Peer-Review 2: UML e Networking

Gruppo 01

Zanca Federico, Zhuge ZhengHao Michele, Wu JiaHui, Zheng Fabio

Valutazione del gruppo 41.

Lati positivi

La suddivisione dei ruoli dediti alla parte di networking, come ad esempio l'estensione dell'interfaccia *Connection* in *RMIConnection* e *SocketConnection* è implementata in modo funzionale e così come anche l'organizzazione dei vari Handler.

Inoltre è stata molto apprezzata la presenza di diagrammi atti a rappresentare il flusso di messaggi durante diversi scenari di attività, tra i vari componenti necessari a implementare le funzionalità di networking, come Client Interface, Client Handler, Client Socket e SocketConnection.

Lati negativi

Dai diagrammi UML pare che non venga usato uno strategy pattern per i messaggi di rete, quindi probabilmente viene usato uno switch che controlli l'Event Type e faccia qualcosa di conseguenza. Una soluzione più elegante sarebbe rendere MessageFromServer e MessageFromClient due interfacce con un metodo *execute(ServerClass o ClientClass).* In questo modo chi riceve il messaggio chiama *message.execute(this)* e sarà poi il messaggio stesso a chiamare il metodo relativo ala gestione di quell'evento sull'istanza del ricevente passata al messaggio come parametro del metodo *execute()*.

Per quanto riguarda le *CommonGoalsCard*, potrebbe essere una buona idea raggrupparli, tramite l'impiego di file json, ad esempio, invece di continuare ad usare 12 classi differenti.

Confronto fra le architetture

Dall'analisi e dal confronto dell'UML e delle soluzioni implementative adottate da noi e dal gruppo 41, noi potremmo essere nella posizione di consigliare alcune idee per implementare le funzionalità avanzate di "Partite multiple" e "Resilienza alle disconnessioni" che abbiamo già adoperato.

Tramite funzionalità di Heartbeat (simil keepAlive), più precisamente tramite la classe *HeartBeatMessage*, è facilmente ottenibile un sistema di controllo per la raggiungibilità attraverso la rete dei client. Più precisamente noi assegniamo un timer ad ogni client che si connette al server; quando avviene la connessione il server avvia un thread che fa partire il timer del client connesso e aspetta che scada. Dall'altra parte il client appena si connette al server avvia un thread che a intervalli regolari manda un HeartBeatMessage al server. Quando il server riceve un HeartBeatMessage resetta il timer del client che ha mandato l'HeartBeatMessage. Se il timer fa in tempo a scadere vuol dire che è passato troppo tempo dall'ultimo HeartBeatMessage ricevuto dal client, quindi il server può considerare il client come disconnesso e procede a svolgere le operazioni del caso (notificare gli altri giocatori, salvarsi che deve saltare il turno di quel giocatore nella partita,...).

Per permettere invece di giocare “Partite multiple” abbiamo fatto uso di un sistema di Lobby (la lobby ha il proprio model e controller che vengono istanziati alla creazione) con classi di messaggi dedicati per ogni fase del gioco e controlli sui flussi di messaggi in entrata ed uscita. Sarà quindi compito della classe Lobby (a cui è associata una partita) aggiungere e rimuovere gli observer legati al model quando un client si connette o si disconnette.