Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Ingegneria Informatica

Prova Finale Ingegneria del Software

Communication protocol

Prof. Giampaolo Cugola

Francesco Rausa	**Federico Paludetti**
Pietro Poggi	**Riccardo Passolunghi**

Introduction

This document aims to provide a comprehensive overview of the communication protocol established between the client and server across the different supported connection methods. To enhance understanding of the interaction sequences, UML sequence diagrams are included.

1 General

At startup, each client can choose between two connection modes: **RMI** or **TCP**. Depending on the chosen mode, the client initializes either an RMI client or a TCP client.

In the RMI mode, the client exports its interface and communicates with an RMI server that manages all registered client interfaces.

In contrast, the TCP mode involves creating a socket-based connection to a TCP server. Upon connection, the server delegates communication tasks to a dedicated MultiClientHandler instance for each connected client, enabling concurrent handling of multiple TCP clients.

1.1 Server Coordination via ServersHandler

In order to support both RMI and TCP connections in a unified and consistent way, the project introduces a central class named ServersHandler. This component is responsible for starting both the TCP and RMI servers as independent threads, providing them with all the necessary shared resources to manage clients and game instances.

The ServersHandler creates and shares the following objects:

- A shared instance of **GamesHandler**, which manages the creation and coordination of multiple games and it works as a dispatcher to route commands from all possible clients (uniquely associated with a token) to a specific game.
- A **global token map** (ConcurrentHashMap<String, VirtualView>) that associates each authenticated client (regardless of connection type) with a corresponding VirtualView, which is responsible for sending game updates to the player.
- A **list of disconnected clients** associated clients (uniquely associated with a token (ArrayList<String>) to support reconnection mechanisms and disconnection handling.

This architecture ensures that both TCP and RMI clients are integrated into the same game flow, and that the **lower-level game logic (handled by GameController) can directly push updates to clients** using the VirtualView abstraction — without knowing or caring whether the client is connected via TCP or RMI.

1.2 The VirtualView Class

The VirtualView class is a server-side component responsible for sending game updates to a specific client.

Each VirtualView instance is associated with a single player and contains the necessary communication references (socket or RMI callback) to deliver messages.

Importantly, the VirtualView is **unidirectional**: it is only used to **send data from the server to the client**. Client commands are not routed through the VirtualView. Instead, clients send commands along with a token, which the server uses to authenticate and dispatch the command to the correct GameController via the central GamesHandler.

This design ensures that:

- The game server maintains full control over outgoing information.
- Incoming commands from clients are securely authenticated and routed independently of the output pipeline.
- The game logic remains protocol-agnostic, while the communication strategy is encapsulated within the VirtualView.

The VirtualView is created when a player joins a game and it is deleted when the client quits the game. So even if the client disconnects the VirtualView continues to receive updates from the game (in particular it receives small deltas each time). In this way the reconnection is guaranteed in a large percentage. It will be enough to authenticate the reconnection and “reconnect” the VirtualView to the client and the client will be able to receive back all the necessary deltas. More on this later (see chapter 4.2).

2 SOCKET CONNECTION

The heart of TCP management is the TCPServer class executed by the ServersHandler on a separate thread. The class listens on a specific port through the *startTcp()* invoked by the ServersHandler to accept new incoming connections. Each time a client connects, the TCPServer creates a dedicated MultiClientHandler object dedicated to that specific connection, executing it on a separate thread.

MultiClientHandler has a loop that constantly reads messages sent by the client. This includes acknowledging messages such as login, reconnect, lobby but also messages for managing pings to keep the connection alive. If the client stops responding, the MultiClientHandler notices it and, after having disconnected the VirtualView from the client, signals the disconnection to the GamesHandler.

When creating a game or joining an existing one, a globally unique token is created and sent to the client.

MultiClientHandler also implements the GhListener interface to asynchronously receive lobby updates and send it to the client if requested.

