Protocollo di comunicazione (basato sulla peer-review 2)

Samuele Allegranza, Matteo Arrigo, Lorenzo Battini, Federico Bulloni
Gruppo AM13

27 giugno 2024

1 Introduzione

Per gestire l'aggiornamento delle `View` di ogni giocatore in seguito a una modifica dello stato della partita (`GameModel`) abbiamo applicato il design pattern `Observer` (che noi abbiamo chiamato Listener per semplicità). Nella nostra implementazione un `GameListener`, che corrisponde in maniera univoca a un giocatore/client, viene aggiunto alla lista di giocatori (`ListenerHandler`) interessati a ricevere aggiornamenti sullo stato della partita (`GameModel`). In questo modo ogni volta che lo stato della partita viene modificato, la classe GameModel si occupa di chiamare il corrispondente metodo di `notify()` di ListenerHandler che a sua volta richiama il corretto metodo di `update()` di GameListener. Essendo quest'ultima un'interfaccia, la classe che la implementa avrà totale libertà su come descrivere i metodi `update()`, che dovranno aggiornare lo stato della partita che risiede sul client e la sua View.
Abbiamo scelto di implementare le funzionalità avanzate "Resilienza alle disconnessioni", "Partite multiple" e "Chat".

2 Socket

Tutti i messaggi sono trasmessi come oggetti grazie all'implementazione dell'interfaccia `Serializable`.
Abbiamo scelto di non adottare la trasmissione tramite formato json perché nonostante sia più leggibile, la serializzazione/deserializzazione è più semplice da implementare in quanto supportata dalle librerie di base di Java.

2.1 Comunicazione da Server a Client

Il `ServerMain` resta in ascolto della connessione di nuovi client, effettuando il binding e poi istanziando `ClientRequestsHandler` che si occuperà di tutte le successive comunicazioni in ingresso da tale client.
I cambiamenti del model corrispondo ad una o più notify() del `ListenerHandler`, che inoltra i cambiamenti in broadcast a tutti i `GameListener` in ascolto tramite delle `update()`. A questo punto sarà la classe `GameListenerServerSocket` che si occuperà della serializzazione del messaggio e della trasmissione dello stesso sul socket.
Fa eccezione solamente `getRooms()`, dato che tale richiesta è la prima ad essere chiamata, quando ancora non è stato scelto il nickname da parte del giocatore e non è ancora stato creato il `GameListenerServerSocket` ad esso associato. In questo caso la risposta viene quindi inviata da `ClientRequestsHandler` al client che ha mandato la richiesta.

2.2 Comunicazione da Client a Server

Lato client, invece, ogni azione principale dell'utente che deve essere inviata al server (pescare o piazzare una carta, mandare un messaggio in chat, ping etc...) viene "tradotta" dalla View in una chiamata ad un metodo corrispondente di `NetworkHandlerSocket` che si occupa di serializzare ed inviare il messaggio al server.
Di ricevere e deserializzare i messaggi che arrivano dal server se ne occupa `ServerResponseHandler` che in base al tipo di messaggio ricevuto chiama i metodi i corrispondenti metodi di `GameStateHandler` (per aggiornare il GameState locale del client) e/o di `View` (per mostrare i cambiamenti all'utente).

2.3 Messaggi

I pacchetti scambiati tra server e client si differenziano in messaggi **command** e messaggi **response**.

- I messaggi **command** sono inviati dal client al server e corrispondono ad un comando che deve essere eseguito sul server. Ogni messaggio command ha associato il nome del giocatore. Un messaggio command provoca l'invio di un messaggio response da parte del server.
- I messaggi **response** sono inviati dal server a tutti i client partecipanti di una specifica partita e generalmente corrispondono alla risposta di un messaggio command inviato da un client specifico.

Pertanto, se un client invia un messaggio command, la risposta response non verrà inviata solamente al chiamante, ma anche a tutti gli altri giocatori della Room. L'unica eccezione sono i messaggi relativi alla chat, che vengono inviati solamente al mittente e ai destinatari. Se l'operazione richiesta da un client mediante messaggio command non è valida, viene inviato un messaggio response contenente un errore al solo client richiedente.

