PEER REVIEW 2 - Gruppo GC17

Puccia Niccolò, Pozzato Luca, Romano Stefano, Saso Edoardo

Descrizione UML Network

La presente peer review si propone di illustrare e descrivere brevemente le scelte architetturali della parte di UML relativa alla rete e ai protocolli di comunicazione client-server.

Communication Setup

In un primo momento il gestore del server inserisce a terminale i comandi per avviare il programma Codex Naturalis, lanciando così ServerMain.
Analogamente, lato client, sarà avviato da terminale il programma e lanciato il ClientMain.

ServerMain

Prima che il clientMain venga lanciato viene eseguito il serverMain che si occupa di creare un RmiServer e un SocketServer e stanziare controller e model. Al server viene passato il controller in modo da poter tramite esso interagire con il model mantenendo comunque una separazione nei compiti.

E' necessario che le istanze di RmiServer e SocketServer siano già presenti al momento dell'avvio di clientMain e che dunque i due main vengano eseguiti sequenzialmente per evitare che, in seguito a ritardi di rete, un istanza di client tenti di mettersi in contatto con un server non ancora creato. Se ciò dovesse verificarsi verrà sollevata un'eccezione che comunica al client l'inesistenza del server a cui vorrebbe collegarsi.

Prima di terminare il serverMain genera una coda di pertinenza del server e un thread per ogni tipologia di server.

La necessità di creare un server specifico per entrambe le connessioni nella fase iniziale del gioco è funzionale al fatto che client diversi possono collegarsi con tecnologie diverse.

ClientMain

Scopo del ClientMain è quello di mettersi in ascolto su stdin dove verrà richiesto all'utente di inserire il nome della tecnologia con la quale desidera instaurare la connessione (RMI o socket) e il tipo di interfaccia scelta (GUI o TUI).

In base al tipo dei parametri inseriti per ognuno dei giocatori, il clientMain andrà a creare un client opportuno. Attualmente, la creazione del client viene gestita creando RmiClient SocketClient a seconda della scelta di connessione, e un thread a loro associato. A queste verrà passata la scelta dell'interfaccia e si occuperanno di creare la Cli o la Gui a seconda del caso. Per rendere più scalabile, estendibile ed efficiente il processo, abbiamo intenzione, in un secondo

momento, di implementare un Abstract Factory Pattern. Tale scelta implementativa comporta la presenza di TuiClient e GuiClient come classi "fabbrica" che espongono pubblicamente i metodi per creare delle istanze di TuiRmiCLient, GuiRmiClient, TuiSocketClient e GuiSocketClient.

Ciascuna di queste classi ereditano dall'interfaccia RMI e Socket il metodo Run() in override che viene chiamato all'interno del clientMain. All'interno del metodo run() vengono svolte le seguenti operazioni, ciascuna secondo la logica specifica della classe che presenta l'override:

1. **Creazione di una delle classi che implementano l'interfaccia view** (TUI o GUI) a seconda della scelta dell'utente passando il client come argomento.
2. **Creazione di una copia semplificata del model** (da ora in avanti denominata minimodel) da associare alla view la cui funzione è quella di conservare per ogni client gli oggetti del gioco che sono visibili dal giocatore. Tale entità verrà modificata dinamicamente nel corso del gioco quando il model, modificato a causa dell'interazione con l'utente, invia in broadcast ai client che lo osservano gli update con una vista modificata dello stato del gioco.
3. **Lancio della view** sopra citata con effetti diversi a seconda che sia un GuiClient o un TuiCLient.
4. **SetUp del client:** creazione di una coda lato client e di un thread ad esso associato (vedi gestione meccanismo command/event).
5. **Binding/ Handshake:** Il client tenta mettersi in contatto con il server. Se è un RmiClient fa la lookup nel registry e avvia la procedura di binding, se è un SocketClient segue il protocollo TCP per l'handshake.

Midgame

Una caratteristica della tecnologia RMI che può rivelarsi controproducente nell'ordinaria gestione delle comunicazioni consiste nel fatto che quando il client chiama un metodo di un oggetto remoto esposto dal server (chiamata che ricordiamo essere sincronizzata) esso attende che tale metodo termini prima di ritornare al chiamante bloccando di conseguenza l'invio di altri comandi da parte del client. Da ciò segue che vengono irrimediabilmente perse le chiamate a metodi dell'oggetto remoto arrivate quando il lock è in possesso di un thread terzo il quale aspetta il termine della sua chiamata che tarda a ritornare.