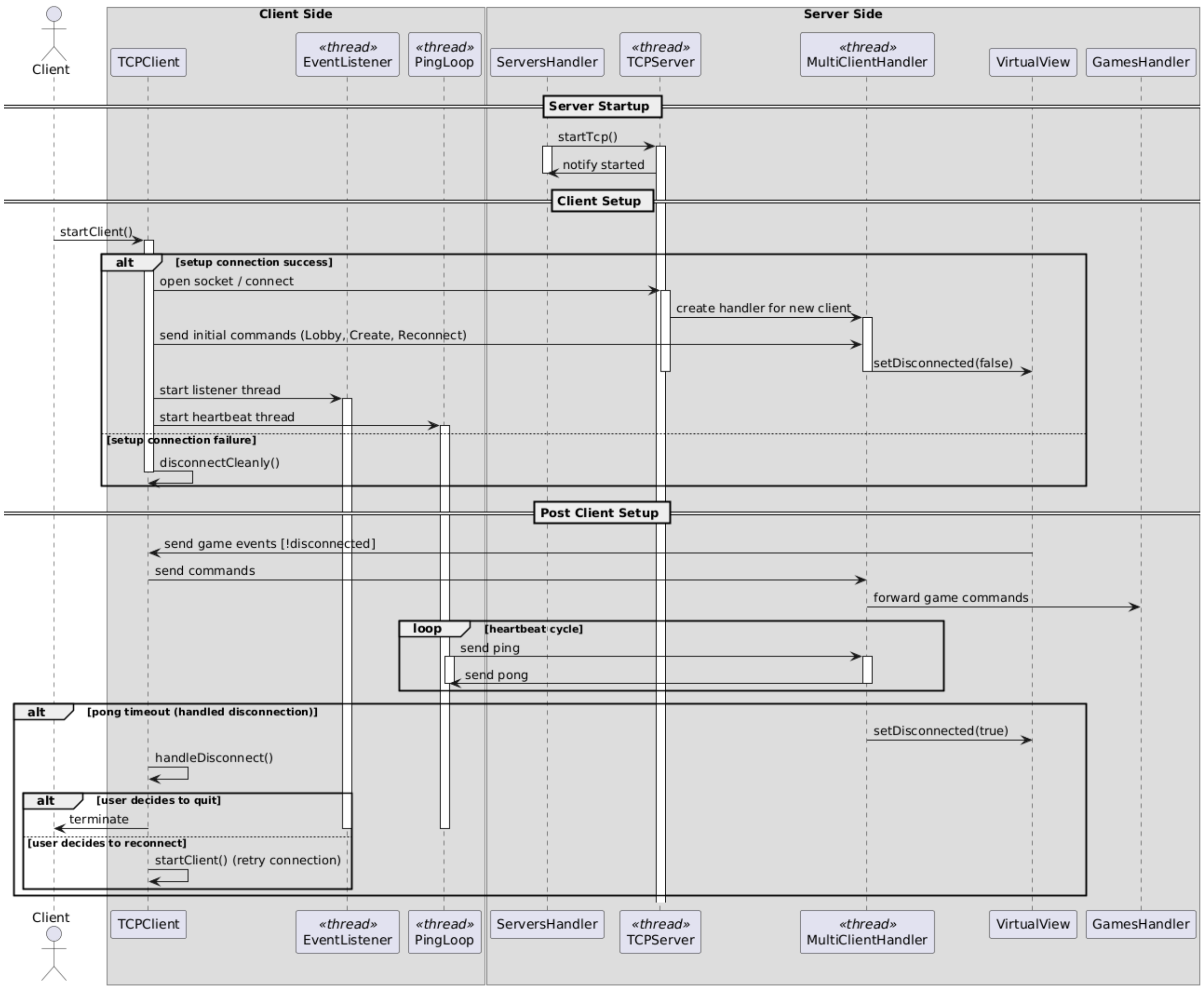
The TCPClient class is responsible for establishing and maintaining a TCP connection between the client application and the game server. The connection process begins when the client calls the *startClient()* method. Inside this method, the client first attempts to establish a connection to the server by invoking the *setup()* method.

During the setup phase, the client tries to open a socket to the server's address and port. If the socket is successfully created, the client initializes input and output streams tied to this socket. If any step fails, the client will attempt to disconnect gracefully.

