

Bajnarola

Progetto di sistemi distribuiti

Davide Berardi Matteo Martelli Marco Melletti
0000712698 0000702472 0000699715

3 dicembre 2015

Sommario

In questo documento verrà descritto *Bajnarola*, un videogame distribuito ispirato al gioco da tavolo *Carcassonne*. Saranno oggetto di questo documento le fasi di progettazione ed implementazione del sistema, con particolare riguardo per l'architettura di rete: uno degli aspetti principali di un software interattivo multiutente distribuito. Dopo aver introdotto il gioco e descritto gli aspetti progettuali ed implementativi, valuteremo il risultato ottenuto con le dovute considerazioni in termini di correttezza, usabilità ed affidabilità.

1 Introduzione

Sempre più giochi da tavolo ormai vantano una versione digitale, caratterizzata dalla possibilità di permettere agli utenti di giocare in modalità multiplayer. Allo stesso tempo sono ancora rare implementazioni completamente *peer-to-peer* di giochi multiplayer, spesso sono invece basate sull'architettura *client-server*. In questo documento descriveremo il nostro lavoro di progettazione ed implementazione della versione digitale di un gioco da tavolo con particolare interesse riguardo l'architettura di rete distribuita utilizzata nella modalità multiplayer.

*Carcassonne*¹ è un gioco da tavolo dei primi anni 2000, che consiste nel creare un paesaggio medievale posizionando e accostando tra loro vari tipi di tessere, che rappresentano una parte di città, un tratto di strada, un campo coltivato o un monastero². Completando più città, strade o monasteri attraverso tali tessere, i giocatori (da 2 a 5 nella versione base del gioco) accumulano i punti necessari a vincere la partita. Al fine di comprendere al meglio le prossime sezioni di questo documento, verranno illustrate ora le meccaniche del gioco.

1.1 Carcassone

All'inizio della partita, una specifica tessera è posizionata sul tavolo scoperta; le altre tessere vengono mescolate e posizionate coperte in un mazzo. Ciascuna di tali tessere rappresenta un frammento di paesaggio, e può contenere uno o più dei seguenti elementi:

- tratti di strada, inclusi incroci e curve;

¹[https://it.wikipedia.org/wiki/Carcassonne_\(gioco\)](https://it.wikipedia.org/wiki/Carcassonne_(gioco))

²Si fa riferimento alla prima versione del gioco in cui non sono presenti fiumi, locande, ponti o altri elementi di paesaggio introdotti dalle varie espansioni.

- aree cittadine racchiuse da mura;
- campi che circondano le città e accolgono le strade;
- un monastero.