Riportiamo di seguito la lista dei tipi di messaggi **command** che possono essere inviati. Per ciascun messaggio sono riportati i relativi attributi, che contengono le informazioni strettamente necessarie per la richiesta.

Messaggi command di gestione della Room di gioco:

- `getRooms`
- `createRoom`
  - `chosenNickname: String`
  - `token: Token`
  - `players: int`
- `joinRoom`
  - `chosenNickname: String`
  - `token: Token`
  - `gameId: int`
- `leaveRoom`

Messaggi command di gestione del flusso di gioco:

- playStarter
  - side: Side
- choosePersonalObjective
  - card: CardObjectiveIF
- playCard
  - card: CardPlayableIF
  - coords: Coordinates
  - side: Side
- pickCard
  - card: CardPlayableIF

Messaggi command di gestione della connessione di rete:

- ping
- reconnectGame
  - nickname: String
  - token: Token

Messaggio relativo all'invio di un messaggio in chat:

- chat
  - text: String
  - receivers: List<PlayerLobby>

Riportiamo di seguito la lista dei tipi di messaggi response che possono essere inviati. Per ciascun messaggio sono riportati i relativi attributi, che contengono le informazioni di risposta relative al messaggio command associato. L'attributo type viene utilizzato per associare la risposta alla richiesta.

Messaggi response relativi alla gestione della Room di gioco:

- resJoinRoom
  - player: PlayerLobby
- resLeaveRoom
  - player: PlayerLobby
- resGetRooms
  - rooms: List<RoomIF>

Risposte relative alla gestione del flusso di gioco:

- resPlayStarter
  - player: PlayerLobby

    - starter: CardStarterIF
    - availableCoords: List<Coordinates>
- resChoosePersonalObjective
    - player: PlayerLobby
    - chosenObjective: CardObjectiveIF
- resPlayCard
    - player: PlayerLobby
    - cardPlayer: CardPlayableIF
    - coordinates: Coordinates
    - points: Integer
    - availableCoordinates: List<Coordinates>
- resPickCard
    - player: PlayerLobby
    - updatedVisibleCards: List<? extends CardPlayableIF>
    - pickedCard: CardPlayableIF
- resPoints
    - pointsMap: Map<PlayerLobby, Integer>
- resWinner
    - players: List<PlayerLobby>

Risposte relative alla gestione della connessione di rete:

- resPing
- resReconnectGame
    - player: PlayerLobby

Messaggio relativo alla gestione di un messaggio in chat:

- resChat
    - text: String
    - sender: PlayerLobby
    - receivers: List<PlayerLobby>

2.4 Sequenza di comunicazione

Nei seguenti sequence diagram abbiamo rappresentato le varie componenti con le loro classi, eventualmente omettendo il suffisso Socket dato che si riferiscono tutte a Socket. La cardinalità dei messaggi è quello che abbiamo scritto nell'introduzione della sezione Socket.

2.4.1 Sequence diagram per la popolazione della Room di gioco

In questo sequence diagram mostriamo la prima parte, in cui un giocatore crea la stanza e poi altri giocatori si uniscono a tale stanza, fino a quando è stato raggiunto il numero di giocatori prefissato dal giocatore che ha creato la stanza.
Questo è un caso di esempio in cui nessun giocatore si sconnette in questa prima fase. Abbiamo tuttavia previsto la possibilità che il giocatore si sconnetta tramite leaveRoom. Se dovesse poi riconnettersi verrebbe trattato come un qualsiasi altro giocatore.