Per ovviare a tale problema abbiamo deciso di rendere asincrono RMI implementando una doppia coda di comandi e eventi:

Commands

La creazione del comando avviene ad opera della classe View. Compito della view è quello di leggere l'interazione con l'utente tramite le modalità previste dall'interfaccia e costruire un'istanza di comando passandogli come parametro tutti e soli gli oggetti coinvolti nell'interazione che l'utente ha avuto con la vista o comunque quelli che servono al controller per chiamare il metodo che modifica lo stato secondo i desideri dell'utente. Tali oggetti vengono prelevati dal minimodel collegato a ogni view.

Esempio: il giocatore vuole piazzare una carta presente nella sua mano sopra un'altra e utilizza un'interfaccia testuale. La view leggerà l'inserimento a terminale del comando PLACE: <IDcartasopra>, <IDcartasotto>, <AngoloCartasotto>, <FronteCarta>. Andrà nel minimodel a cercare nella mano del giocatore una carta con IDcartasopra e nella struttura a cercare una carta con IDcartasotto. Una volta trovati li preleva e costruisce un nuovo comando passando tali oggetti all'interno del costruttore, insieme.

Da ciò deriva che il costruttore della classe Command presenti diversi override, uno per ogni tipologia di comando previsto. Di seguito elenchiamo quelli individuati:

- CreateGameCommand
- JoinGameCommand
- ChooseCommand
- PlaceCardCommand
- DrawCardCommand

Una volta preparato il comando viene chiamato il metodo SendCommand() del client.

Tale metodo prende l'oggetto comando generato e lo mette nella coda del client senza attendere alcuna risposta e dando pertanto la possibilità a tutti i comandi di essere accodate.

Meccanismo a doppia coda

Per ovviare al problema di connessioni troppo lente tali per cui in uno scenario a singola coda tra client e server il problema tipico dell'RMI potrebbe ripresentarsi (il metodo che mette in coda il comando sul server impiega troppo tempo a ritornare e nel frattempo il client crea nuovi comandi abbiamo previsto un meccanismo a doppia coda: una per accodare i comandi della view al client e una per accodare quelli del client al server.

In questo modo, alla chiamata del metodo sendCommand(), il client mette in coda il comando e ritorna subito dopo senza passare per la rete permettendo a richieste temporalmente molto vicine di essere gestite e progressivamente incanalate prima verso la coda del server tramite il risveglio di un thread apposito e poi passate al controller.

Se il client è un RmiClient il comandi viene prelevato dalla coda del client e passato come parametro al metodo receiveCommand() che l'oggetto remoto server espone. Se invece il client è un SocketClient allora il thread risvegliato dovrà serializzare il comando contente gli oggetti utili al controller prima di scrivere sullo stream. Sarà compito dello Skeleton lato server leggere il comando da stream, deserializzarlo e chiamare il metodo del server receiveCommand().

La chiamata di receiveCommand() sul server ha come effetto l'aggiunta del comando alla coda del server e il risveglio di un thread che andrà ad estrarre progressivamente i comandi dalla coda.

Lo stesso thread si occuperà di processare il comando chiamando direttamente il metodo execute() della classe Command passando il controller come argomento (tale metodo sarà in override nei vari tipi di comando sopra elencati chiamando di conseguenza i vari metodi del controller).

Events

L'effetto della chiamata dei metodi del controller da parte del comando si concretizza nel cambiamento del model. Il model, modificato dall'azione del player, deve notificare alla view il delta dei cambiamenti, in modo che ogni client possa conservare una vista aggiornata nel minimodel e, in funzione di questo, modificare l'interfaccia.

Gli eventi di aggiornamento della view identificati vengono elencati sotto:

- CreateGameEvent
- JoinGameEvent
- ChooseEvent
- StartGameEvent
- PlaceEvent
- DrawEvent
- EndGameEvent
- ErrorEvent
- WaitEvent

L'evento generato viene propagato fino al server che lo inserisce in una coda di uscita. Quindi processa un evento alla volta, chiamando il metodo receiveEvent() esposto dall'oggetto remoto client o dal suo sostituto socket ClientSkeleton.

Analogamente al percorso di andata, nel caso di RMI avremo l'impressione che il metodo del client venga chiamato in maniera diretta; mentre, nel caso di Socket, avremo la serializzazione e la scrittura sullo stream da parte dello Skeleton, la lettura dallo stream e la deserializzazione ad opera del client, e infine la chiamata del metodo receiveEvent() del client.

L'evento verrà inserito in una coda di eventi in entrata. Nuovamente gli eventi saranno processati uno ad uno, stavolta da parte del client che chiama il loro metodo doJob() e gli passa come argomento il suo minimodel.

Il metodo doJob() ha un override per ogni tipo di evento e si occuperà di chiamare tutti i setter del minimodel con le specifiche informazioni aggiornate.

Finalmente il minimodel può chiamare il metodo esposto dalla view viewUpdate() con le nuove informazioni e l'utente potrà vedere a schermo i cambiamenti.