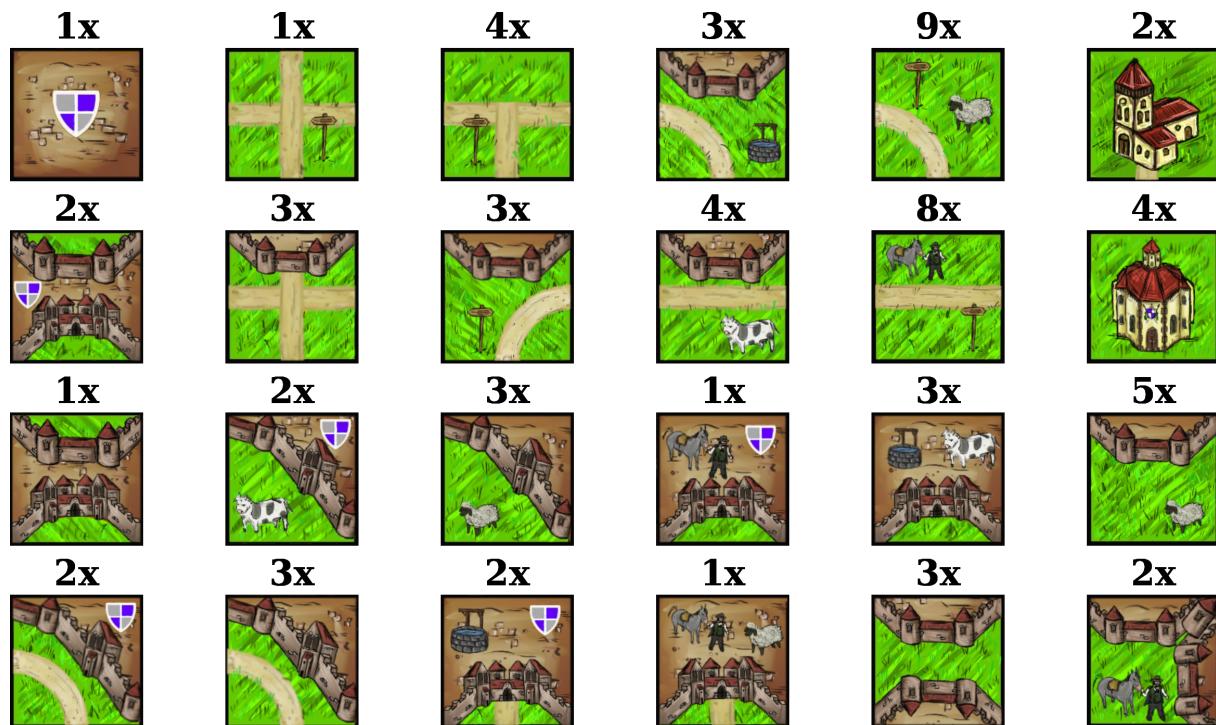


Figura 1: Elenco delle tessere presenti nel gioco

A turno ogni giocatore estrae una tessera dal mazzo e la posiziona scoperta sul tavolo a contatto con almeno una tessera già piazzata, in modo da mantenere la coerenza con le vicine, estendendo eventuali strade, campi, o città già presenti.

Dopo aver posizionato la tessera, il giocatore può decidere di piazzare una pedina, detta *seguace* o *meeple*, su un elemento del paesaggio presente sulla tessera appena posizionata reclamandone la proprietà, a patto che non sia già posseduto da un altro giocatore. Se un elemento diviene una congiunzione di due elementi dello stesso tipo precedentemente non collegati, può ricadere sotto la proprietà di più giocatori; in questo caso, chi ha il numero maggiore di pedine sull'elemento rimane il proprietario, se sussiste un pareggio tutti i giocatori con il numero massimo di seguaci posizionati sull'elemento sono ritenuti proprietari e otterranno il punteggio relativo.

Quando un elemento viene completato, se ad esempio le mura di una città vengono chiuse o se una strada ha due estremità, il proprietario acquisisce i relativi punti e tutte le pedine che erano sull'elemento vengono restituite ai relativi giocatori. Il punteggio di un elemento è dipendente dal numero e dal tipo di tessere che lo compongono.

Unica eccezione viene fatta per i poderi, campi coltivati delimitati dagli altri elementi, che non vengono mai completati e sono soggetti ad un conteggio speciale al termine della partita.

Il gioco termina con il piazzamento dell'ultima tessera, dopo il quale viene effettuato il conteggio finale dei punteggi derivanti dagli elementi non completati posseduti dai giocatori; vince

il giocatore che, dopo il conteggio finale, ha totalizzato più punti³.

La semplice struttura del gioco e la sua organizzazione a turni rende interessante l'approccio distribuito, in quanto si evita di incorrere in problemi di prestazioni tipici dei giochi reattivi. Questi ultimi infatti possono avere requisiti prestazionali molto stringenti. In particolare spesso richiedono che le comunicazioni di rete siano a bassa latenza al fine di evitare artefatti grafici fastidiosi dovuti ai ritardi di rete o all'overhead di comunicazione.

Come già accennato, il gioco da noi considerato necessita invece di requisiti prestazionali più laschi, soprattutto per i lunghi tempi decisionali degli utenti (comparati ai tempi di rete e computazionali).

2 Aspetti progettuali

Il gioco in questione è stato progettato nell'ottica di un sistema resistente ai guasti.

L'aspetto critico studiato maggiormente per raggiungere tale obiettivo è stato senz'altro la comunicazione, pensata per risultare robusta e allo stesso tempo in grado di fornire reattività ai client di gioco, nonostante quest'ultimo non utilizzi uno schema in tempo reale.

2.1 Il gioco

Le regole del gioco originale sono state lievemente semplificate per evitare eccessive complicazioni in fase di sviluppo, nello specifico abbiamo eliminato la meccanica dei poteri che avrebbe richiesto di strutturare dei meccanismi di valutazione complessi.