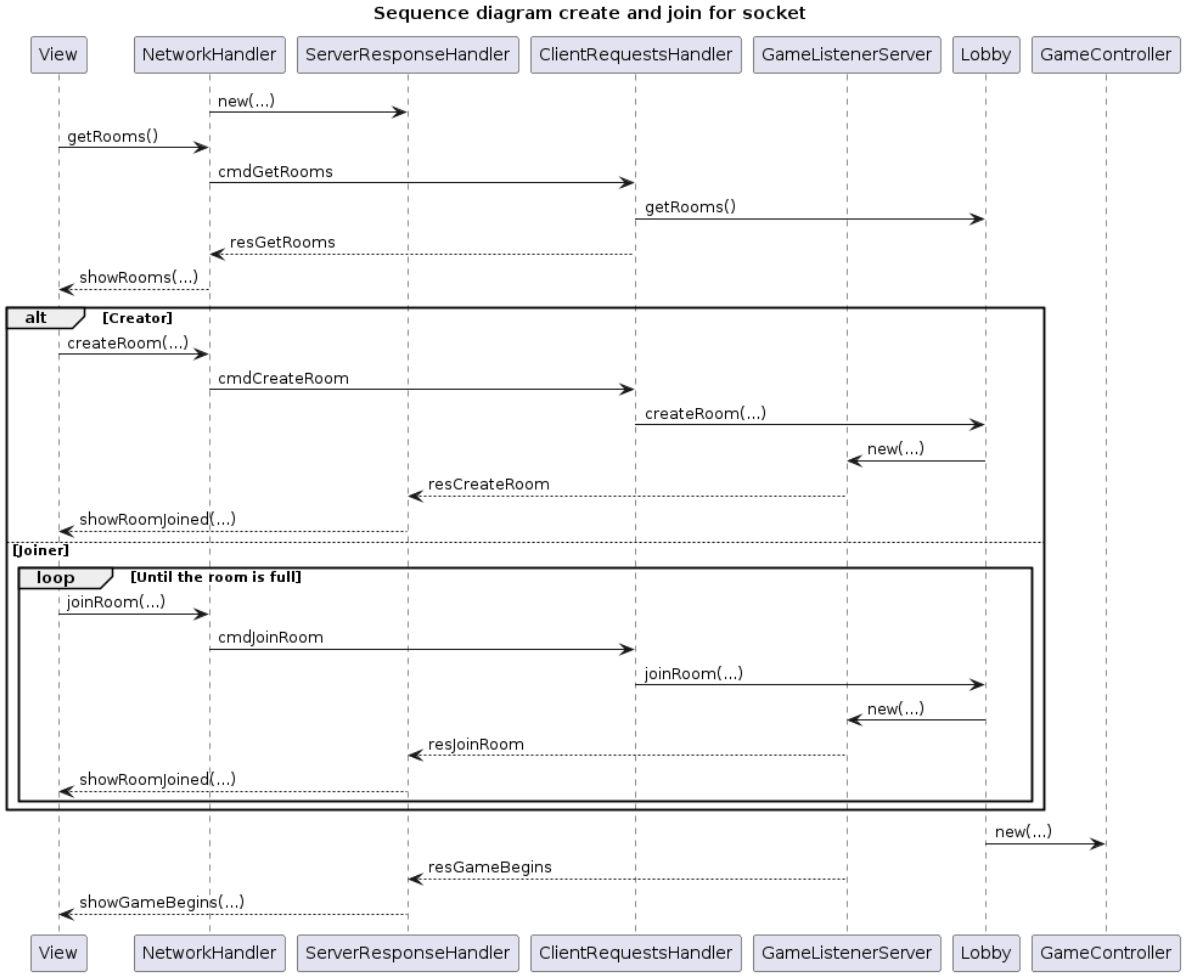

Figura 1: Sequence diagram per la popolazione della Room di gioco.

2.4.2 Sequence diagram per la fase di inizializzazione del gioco

Questa è la fase iniziale del gioco, in cui ogni giocatore deve fare le 2 scelte iniziali su come giocare la carta iniziale e quale carta obiettivo scegliere tra le 2 pescate.

