Network Pattern Documentation

Andrea Santarsiero, Andrea Pianini, Lorenzo Stani, Luca Sartori

June 22, 2025

Contents

1 Lobby Phase: Creating and Joining a Match

This phase covers everything that happens *before* the actual game starts. Two transport layers are available—RMI and plain sockets—but the application–level protocol (the `Action` objects) is identical.

Workflow

a) **Bootstrap**: the client chooses either RMI or Socket and opens the underlying connection.

b) **Session Registration**: the client sends `RegisterRMISessionAction` or `RegisterSocketSessionAction`. The server replies with a **session token** τ.

c) **Create Match (optional)**: the lobby owner issues `CreateMatchAction`; the server instantiates a `GameContext` and returns its identifier.

d) **Join Match**: every participant sends `ConnectToGameAction` carrying $\langle username, \tau, matchId \rangle$. The server validates τ and either accepts or rejects.

Sequence Diagrams

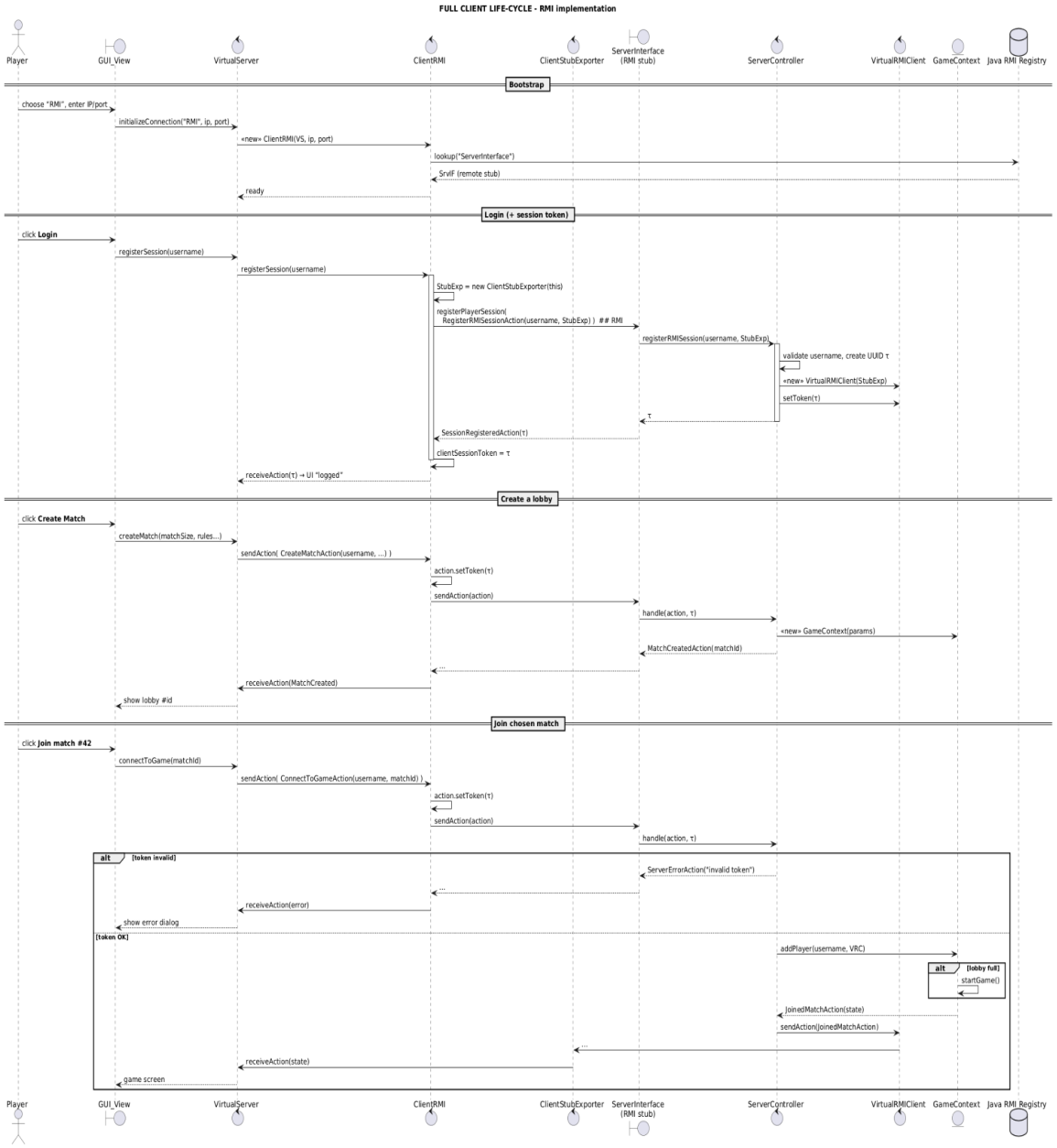

Figure 1: Creating/Joining a match — *RMI variant*

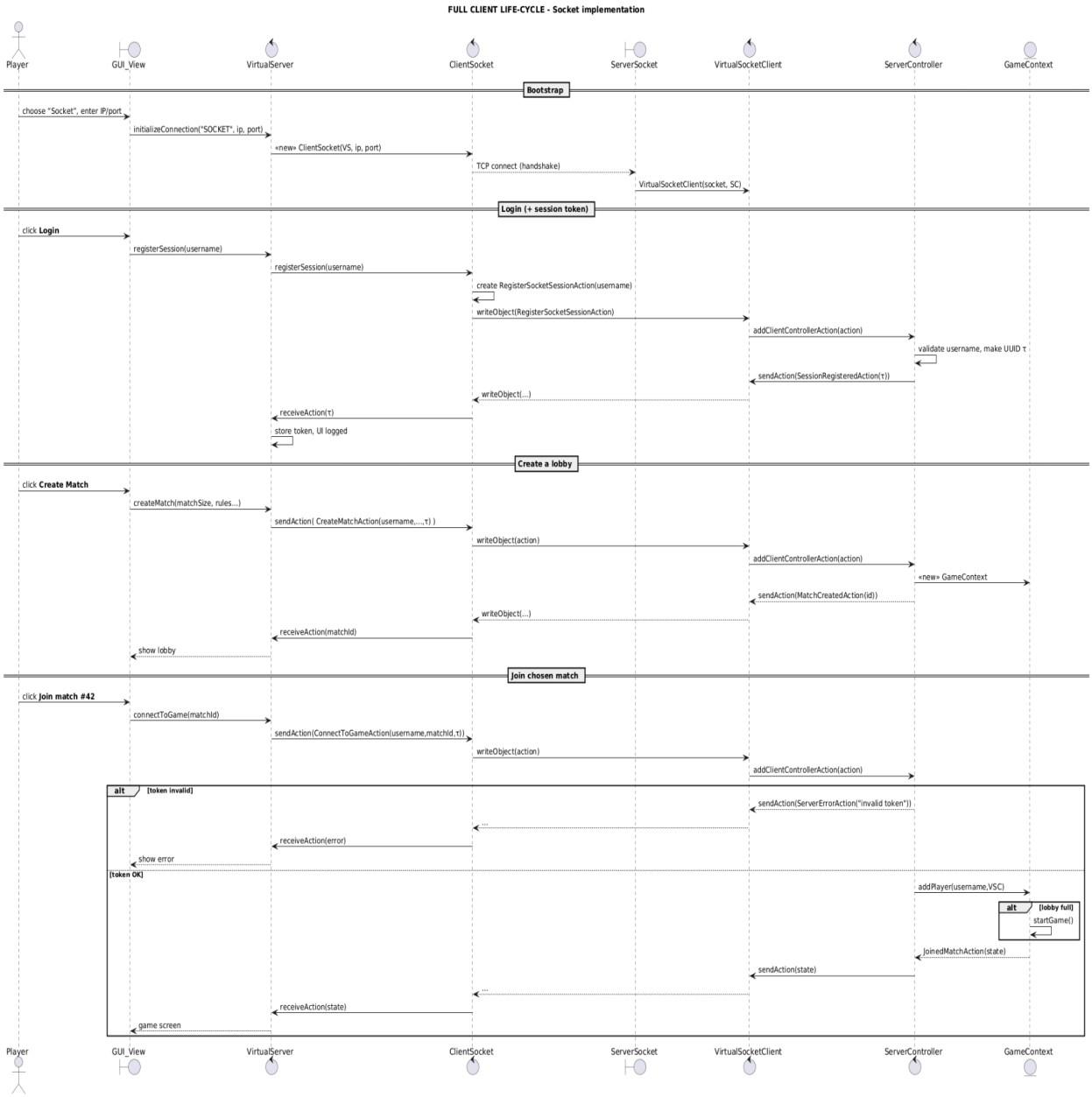

Figure 2: Creating/Joining a match — *Socket variant*

2 Executing an In-Game Action

Once the game has begun, every client action follows the *"Command→Validate→Broadcast"* pattern.

Workflow

a) The view constructs a concrete `ClientGameAction` and delegates it to the `VirtualServer`.

b) The transport layer (RMI or Socket) wraps the action with the stored token τ and forwards it to the server.

c) **ServerController** validates τ and game state, then calls the appropriate method on **GameContext**.

d) The resulting server-side action (or error) is multicasted back to all interested clients.

We illustrate this with the `GetFreeShipCardAction` / `SendFreeShipCardAction` pair.

