Peer-Review 2: Controller & Network

Massimo Albino Sandretti, Cristiano Sartori, Mario Valente, Alberto Francesco Visconti

Gruppo GC52

Valutazione della documentazione della parte di Controller e Network del gruppo GC42.

Lati positivi

Apprezziamo il fatto che i consigli della scorsa Peer Review abbiano generato un riscontro e che alcuni punti sono stati adottati o presi in considerazione.

La qualità della documentazione ci ha permesso di comprendere senza problemi il lavoro svolto (oltre a inevitabili problematiche relative all'estraneità al progetto) e di trarre le nostre conclusioni.

Riteniamo notevole lo stato descritto per quanto riguarda GUI e TUI. Anche l'introduzione di un'interfaccia grafica per il server è un'idea interessante, da cui prenderemo sicuramente spunto.

La scarsa differenza che c'è tra le architetture dei nostri Controller lascia poco spazio di discussione. Questo per noi è inevitabilmente un lato positivo dal momento che riteniamo la nostra implementazione (e di conseguenza anche la votra) essere corretta. Infatti, come sarà evidente proseguendo la lettura, la quasi totalità delle nostre discussioni vertono sulla parte di rete, sulla quale abbiamo seguito strade diametralmente opposte.

Lati negativi

- la scelta di effettuare il binding sul registry RMI dell'oggetto remoto di ogni player. Questa soluzione ci sembra non ottimale (suggeriamo anche di testarne il funzionamento su macchine diverse, se non è stato ancora fatto; questo perché, secondo una rapida ricerca su internet, risulta che un processo non può effettuare il binding in un registry su una macchina che non sia la stessa su cui gira il processo stesso). Per chiamare i metodi di un oggetto remoto non è necessario che questo sia bindato ad un registry, ma è sufficiente avere un riferimento all'oggetto stesso, che il client potrebbe passare in fase di connessione. Questo evita il problema di dover garantire nomi univoci per questi oggetti. Risulta inoltre più difficile che si generino errori di comunicazione ed è più facile gestire eventi come la disconnessione.

- non è molto chiaro il problema descritto che riguarda i listener e la necessità di serializzarli. Inoltre, non ci convince a pieno il fatto che sia GameController a fare da intermediario tra gli eventi nel Model e gli update della View. Questo, secondo noi, va contro i principi del pattern MVC stesso. Se una volta generato un evento (da un observable), questo potesse essere inviato direttamente (con eventuale traduzione in messaggio/chiamata remota) al client, questo sarebbe più in linea con le indicazioni fornite. Nella nostra implementazione, a svolgere le veci di Listener sono degli oggetti intermedi che gestiscono la comunicazione con

i client (solo in uscita nel caso di RMI e a doppia via nel caso dei socket), che noi chiamiamo ClientHandler.

- Durante un precedente laboratorio era stato sconsigliato di utilizzare un singolo oggetto remoto che riceva messaggi da tutti i client (per poi effettuare il dispatch in base al game e/o al player). Nonostante anche la nostra architettura iniziale prevedesse questo tipo di struttura, ci è sembrato opportuno mutarla nella forma richiesta. La vostra struttura ci sembra sufficientemente solida per permettere questo aggiornamento senza problemi.

- Sulla stessa linea del punto precedente, notiamo che tutti i messaggi provenienti da client vengono ricevuti da un unico oggetto (il singolo SocketControllerServer. Nonostante sia vero che questo rende simmetrico il comportamento del SocketControllerServer a quello dell'RMIControllerServer, ci sembra sconveniente per lo stesso motivo del punto precedente. Chiaramente se i Client RMI comunicano direttamente con il GameController, ci si aspetterebbe che, in un modo o nell'altro, anche i Client socket lo facciamo. Inoltre, se il client fosse sollevato dall'onere di specificare il gameID per ogni interazione (salvo l'unione ad un gioco), questo binding andrebbe fatto all'interno del SocketControllerServer o tramite la creazione di vari SocketControllerServer. E nessuna delle due soluzioni è più simmetrica con l'equivalente RMI. Studiando il Sequence Diagram ci sembra evidente che SocketControllerServer debba implmenetare RemoteViewController. Per essere inserito all'interno della lista di oggetti di questo tipo nel GameController, in modo che quest'ultimo possa inoltrare aggiornamenti anche ai Client Socket.

Confronto tra le architetture

Notiamo una notevole somiglianza in determinati aspetti dell'architettura. Soprattutto per quanto riguarda i controller. Anche la scelta di utilizzare un identificatore per riconoscere i client a garantire un minimo grado di autenticazione è un punto in comune tra i nostri lavori. Nonostante questo, ci sono molte differenze sostanziali, soprattutto per la gestione della comunicazione di rete. Vogliamo esporre le principali nei punti che seguono.

- Notiamo, dalle interfacce remote, che è il server stesso a porre richieste esplicite al Client. Questa è una soluzione che riteniamo valida. Noi, d'altro canto, abbiamo optato per un Client che gestisca autonomamente la propria logica. Il server si limita a comunicare lo stato del gioco (aggiornamento del turno, aggiornamento della fase del gioco, aggiornamento della fase del turno) ed è il client a sapere quale operazione deve "proporre" alla user interface.

- nella documentazione non viene fatto riferimento alla decisione presa in merito alla sincronizzazione dei socket o alla desincronizzazione di RMI. Dato che tutte le comunicazioni vengono trattate come chiamate (e non come messaggi) e dato che non notiamo code all'interno dei Controller, supponiamo che l'opzione scelta sia la sincronizzazione dei Socket. Come illustrato nel punto successivo, noi abbiamo optato per la desincronizzazione di RMI, che ci sembrava l'approccio più naturale.

- non ci è chiaro al 100% come avvenga la traduzione da messaggio a chiamata a metodo. Dalla documentazione sembrerebbe che ci siano dei metodi adibiti a questo compito all'interno dei SocketControllerServer e SocketClient. Questa soluzione è valida e non la contestiamo. La nostra soluzione prevede un uso estensivo dell'oggetto Messagge (e sottoclassi). Facciamo infatti l'operazione opposta alla vostra. Ogni comunicazione è trattata come Messaggio che, all'evenienza, viene trasformato in una Remote Method Invocation. I messaggi stessi contengono al loro interno un metodo che permette di "eseguire" l'operazione che il messaggio richiede. Questo porta con sé vari vantaggi:
  - non è necessario convertire esplicitamente i messaggi in chiamate a metodo (operazione sicuramente più laboriosa di fare la trasformazione inversa)
  - i messaggi dei client (sia socket che RMI, anche se questi ultimi vengono creati in risposta a chiamate remote), una volta arrivati al GameController (o alla GameCollection) possono essere inseriti direttamente in una BlockingQueue e processati in modo asincrono.
  - Si alleggerisce il lavoro svolto dagli oggetti remoti stessi. Questi infatti (come esposto nel punto precedente) si limitano a costruire messaggi corrispondenti al metodo chiamato (con i dati passati come parametri) e posizionarli in una coda.
  - i messaggi possono essere creati direttamente all'interno del Model e passati (tramite listener/observer) ai gestori della comunicazione con i client e inoltrati direttamente.
  - L'operazione di mandare messaggi tramite RMI è molto semplice. Dato che ogni messaggio può essere "eseguito" passandogli una RemoteView, per eseguire la chiamata remota è sufficiente "eseguire" il messaggio passandogli il riferimento alla RemoteView. Per i client Socket è sufficiente che questi ultimi lo "eseguano" direttamente passando la propria View "concreta". Simile è anche il funzionamento per i messaggi mandati dal client al server.