Peer-Review 2: Communication Protocol

Andrea Sanguineti, Emanuele Santoro, Andrea Piras
Gruppo GC-45

9 maggio 2022

Valutazione del protocollo di comunicazione del gruppo GC-55.

1 Lati positivi

Complessivamente il protocollo di rete è ben fatto perchè risulta chiaro quando il client deve inviare un messaggio o quando invece ne riceve uno tramite ack o messaggi di errori.
Il processo per eseguire il login e quello che poi porta a creare il Game funzionano bene anche se potrebbero essere integrati in un diagramma unico. La divisione che è stata messa nel server e nel client per gestire il socket, con *ClientConnectionInHandler* e *ClientSocketInServer*, è buona perchè nasconde alle altre classi gli stream e la serializzazione del model.
Infatti la gestione dei turni è chiara e coerente con il model descritto, suddivisa nelle varie fasi di pianificazione ed azione. E' anche facilmente scalabile nel caso il regolamento venisse modificato.
Buona la serializzazione delle classi *MatchInfo* e *PlayerInfo* del model, infatti è comoda la funzione *getMatchStatus*, che permette di selezionare tramite boolean gli attributi da aggiornare in base alle esigenze.

2 Lati negativi

L'uml della parte di rete a primo impatto non risulta essere molto leggibile soprattutto per il fatto che non è ben chiaro cosa sia l'*ActionChoice* e quali messaggi poi vengono scambiati per eseguirla e in che modo le varie interfacce *Listener* e *Observable* sono collegate tra loro, come ad esempio l'interfaccia *Listener* nel package *Server* implementata solo da una classe esterna al package.

Sempre relativo alle interfacce la *ClientConnection* è implementata da tre classi le quali implementano in maniera diversa listener e observable. Sarebbe meglio infatti dividere quella parte di connessione gestita dal socket di *ClientSocketInServer* con *ClientHandlerSocket* dal pattern con gli observable e listener per aggiornare le view e il controller. In questo modo infatti classi quali *View* e *ClientConnectionHandler* implementando entrambi quando viene chiamata la *notify* dell'observable chiamano anche su loro stessi la *onAction*. Abbiamo inoltre notato alcuni problemi:

- L'attributo *currentMatchIsCLI* non è riferito all'intero game, infatti ogni giocatore può scegliere indipendentemente se giocare in modalita CLI o GUI, come specificato nei requirements. Di conseguenza tutte le funzioni che utilizzano questo attributo sono da revisionare.
- Nel diagramma del login per chiedere al model se esiste un match gli passa l'username ma si potrebbe evitare di passare perchè la creazione del match non implementando le aprtite multiple è indipendente dal player.
- Le classi che implementano *Message* potrebbero essere unificate alle *ServerResponse* perchè sono entrambi scambiate dal server al client
- Il vostro client utilizza un unico thread per inviare e ricevere i messaggi dal server, quando si potrebbe fare un thread a parte nel client dedito alla ricezione e alla *notify* delle classi che devono mostrare il messaggio ricevuto. In questo modo si potrebbe anche permettere di vedere le azioni degli altri giocatori quando non è il proprio turno.

3 Confronto tra le architetture

Il vostro controller prevede una classe di controllo per validare un'azione del client, a differenza del nostro che svolge il controllo all'interno delle varie funzioni del controller. Effettivamente la vostra implementazione è più generica e facilmente modificabile.
La vostra implementazione di *ServerResponse* fa uso di un enumerator per distinguere il tipo di risposta del server, mentre noi la decodifichiamo attraverso il tipo dinamico del messaggio serializzato dal server. Tutti i messaggi implementano la stessa interfaccia *Messagge*, e quindi ognuno possiede la stessa funzione *execute*, implementata in modo diverso.