Sequence Diagrams

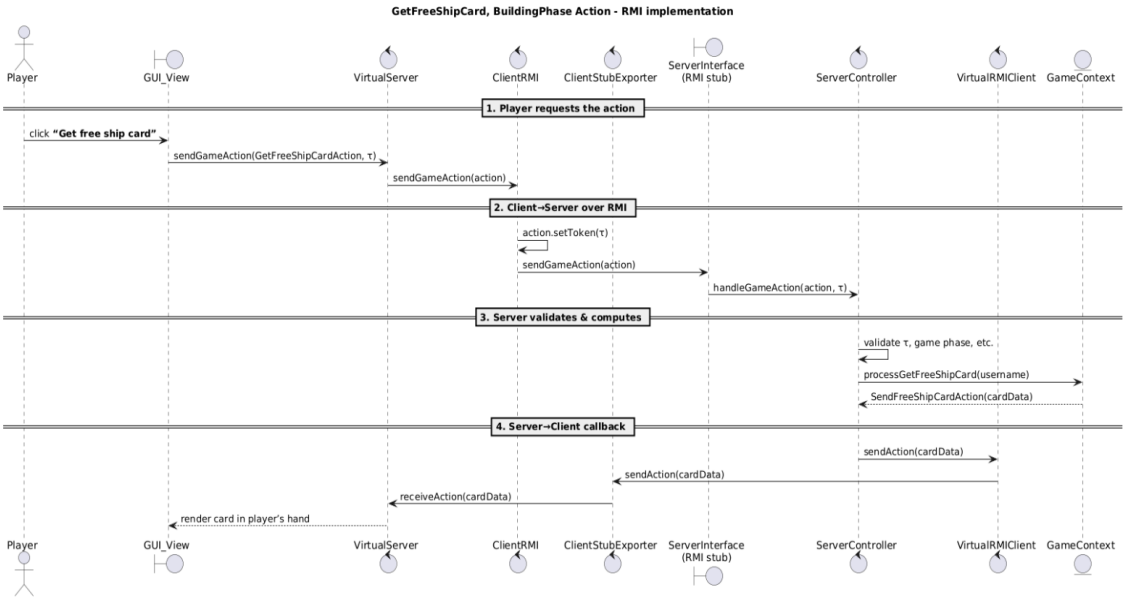

Figure 3: Taking an action — *RMI variant*

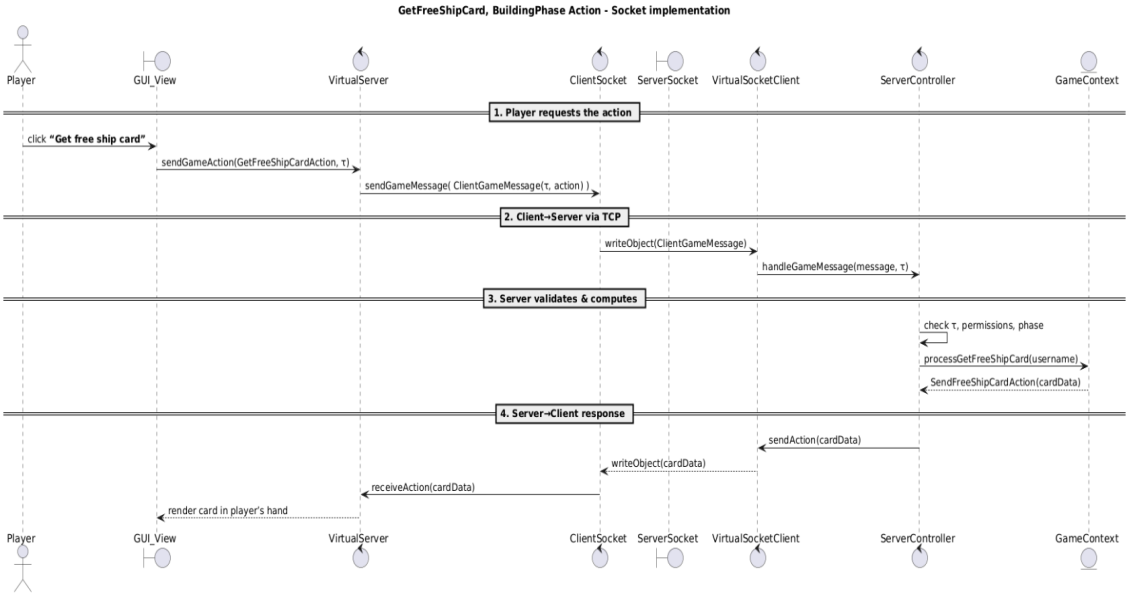

Figure 4: Taking an action — *Socket variant*

3 Drawing a Card

Card extraction is a specialised *ClientGameAction*. We use `GetAdventureCardAction` (client) and `SendAdventureCardAction` (server) as the concrete example.

Workflow

a) Player triggers `GetAdventureCardAction`; the client attaches τ and transmits it.

b) Server validates, pops the top card from the appropriate deck, and encapsulates it in `SendAdventureCardAction`.

c) The card data is pushed to the requesting client (and possibly to all observers) for UI rendering.