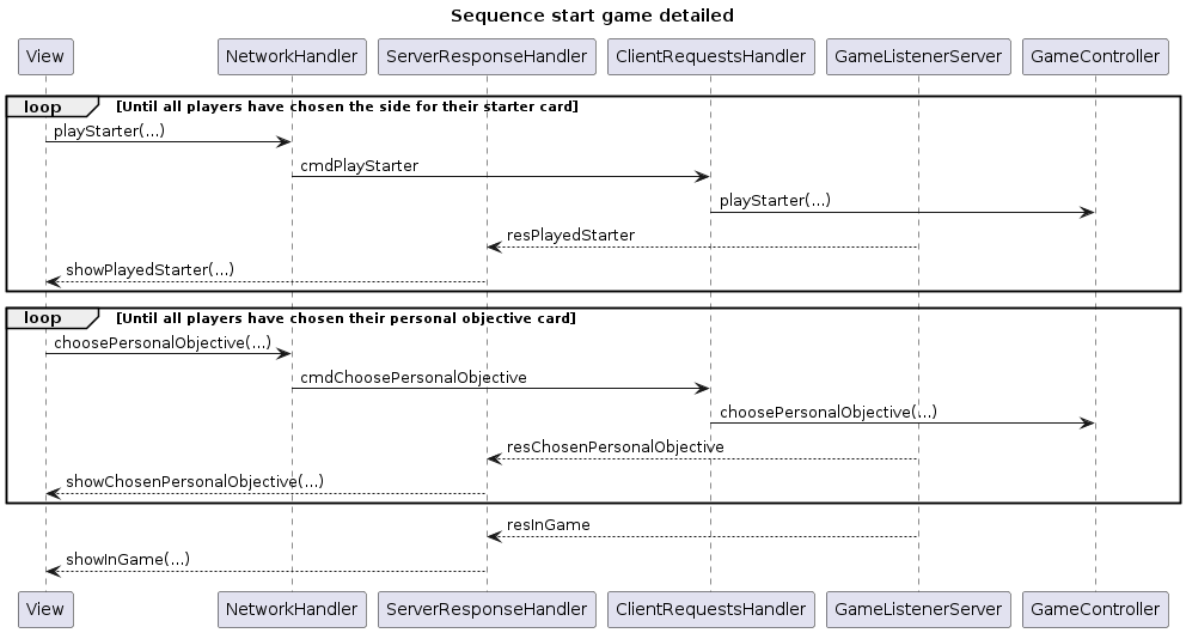

Figura 2: Sequence diagram dell'inizializzazione della partita.

2.4.3 Sequence diagram per la fase a turni del gioco, fino alla fine

Questa è la fase a turni del gioco, nella quale i giocatori pescano e giocano le carte secondo l'ordine stabilito.
Si distingue una prima sotto-fase che termina quando un giocatore ha raggiunto 20 punti oppure sono stati finiti entrambi i deck. In seguito vi è la fase finale in cui si completa il turno in corso e ciascun giocatore gioca un turno addizionale.
Infine, vengono notificati a tutti i giocatori i loro punteggi aggioranti considerando le carte obiettivo, e il vincitore della partita.