La logica del gioco è divisa in due strati paralleli, quello “fisico” rappresentato dal tabellone che viene man mano creato e descrive la condizione esatta del tabellone di gioco e quello “logico” composto dagli elementi del paesaggio, che ne valuta lo stato ed informa lo strato fisico del completamento degli elementi.

I due strati agiscono nelle due fasi del turno di un giocatore: lo strato fisico controlla che sia possibile piazzare la tessera pescata dove desidera il giocatore, garantendo che non venga violata la continuità del paesaggio; lo strato logico controlla la proprietà dei vari elementi del paesaggio presenti sulla tessera appena posizionata e permette o meno il piazzamento dei meeple, dopo di chè aggiorna i punteggi dei giocatori se rileva il completamento di un elemento di paesaggio.

Per valutare il completamento dei tre diversi elementi di paesaggio considerati vengono utilizzate tre tecniche specifiche:

Monasteri

sono completi quando tutte le otto tessere adiacenti sono state piazzate, è quindi sufficiente contare il numero dei vicini.

Strade

sono complete quando hanno incroci o città che le racchiudono, vengono contate quindi le tessere appartenenti alla strada contenenti questi due elementi, quando sono 2 la strada è completa⁴.

³Rimandiamo alla documentazione ufficiale di Carcassonne per ulteriori dettagli.

⁴Unico caso speciale è quando la strada si chiude su se stessa, in questa circostanza risulta completa.

Città

sono complete quando tutto il muro di cinta è chiuso e non sono presenti “buchi” all’interno, viene quindi tenuto il conto di quanti lati aperti sono rimasti su tutto il confine della città, quando questo numero scende a zero l’elemento è completo.

2.2 La comunicazione

L’aspetto principale dell’intero sistema risulta l’architettura paritaria della rete, questa è stata progettata utilizzando una topologia token ring. Questo tipo di topologia fa in modo che il gioco possieda un **ordine** tra i vari giocatori basato su un tiro di dado come si farebbe in una partita reale.

Come appena introdotto, la rete è stata inizialmente pensata come un *token ring*, il problema di questo approccio però risulta nell’aggiornamento dello schema di gioco: se la rete fosse completamente strutturata ad anello i pacchetti dovrebbero percorrerlo interamente per giungere a tutti i giocatori, oppure aspettare un intero turno di gioco per giungere al sistema di aggiornamento, complicando inutilmente il sistema.

Per questi motivi è stata successivamente scelta una topologia *completamente connessa*: anche se lo schema di gioco non presenta particolari requisiti di latenza, tutti i nodi contatteranno ogni volta il possessore del turno, il quale, terminata la sua azione, restituirà il controllo al nodo richiedente che potrà aggiornare il suo stato locale.



Figura 2: Tipologia token ring (sinistra) e tipologia completamente connessa (destra).

La tolleranza ai guasti di tipo crash è quindi stata progettata in quest’ottica: ogni nodo potrà accorgersi del crash di un altro giocatore all’istante (o al momento dell’interrogazione), e riconfigurerà l’ordine dell’anello in maniera appropriata.

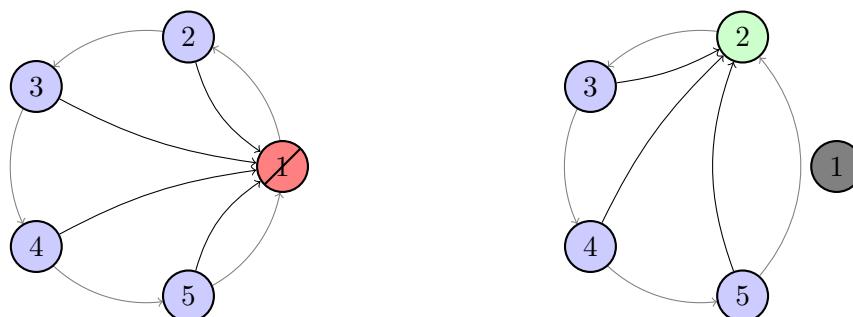


Figura 3: Riconfigurazione dei client dovuta ad un crash

3 Aspetti implementativi

3.1 Il desing pattern MVC

Il sistema è stato implementato utilizzando il design pattern *MVC*, Model View Controller, questo design pattern aiuta lo sviluppo di applicazioni “a camere stagne”, separando il modello (la logica e il motore di gioco) dalla grafica e dalla sua presentazione all’utente tramite un’entità denominata **controller**. Nello scenario del progetto questa entità è stata dotata della logica di gestione distribuita dello stato dei vari mondi di gioco.