Once the connection is successfully established, the client marks itself as connected and starts two background threads: one for listening to incoming messages from the server (*EventListener*), and another to periodically send heartbeat messages (*PingLoop*).

If the client does not receive a “pong” response within a defined timeout period, it assumes the connection is lost and triggers the disconnection process.

Disconnection is handled carefully to close the socket and stop the background threads



In this diagram the reconnection interaction is not shown properly (see chapter 4.2).

3 RMI Connection

The RMI server handles multiple client connections by keeping references to the client’s ClientInterface objects, which are exported by the clients. At startup, the server registers itself and starts an executor that regularly iterates over all clients to check their connection status using timestamps updated whenever clients call the *receivePong* method. For lobby events, the server asynchronously creates a thread to send updates to the clients, saving their ClientInterface references for future communication. During the login process, the server generates a unique token for each game and calls a method in the GamesHandler, which queues the login request. Meanwhile, the RMI server marks the client as pending and associates it with the token, but the token is not immediately sent to the client. The GamesHandler processes the login asynchronously, and if successful, it calls a method on the RMI server to remove the client from pending, send the token to the client, and link a VirtualView to that client. All client commands sent through RMI are routed via the GamesHandler and placed into queues in the GameController for asynchronous execution by worker threads. This design makes the RMI connection effectively asynchronous, enabling the server to manage many clients and game logic concurrently without blocking.

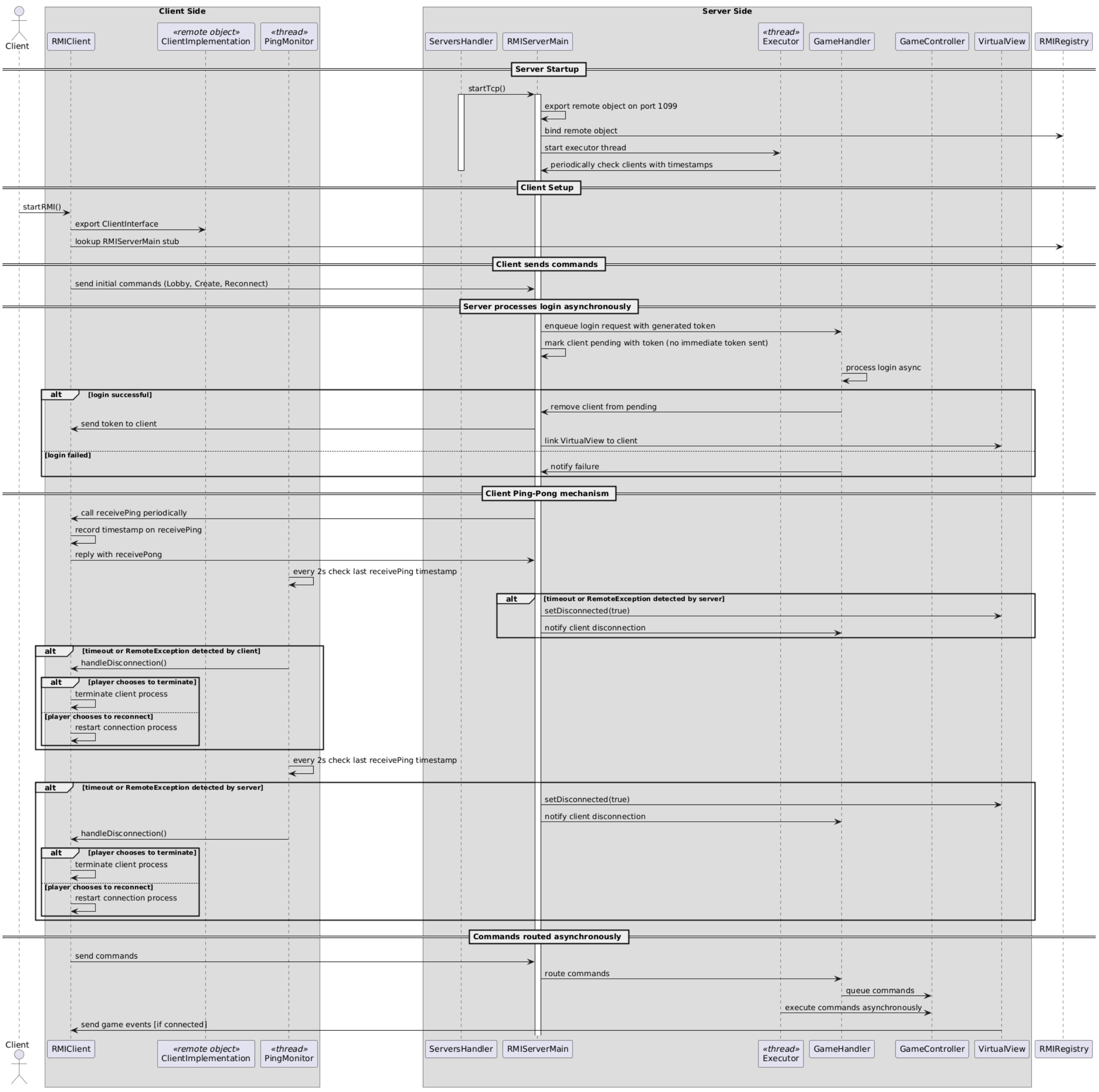
The RMI client first exports its remote object (implementation of ClientInterface). Then, it enters a setup phase in which it attempts to retrieve the server-side remote object by performing a registry lookup on “CommandReader”.