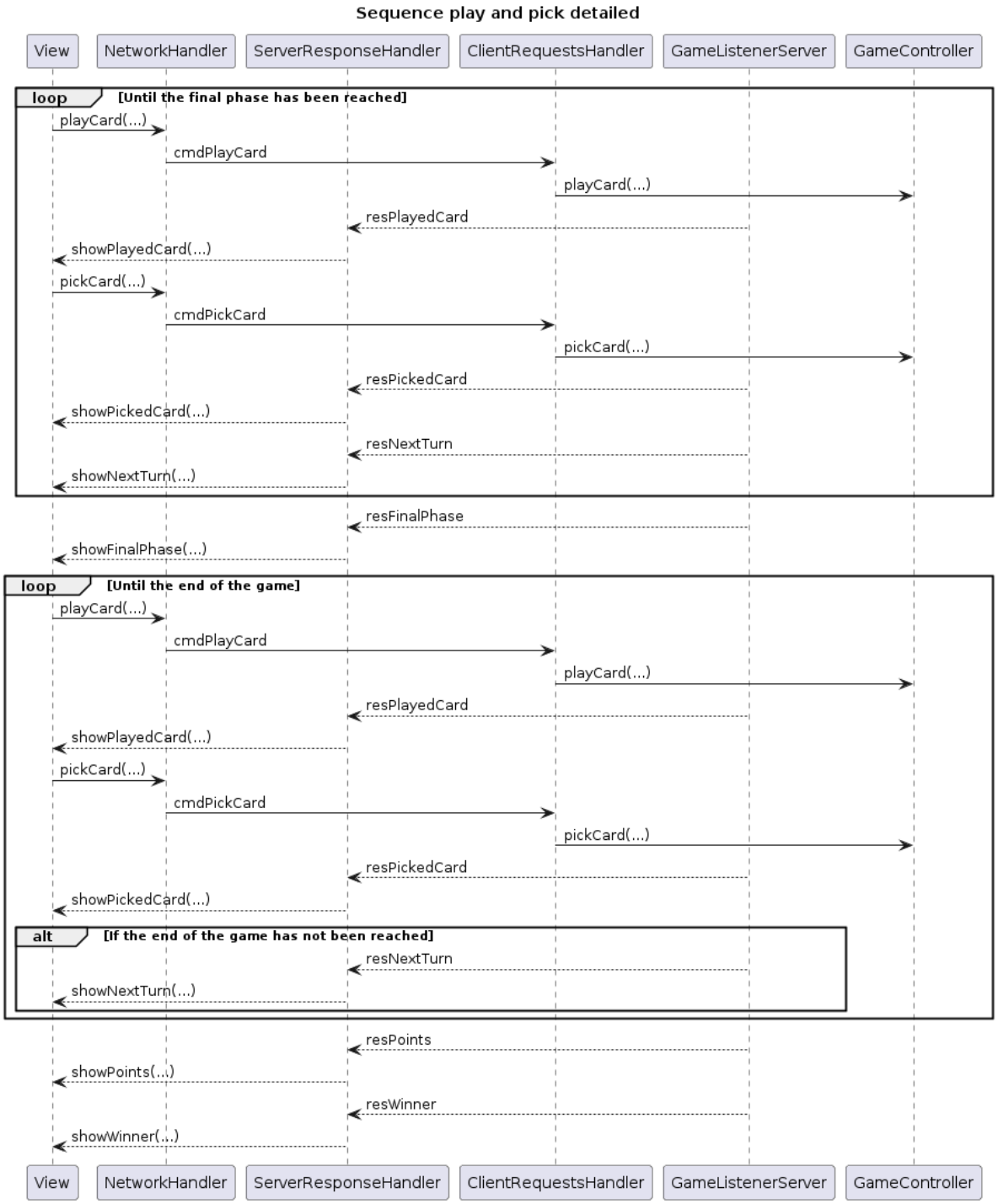

Figura 3: Sequence diagram della gestione del flusso di gioco.

2.4.4 Sequence diagram per la sconnessione e riconnessione di un giocatore

In queto sequence diagram mostriamo un esempio di sconnessione e riconnesione di un giocatore durante la fase a turni del gioco (per la relativa funzionalità avanzata).
Anche se in questo esempio non è mostrato, abbiamo gestito la possibilità che resti solo un giocatore connesso. In tal caso si interrompe la partita e viene fatto partire un timer, allo scadere del quale si verifica il numero di giocatori connessi: se ce ne sono più di 1, la partita riprende; se ce n'è uno,

viene decretato vincitore e viene terminata la partita; se non ce n'è nessuno, viene terminata la partita.