3.2 Implementazione dello schema di gioco

In questo paragrafo vedremo brevemente l’implementazione dello schema e della logica di gioco. La figura 4 mostra il diagramma delle classi del componente **model** il quale racchiude tutte le funzionalità della logica di gioco.

Come anticipato dalla fase progettuale, il model implementa i meccanismi che permettono il collegamento logico delle tessere fra di loro, il rilevamento degli elementi completati (città, strade, monasteri) e l’assegnamento dei punti ai loro rispettivi proprietari.

3.3 L’interfaccia grafica

L’interfaccia grafica (componente **view**) è basata sul motore Slick2D⁵, questo mette a disposizione una libreria abbastanza semplice per realizzare interfacce grafiche anche complesse prive di rendering 3D.

L’intera esecuzione dell’interfaccia grafica è affidata ad un thread dedicato, una classe principale gestisce il dispatching degli aggiornamenti grafici e la lettura dell’input.

Il motore di gioco lancia la funzione di rendering (*refresh rate*) impostata, la classe principale delega dunque la composizione della schermata alla classe che implementa scena attualmente visualizzata. Parallelamente viene anche lanciata (con cadenza meno regolare) una funzione di aggiornamento (`void update()`) che interroga il controller e recupera nel thread della grafica gli eventuali aggiornamenti in coda (ad esempio tessere piazzate e punteggi segnati).

Alla stessa maniera, la gestione degli input viene passata alla scena corrente dalla classe principale in modo da interpretare le pressioni dei tasti e i movimenti del mouse nella maniera corretta.

3.3.1 Interazione

In figura 5 è mostrata una schermata di gioco, la parte centrale è occupata dal tavolo su cui vengono piazzate le tessere ed i meeple. Sui contorni della visuale di gioco vengono mostrati gli elementi dell’HUD (*Heads-Up Display*):

- in alto a sinistra i punteggi attuali dei partecipanti ancora in gioco;
- in alto a destra compare un pulsante per attivare e disattivare lo zoom quando lo scenario eccede i limiti dell’HUD stesso, in questo caso si può spostare la visuale muovendo il puntatore verso i bordi della schermata oppure utilizzando le frecce direzionali sulla tastiera;
- in basso a sinistra i meeple ancora in mano al giocatore.