If the lookup succeeds, the client enters the main input loop, where it can send commands such as “Lobby”, “Join” or “Create”. Once the player joins or creates a lobby, a *ping monitor* is started to check the connection health.

The RMI client implements the ClientInterface, which defines remote methods such as receiveEvent and receivePing. When the client receives a *receivePing* call from the server, it records the timestamp and replies with a *receivePong* call.

The ping monitor runs periodically and checks whether the time since the last *receivePing* exceeds a certain threshold. If too much time has passed without a ping, or if a RemoteException is thrown during communication, the client assumes the connection has been lost and takes appropriate action.

This mechanism allows the server to detect unresponsive clients and allows clients to handle unexpected disconnections gracefully.


Client Side
Server Side
Client
RMIClient
«remote object» ClientImplementation
«thread» PingMonitor
ServersHandler
RMIServerMain
«thread» Executor
GameHandler
GameController
VirtualView
RMIRegistry
Server Startup
startTcp()
export remote object on port 1099
bind remote object
start executor thread
periodically check clients with timestamps
Client Setup
startRMI()
export ClientInterface
lookup RMIServerMain stub
Client sends commands
send initial commands (Lobby, Create, Reconnect)
Server processes login asynchronously
enqueue login request with generated token
mark client pending with token (no immediate token sent)
process login async
alt [login successful]
remove client from pending
send token to client
link VirtualView to client
[login failed]
notify failure
Client Ping-Pong mechanism
call receivePing periodically
record timestamp on receivePing
reply with receivePong
every 2s check last receivePing timestamp
alt [timeout or RemoteException detected by server]
setDisconnected(true)
notify client disconnection
alt [timeout or RemoteException detected by client]
handleDisconnection()
alt [player chooses to terminate]
terminate client process
[player chooses to reconnect]
restart connection process
every 2s check last receivePing timestamp
alt [timeout or RemoteException detected by server]
setDisconnected(true)
notify client disconnection
handleDisconnection()
alt [player chooses to terminate]
terminate client process
[player chooses to reconnect]
restart connection process
Commands routed asynchronously
send commands
route commands
queue commands
execute commands asynchronously
send game events [if connected]

In this diagram the reconnection interaction is not shown properly (see chapter 4.2).

