Peer-Review 2: Architettura di Rete

Alberto Schiaffino, Federico Spoletini, Alessandro Zenati

Gruppo 26

Valutazione dell'architettura di rete del gruppo 36.

Lati Positivi

Il protocollo è chiaro e ad alto livello non presenta criticità che impedirebbero uno scambio corretto di messaggi fra Client e Server, riportiamo i principali aspetti positivi:

- **Divisione del gioco in fasi**: Le fasi sono necessarie per verificare che il flusso del gioco sia sempre corretto. Lato client servono per stampare nella giusta sequenza le richieste inoltrate al giocatore, lato server per sollevare e gestire eventuali eccezioni.

- **Divisione della fase di action**: Le singole mosse che può fare un giocatore nella fase di action devono essere separate e inviate singolarmente al server, quindi è corretto l'utilizzo di tre messaggi diversi (useAssistantCardMessage, moveStudentMessage, moveGoddessMessage). La divisione è necessaria per gestire le varie eccezioni o l'utilizzo, in un qualsiasi momento del turno, di una carta esperto.

- **Avviso a tutti i client:** ogni volta che il model viene modificato ogni giocatore deve avere la versione aggiornata del gioco nella sua cli/gui per poter seguire l'evoluzione del gioco dopo le mosse degli altri giocatori.

- **Sincronizzazione nel setup:** è corretto che il server aspetti passivamente una richiesta del client, per poi fornirgli lo stato della partita e la possibilità di settare numero di giocatori e modalità se è il primo giocatore a collegarsi alla lobby.

- **Utilizzo dei messaggi e clientHandler:** il clientHandler gestisce e instrada tutti i messaggi serializzati al client. Ottima soluzione soprattutto per la scalabilità dato che message è un'interfaccia.

Lati Negativi

- **La sincronizzazione dei passi planning e action:** non è interamente passiva. Infatti, il primo messaggio inviato ad ogni passo planning o action è sempre una richiesta del server (askPlayerAction()). Invece il server dovrebbe passivamente aspettare la mossa di un client, e una volta ricevuta e valutata la sua legittimità attraverso dei controlli, prosegue modificando il model e inviando la setupInfoMessage(). Nel caso della CLI, la richiesta all'utente di eseguire una mossa dovrebbe essere gestita interamente lato client e non attraverso un'esplicita richiesta del server.

- **SetUpInfoMessage()**: dall'UML non è chiaro cosa venga inviato a tutti i client dopo l'aggiornamento del model. Una soluzione è serializzare le parti fondamentali del model per poter aggiornare lato client la cli/gui e mostrare a tutti i client dopo ogni mossa l'intero stato della partita (comprese le board degli altri giocatori).

- **Messaggi lanciati dal server:** oltre al setupInfoMessage, il server dovrebbe avvisare i client quando avvengono determinati eventi, ad esempio la vittoria e il termine della partita. In generale, non è chiaro dall'UML come funzioni la ricezione dei messaggi dal server al client.

Confronto tra architetture di rete

Il nostro protocollo di comunicazione è simile in molti aspetti, ma l'architettura di rete è sostanzialmente diversa. Abbiamo un gameHandler sia lato client che lato server per gestire eventi in entrambe le direzioni, in cui attraverso la "reflection" vengono chiamati i vari metodi "update" con overload; infatti, il gameHandler contiene una lista di EventListener, che nel nostro caso sono il Controller e la View. A livello di rete, server e client hanno entrambi due classi: un socketReader per gestire tutti gli eventi in input con una BlockingQueue e un socketWriter con un'altra coda per gli eventi in output. Per gestire l'aggiornamento della View, dopo ogni mossa lanciamo dal server un evento con all'interno una classe serializzata che contiene tutti gli attributi necessari per poter ricreare lo stato del gioco lato client.