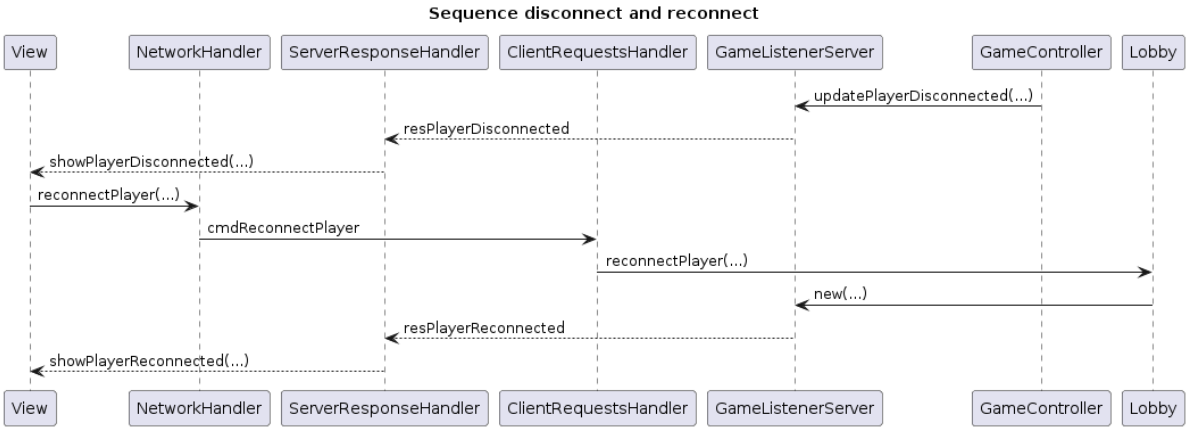

Figura 4: Sequence diagram della per la sconnessione e riconnessione di un giocatore.

3 RMI

Abbiamo implementato anche la comunicazione tramite RMI, che nasconde la presenza della rete tra client e server. La struttura di base dei sequence diagram rimane inalterata, ma le classi intermedie che codificano e decodificano esplicitamente i pacchetti da mandare in rete sono sostituite da altre implementazioni di interfacce comuni che eseguono direttamente chiamate a metodi remoti. Si rimanda all'UML per tali implementazioni.

Così con RMI un generico evento client diventa una chiamata effettuata da `NetworkHandlerRMI` alle classi `LobbyRMI` o `GameControllerRMI` (gli oggetti remoti esposti dal server). Questo comporta una modifica dello stato del gioco, che verrà notificata tramite `ListenerHandler` alle corrispondenti classi `GameListenerServerRMI` (classi presenti nel server, uno per ogni client in gioco). Queste classi effettuano chiamate remote su `GameListenerClientRMI` (classi presenti nel client ed esposte come oggetti remoti in rete, corrispettive di quelle del server), che cambieranno lo stato del gioco per il client e quindi modificheranno la view.

L'unica eccezione a questo è il metodo `getRooms()` di `GameControllerRMI`, che ritorna direttamente la lista delle stanze

4 Interfacce comuni

Quindi entrambe le modalità di comunicazione funzionano con delle loro classi interne, ma il flusso dei messaggi e delle azioni conseguenti si ricongiungono in modo che le classi che vengono dopo, sia nel client sia nel server, possano lavorare senza dipendere dal particolare protocollo scelto. In particolare possiamo distinguere:

- Flusso client - server: dal client le comunicazione partono dalla classe `NetworkHandler`, che si preoccupa di inviare il comando col giusto protocollo di comunicazione al server. Poi nel server i flussi si ricongiungono in `Lobby` o `GameController`, con i giusti metodi chiamati o da `LobbyRMI` e `GameControllerRMI` (per RMI) o da `ClientRequestsHandler` (per Socket).

- Flusso server - client: dal server le risposte al client partono come update tramite l'interfaccia `GameListener`, che si preoccupa di inviare il messaggio col giusto protocollo di comunicazione al client. Poi nel client i flussi si ricongiungono tramite `GameState`: infatti entrambe le classi `GameListenerClientRMI` (per RMI) e `ServerResponseHandler` (per Socket) modificano lo stato del gioco tramite questa classe