4 Playing the Game

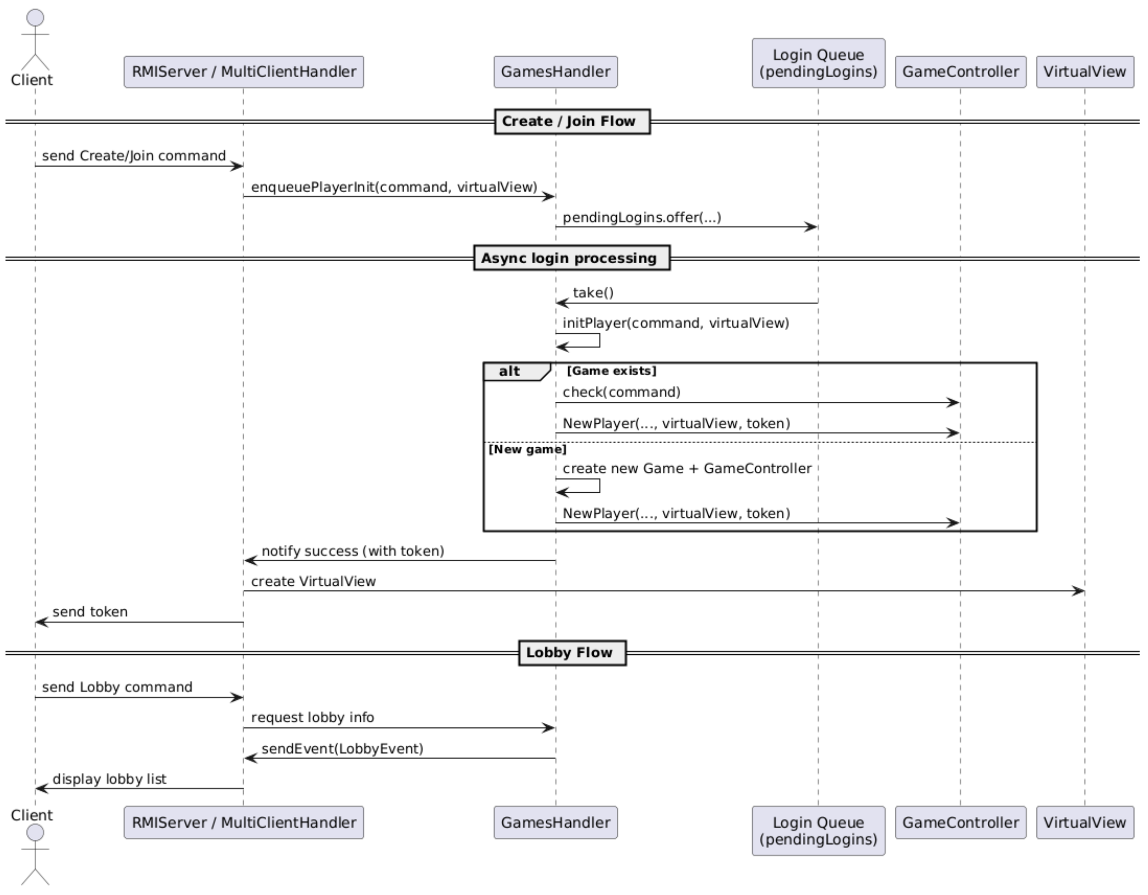
The initial connection between a client and the server is handled by a *connection-specific proxy*: if the client uses RMI, the first interaction occurs via the RMIServer; if the client uses TCP, it connects through the MultiClientHandler. Regardless of the protocol, the client can perform three main actions during the initial phase: **Create**, **Join**, and **Lobby**.

When a client sends a **Create** command to start a new game, the request is handled *asynchronously*. The command is enqueued in a centralized login queue (pendingLogins) managed by the GamesHandler. Once processed successfully, the client receives a **token** asynchronously, which serves as a unique session identifier and can later be used for reconnection. The same logic applies to the **Join** command: the client requests to join an existing game, and if accepted, a new VirtualView is instantiated and associated with the client. The entire process remains asynchronous and transparent to the client.

The **Lobby** command allows a client to request a list of currently available games. Both RMIServer and MultiClientHandler act as listeners to the GamesHandler, which maintains a global view of all ongoing games and lobbies. This means clients can receive up-to-date lobby information asynchronously from the server, regardless of the connection method.

