Peer-Review 2: Controller e Network

Aryan Sood, Federico Villa, Matteo Vitali, Marco Zanzottera
Gruppo GC19

10 maggio 2024

1 Lati positivi

La classe `CommonServer` svolge un'ottimo compito nel dividere le funzionalità del server, con la creazione di lobby per separare le azioni relative ad una partita da quelle di connessione al server. Questo permette di poter gestire più facilmente la funzionalità avanzata delle *Partite Multiple*, così come permette all'utente di uscire da una partita senza dover chiudere l'intero programma. Questa separazione consente di ridurre il numero di azioni che un giocatore connesso può fare: un client connesso ad una partita non può creare un nuovo gioco o entrare in una lobby. Per gestire la funzionalità avanzata della *Resilienza alle disconnessioni* l'introduzione di un `ping` è molto utile: permette di far individuare al server quando un client non è più connesso e quando dopo una disconnessione un dato client si riconnette.

2 Lati negativi

Riteniamo che il gioco possa risultare più inefficiente con gli attuali messaggi di richiesta di informazioni: se prima di poter mandare al server la richiesta di effettuare alcune mosse devo richiedere se è il mio turno, rischio di dover aspettare una conferma prima di poter capire quali azioni posso effettivamente fare. Risulterebbe più efficiente lasciare al server il compito di sincronizzare i vari turni, le varie partite e mandare i relativi messaggi al momento opportuno. Ad esempio, il messaggio `ISMYTURN_REQUEST` potrebbe essere gestito in questo modo: il server quando riceve una `DRAW_REQUEST` sa

che il prossimo giocatore che potrà effettuare una mossa sulla propria stazione di gioco sarà un altro, quindi notifica a tutti i giocatori connessi in un certo gioco un messaggio che sancisce che il turno di un giocatore è finito (eventualmente specificando il nickname del giocatore successivo). Applicando quest'approccio, quindi il server manda automaticamente le informazioni ai client e non aspetta che siano loro a richiederle, si potrebbe ottimizzare una partita e rendere anche meno complessa la richiesta di informazioni, che attualmente richiederebbe al client un thread dedicato che manda costantemente richieste per determinare le azioni che un client può svolgere. Avendo implementato il ping si potrebbe automaticamente determinare quando un client non è più connesso e quindi il server potrebbe autonomamente capire quando non deve più mandare i messaggi ad un certo client.

3 Confronto tra le architetture

Pur avendo anche noi utilizzato dei messaggi per determinare lo stato di connessione del client (`ping`), abbiamo adottato un approccio basato su due tipologie di messaggi di ping: il primo mandato dal client verso il server e il secondo spedito dal server verso il client. Così facendo il server riesce a capire efficacemente quando un client non è più connesso e un client può capire quando il server non riesce più a raggiungerlo e quindi quando deve iniziare la procedura di riconnessione al server.

Nel nostro approccio abbiamo separato i messaggi in due diverse classi in base al destinatario: `MessageToServer` e `MessageToClient`. Questo approccio rende più facile la ricezione dei messaggi, per la quale anziché utilizzare una catena di `instanceof` abbiamo utilizzato un pattern `visitor`. La minore complessità deriva dal dover gestire, ad esempio, in fase di ricezione da parte del server solo i messaggi che estendono la `MessageToServer` e non tutti i possibili messaggi creabili. Questo pattern potrebbe, ad esempio, essere applicato per rendere più pulita la divisione delle azioni da svolgere nel metodo `manageInMessage(message)`.