⁵<http://slick.ninjacave.com/>

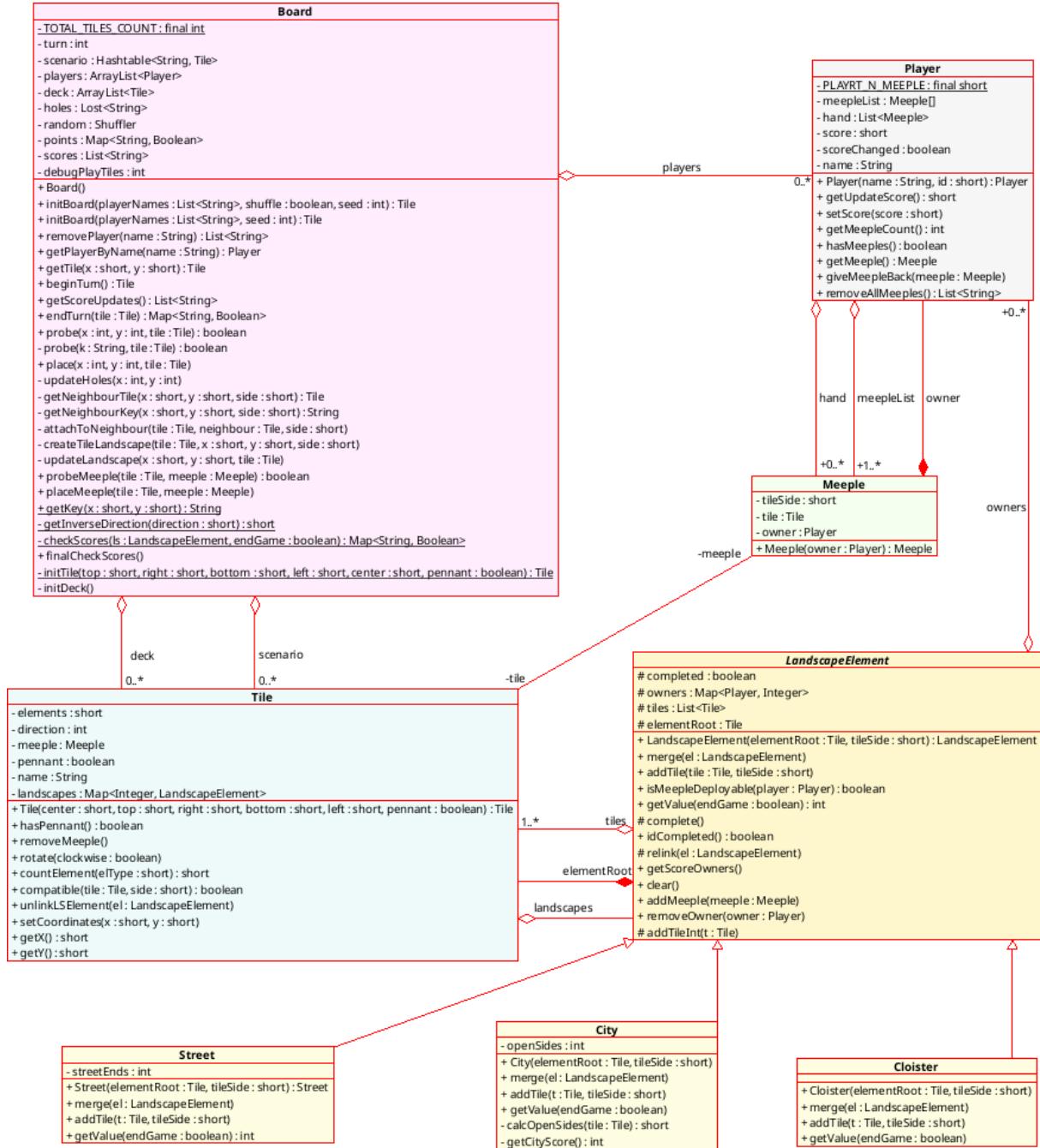


Figura 4: Diagramma delle classi del modello di gioco



Figura 5: Schermata di gioco

Inoltre se un giocatore è di turno l'HUD conterrà anche:

- in basso al centro il pulsante di confirma del piazzamento;
- in basso a destra la tessera da piazzare;
- *una volta confermato il piazzamento* (fig. 6) sulla tessera da piazzare vengono evidenziate le posizioni su cui è possibile mettere un meeple (ammesso che il giocatore ne abbia ancora in mano).

Il giocatore di turno, spostando il mouse sui confini dello scenario creato, può vedere un contorno nero che indica la possibilità di tentare un piazzamento. Un click sinistro del mouse effettuerà il tentativo, mostrando la tessera in trasparenza se il piazzamento è possibile o un effetto rosso in caso contrario. Utilizzando la rotellina del mouse o il click destro è possibile ruotare la tessera.

Una volta confermato il piazzamento si può decidere di mettere un meeple su un elemento della tessera appena piazzata cliccando sul contorno del meeple e lo si può rimuovere cliccando sul meeple stesso.

Al termine di un turno ogni giocatore vedrà i risultati del piazzamento effettuato, gli elementi del territorio completati vengono evidenziati e, se qualche giocatore li possedeva, il punteggio dei singoli elementi viene mostrato sopra agli stessi, come anche il piazzamento o la restituzione dei meeple.