In both the **Create** and **Join** workflows, if the command is successful, a VirtualView is always created and bound to the client. This VirtualView will act as a communication bridge between the server-side game logic and the client, forwarding events and updates throughout the game session.

The **Reconnect** command is also supported but will be covered in detail in the next section



4.1 Client Disconnection Handling

When a client disconnects, the proxy component—either the RMIServer or the MultiClientHandler, depending on the communication protocol—detects the event and immediately invokes the playerDisconnected(String token) method on the GamesHandler. This method is responsible for orchestrating the disconnection logic across all ongoing games. Using the provided session token, the GamesHandler identifies the corresponding GameController and notifies it that the associated player is no longer connected.

Upon detecting the disconnection, the proxy "detaches" the client from its associated VirtualView. From that point on, the VirtualView stops sending updates to the client, but it continues—as it always does—to record all state deltas internally. This behavior ensures that the entire sequence of game updates remains available for when the client reconnects.

From that moment on, the player is considered temporarily inactive. The GameController continues the game flow as normal, but will automatically apply default or fallback decisions for that player whenever an input is required. These default commands ensure that the game proceeds smoothly even in the absence of explicit client input, maintaining game consistency and preventing delays.

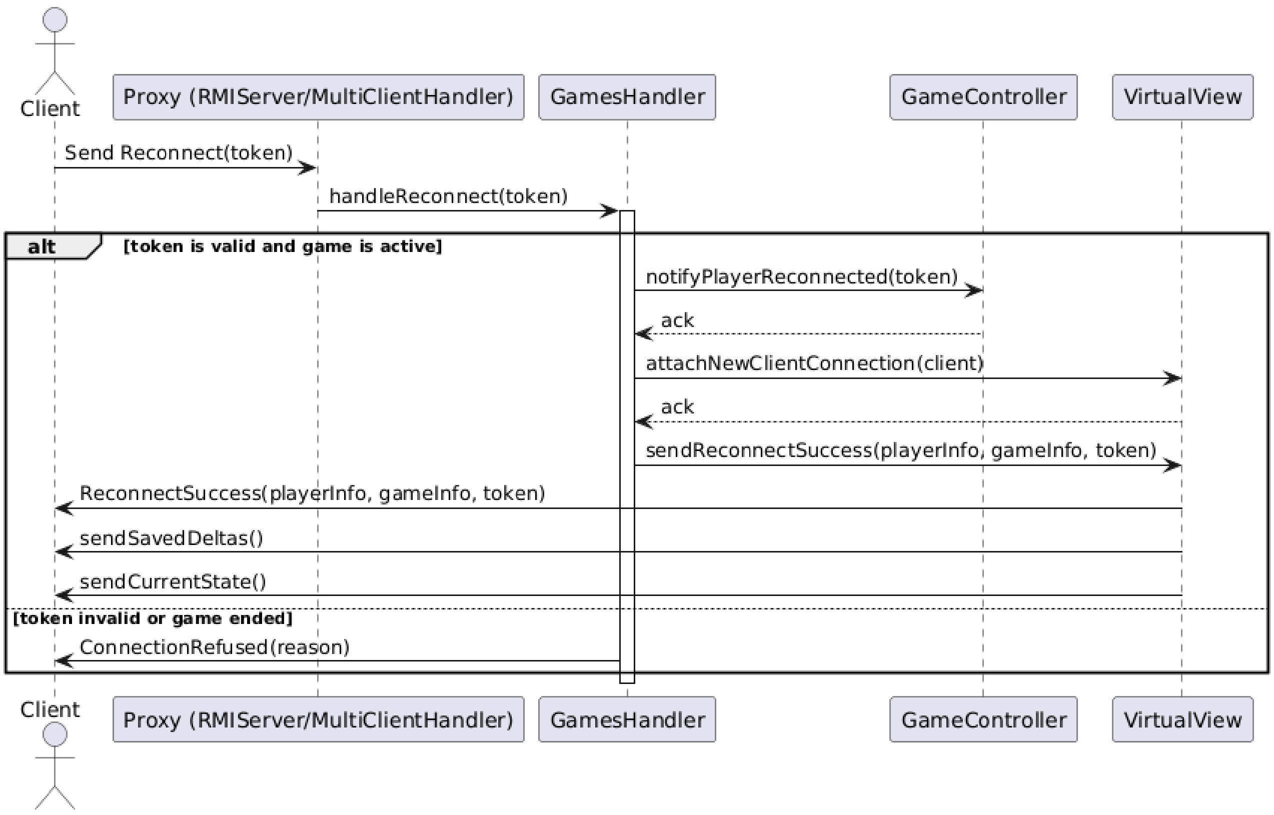
4.2 Client Reconnection

Clients may attempt to reconnect after a disconnection, either due to temporary network issues or because the application was manually closed and later reopened. Reconnection is fully asynchronous and handled through the same proxy component (RMIServer or MultiClientHandler) that manages the initial connection. If the client remained active and simply lost connection, reconnection attempts may be made automatically in the background, transparently using the session token previously stored in memory.

In the case where the client has been fully closed and reopened, the user is prompted to manually input the session token. Regardless of the connection method used previously, the client is free to switch protocols during reconnection. For example, a client originally connected via RMI can reconnect using TCP, and vice versa, with no impact on the session.

Once the reconnection request reaches the proxy, it forwards the request and token to the GamesHandler, which asynchronously validates the token, ensuring it corresponds to an active game session. If the validation succeeds, the GamesHandler notifies the appropriate GameController, which updates the player's state to mark them as reconnected and suspends any default actions that were being executed in their absence.