Sequence Diagrams

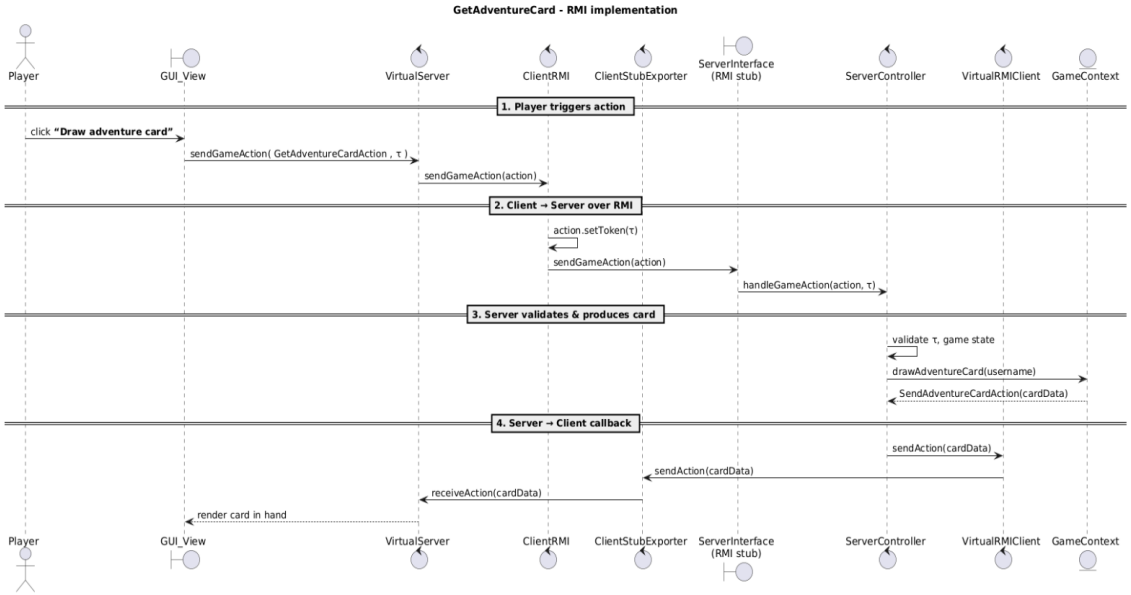

Figure 5: Drawing a card — *RMI variant*

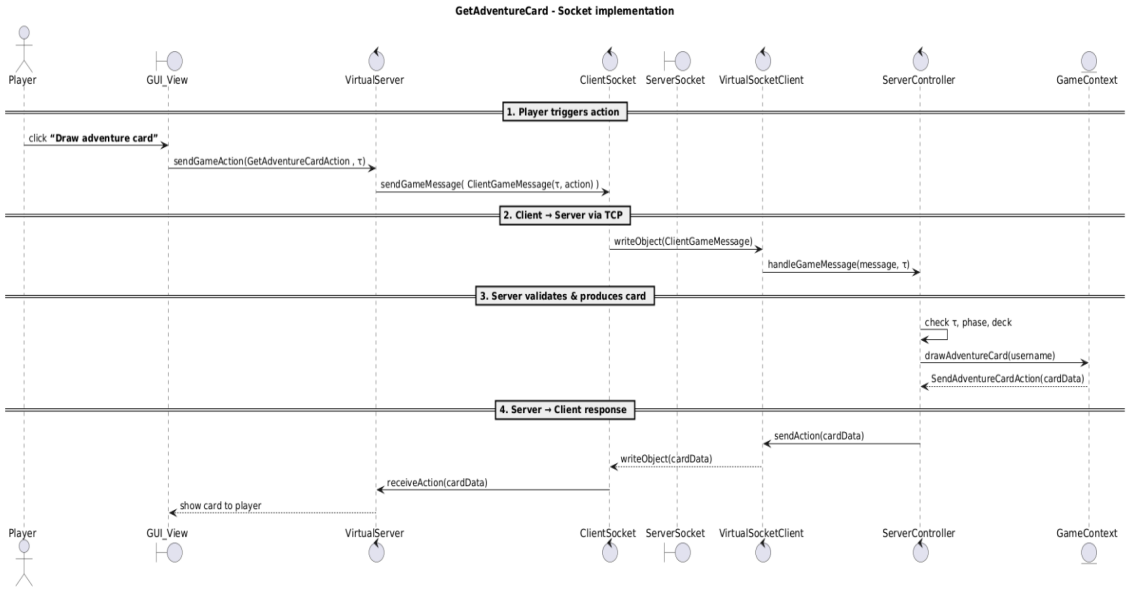

Figure 6: Drawing a card — *Socket variant*

Conclusion

The project employs a command pattern: all high-level messages are plain Java objects implementing `ClientControllerAction` or `ClientGameAction`. RMI and Socket layers differ only in *how* those objects cross the network; the semantics remain identical, which keeps the server logic (and most of the client) strictly independent from the chosen communication technology.