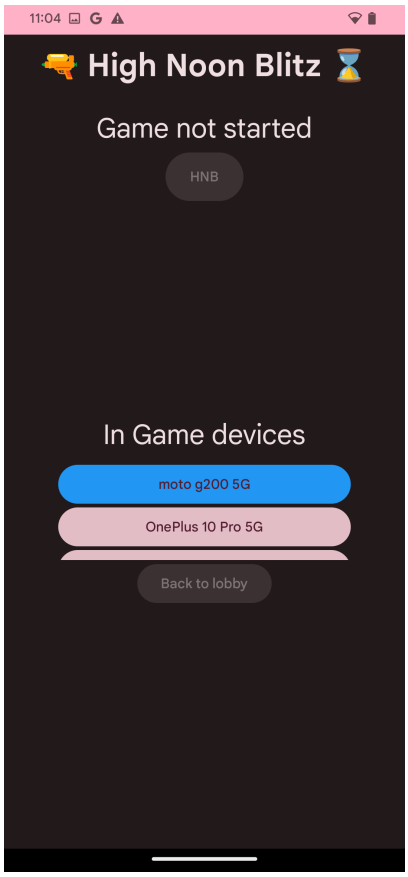


Figure 20: Start of the Game

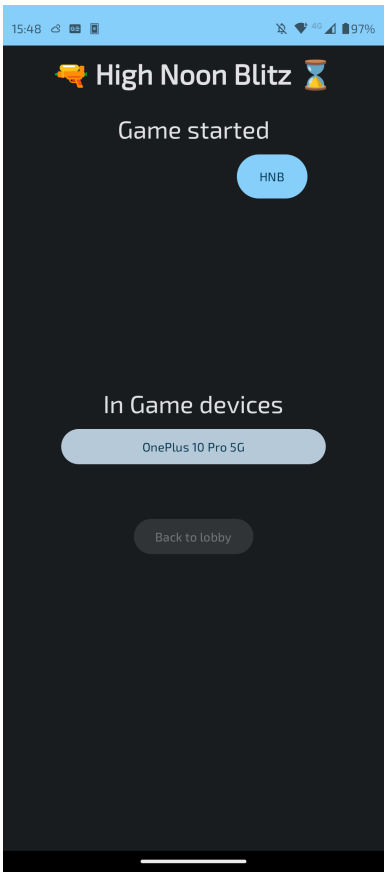


Figure 21: Game ongoing

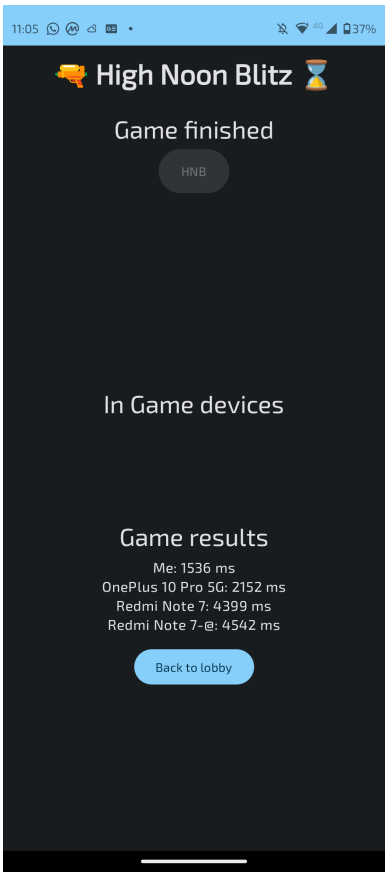


Figure 22: Game finished

Conclusion

This project was a challenging yet rewarding experience, allowing us to apply theoretical knowledge in a practical setting. We successfully implemented a Bluetooth-based multiplayer game, focusing on network management, device communication, and user experience. The project not only enhanced our technical skills but also provided valuable insights into the complexities of real-time communication and game design. We are quite proud of the result, altough being the project being start at the first semester of University, and being finished during the last one, has been discontinued for some time and different knowledge and skills were applied to the project, related to the course we were attending.

Future works

The project has some problems regarding the connection between devices. Sometimes the connection between the phones fails, even if the devices are close to each other and the app retries to do it. This is a problem that we have not been able to solve, and that we would like to work on in the future.

Sometimes also the navigation occurs in some minor problems, which would be nice to fix, altough it doesn't alterate the game experience too much.

What did we learned

We learned how to better code in Kotlin, and how Bluetooth works in Android. We also learned the challenges and outcomes of developing a fully connected network and the fault-tolerance algorithms related to it. Also the world of permissions in Android was explored.