Upon successful reconnection, the client receives a confirmation message containing relevant session details, such as the player’s name, game ID, game level, and token. Most importantly, the proxy reattaches the client to their previously associated VirtualView. This triggers a **reconnect phase**, during which the VirtualView replays all recorded deltas accumulated during the disconnection period. This ensures the client is fully synchronized with the current state of the game, and gameplay can resume seamlessly.


Client
Proxy (RMIServer/MultiClientHandler)
GamesHandler
GameController
VirtualView
Send Reconnect(token)
handleReconnect(token)
alt
[token is valid and game is active]
notifyPlayerReconnected(token)
ack
attachNewClientConnection(client)
ack
sendReconnectSuccess(playerInfo, gameInfo, token)
ReconnectSuccess(playerInfo, gameInfo, token)
sendSavedDeltas()
sendCurrentState()
[token invalid or game ended]
ConnectionRefused(reason)
Client
Proxy (RMIServer/MultiClientHandler)
GamesHandler
GameController
VirtualView

4.3 “Gameplay” commands

All remaining client commands—including standard in-game actions and the Quit command—are processed asynchronously. Once received by the proxy component (RMIServer or MultiClientHandler), these commands are forwarded to the GamesHandler, which routes them to the appropriate GameController based on the session token.
Each GameController maintains an internal queue of pending commands. These queues are continuously monitored by dedicated worker threads that asynchronously consume and execute commands in the order they are received. This design ensures that game logic is processed concurrently without blocking the main server loop, and it allows multiple games to progress in parallel.

The Quit command, like any other, is handled in the same fashion: it is enqueued, processed by the GameController, and leads to the proper cleanup and disconnection of the player if necessary.

The communication model between the server and client is based on two types of messages: **events** and **commands**.
Events are server-to-client messages used to update the client's view. In the case of a TCP connection, these events are serialized using JSON via Jackson and sent through the socket. In contrast, for RMI, events are delivered by directly invoking methods on the client-side stub, allowing immediate access to updated information.

On the other hand, commands are client-to-server messages that represent user actions. These are implemented using the **Command Pattern**: each command encapsulates the data required for execution and exposes a single execute() method. In TCP, commands are serialized objects transmitted over the network, while in RMI, they are sent as method calls on a shared remote interface.

The client itself is designed to be **stateless**—it does not hold any authoritative game data such as the player board or current resources. Instead, it maintains only a minimal internal state representing the current interaction phase (e.g., waiting for input, choosing a card, responding to an event). All actual game data is managed server-side and pushed to the client through events when necessary.