Peer-Review 2: Protocollo di comunicazione

Riccardo Piantoni, Matteo Rossi, Federico Pinto, Jacopo Sacramone
Gruppo GC12

8 maggio 2024

Valutazione del protocollo di rete del gruppo GC02

1 Lati positivi

Nel diagramma UML è evidente la presenza di diverse classi controller, quali PlayerController, ChatController, MatchController e DeckController. L'utilizzo di una struttura che sfrutta dei controller multipli è una scelta implementativa efficace, poiché consente una chiara suddivisione delle responsabilità all'interno dell'architettura del software.

La rete è progettata in modo tale che ogni comando sia sempre gestito dal controller dedicato. Ad esempio, il DeckController riceve tutti i messaggi strettamente legati alla gestione del mazzo di carte. Tale approccio assicura una comunicazione più chiara e ordinata e garantisce nuovamente la suddivisione delle responsabilità.

2 Lati negativi

Nel diagramma proposto non è presente un'interfaccia comune tra Socket e RMI. Questo crea una sovrapposizione funzionale tra alcuni attributi quali "clientTypes" nel MatchController e "isRMI" nel PlayerController. Entrambi sono utilizzati per gestire le differenze nelle tecnologie di comunicazione implementate, ma tale scelta introduce una ridondanza non necessaria e aumenta la complessità del codice, non rispettando il principio DRY ("Don't Repeat Yourself"). Si potrebbe invece inserire un layer di astrazione che ha il compito di orchestrare la rete: questo livello offrirebbe quindi un'interfaccia comune dedicata alle operazioni della rete, rendendo trasparente ai livelli sottostanti il tipo di tecnologia di comunicazione scelto dai client.

Il formato previsto per la creazione di messaggi è mutevole in relazione al tipo di informazioni che è necessario scambiare. La scelta di costruire stringhe che concatenano più argomenti nel caso di metodi basilari, distribuire all'interno delle celle di un array nel caso sia necessario scambiare molti argomenti o serializzare oggetti in formato JSON nel caso di informazioni più strutturate non è omogenea e richiede una logica di parsing dei messaggi scambiati declinata per ogni struttura del messaggio esistente. Sarebbe più opportuno uniformare tutte le azioni eseguibili in una struttura generica, per esempio incapsularle all'interno di una classe Message, la quale viene poi serializzata e deserializzata attraverso le strategie note, come il formato JSON, per essere scambiata sulla rete (si noti come questo non è altro che un

modo di implementare il pattern Command). Inoltre, non è chiaro perché le risposte debbano essere caratterizzate dal suffisso *Server*, dato che l'unico agente che comunica con il client è il server stesso.

3 Confronto tra le architetture

Non riscontriamo molte similitudini tra le due architetture, che sono diametralmente opposte in quanto a gestione di rete e logica di gioco, ma ugualmente valide per raggiungere i requisiti previsti.

Come prima osservazione, si riscontra una differenza sostanziale nell'organizzazione dei Controller tra le architetture: a differenza del gruppo revisionato, noi abbiamo optato per l'utilizzo del pattern State al fine di regolare i permessi di esecuzione delle determinate azioni di gioco ai vari client. Nel gruppo revisionato, invece, tale controllo di legalità dell'azione richiesta viene (presumibilmente) eseguito da un unico MatchController.

Ad alto livello, il protocollo di comunicazione risulta si rivela essere simile per entrambi i gruppi, in quanto consiste nell'invio di un messaggio da parte del client e nella conseguente risposta in modalità asincrona da parte del server.

Relativamente alla comunicazione client/server, l'identificazione del client mittente di una richiesta avviene in modi diversi nelle rispettive architetture: il gruppo revisionato lo identifica attraverso il nickname del giocatore, mentre la nostra autenticazione si basa sulla singola istanza di VirtualClient a lui associata, in modo tale che un client malevolo non sia comunque in grado di compiere azioni per conto di un altro giocatore. La nostra scelta deriva dal fatto che idealmente il server dovrebbe accertarsi della provenienza del messaggio deserializzato, e non affidarsi alla trasmissione del nickname come suo parametro, ma questo esula dagli scopi del progetto.

L'invio dei comandi in rete da noi progettato adotta il meccanismo di serializzazione offerto da Java, mentre il gruppo revisionato sfrutta un ibrido di stringhe e oggetti serializzati in JSON. Il formato JSON è invece da noi utilizzato per caricare le risorse testuali delle carte sul client e istanziare le classi delle carte un'unica volta in fase di avvio del server.

Nel progetto da noi sviluppato, la gestione della comunicazione e della rete è progettata e intesa come un layer di astrazione superiore alle operazioni eseguite dal controller. Tale scelta permette di uniformare l'esecuzione dei comandi, la quale diventa di competenza di un'unica classe, il ServerController, e di renderla trasparente rispetto alla tecnologia di comunicazione scelta da parte del client, disaccoppiandola. L'introduzione di nuove funzionalità all'interno dell'applicazione è quindi possibile integrando i metodi del Controller in un'unico punto del codice e non richiede quindi di occuparsi singolarmente dei dettagli implementativi del metodo di comunicazione.

Data la scelta di implementare la funzionalità di Partite Multiple, la nostra architettura è nettamente influenzata dal multithreading, in particolare nelle operazioni esterne alla singola partita: tale influenza risalta anche all'interno del protocollo di comunicazione, vista la scelta di utilizzare una thread pool per l'esecuzione dei comandi ricevuti. Al contrario, il gruppo revisionato non ne ha probabilmente bisogno, in quanto non sembra voler implementare tale funzionalità al momento.