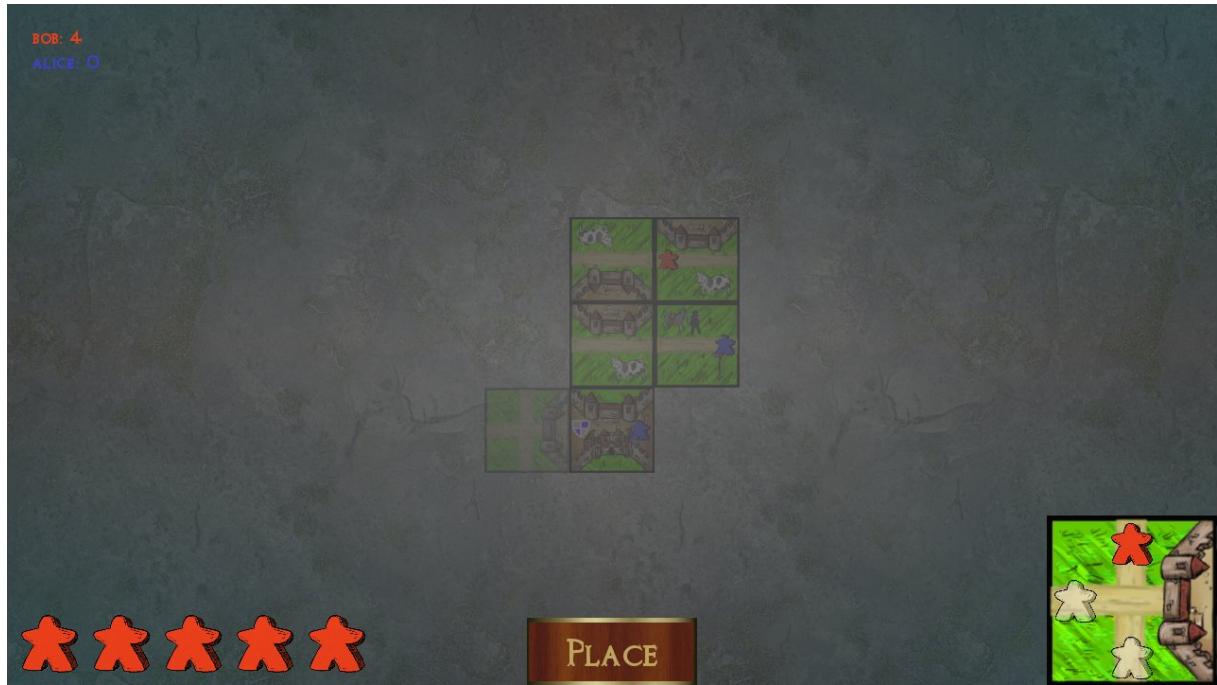


Figura 6: Schermata di gioco: piazzamento meeple

3.4 La gestione e la distribuzione della rete

3.4.1 Registrazione presso la lobby

Il primo passo svolto da ogni nodo di rete è la registrazione presso un server centralizzato comune, che implementa una o più “stanze” di gioco: le cosiddette lobby.

Questo server aspetta la registrazione del numero specificato di partecipanti, inviando loro la lista dei giocatori per poi lasciargli il pieno controllo, chiudendo la stanza.

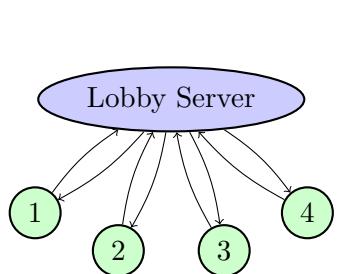


Figura 7: Registrazione presso il server lobby

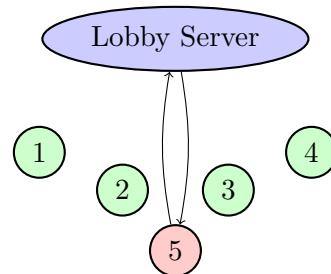


Figura 8: Eccezione del server alla richiesta di una lobby chiusa

3.4.2 Lo schema con leader dinamico.

Lo schema di distribuzione si basa sull’elezione di un leader “dinamico”, questo leader è deciso in base ad un lancio di un dado iniziale (nella realta’ implementato come un random intero a 32 bit). Il lancio viene distribuito tra i vari player, il giocatore con punteggio maggiore viene quindi dichiarato come leader corrente, e da lì a decrescere. Inoltre il valore del tiro di dado del

leader iniziale viene utilizzato da tutti i giocatori come seme per mescolare il mazzo di tessere, in modo da mantenere lo stato iniziale coerente.

La topologia risulta quindi una rete monodirezionale con passaggio del testimone (token ring) per la decisione del leader corrente.

3.4.3 Aggiornamenti allo stato

Uno degli aspetti più importanti del sistema di comunicazione del gioco è la presenza di classi di differenza (classe **TurnDiff**), le suddette classi rendono la comunicazione il più leggero possibile essendo composte da:

- Le coordinate della tessera posizionata;
- la rotazione della tessera posizionata;
- la posizione relativa del meeple se presente;
- il turno e il giocatore correnti.

| TurnDiff |
|--------------------------|
| - x : short |
| - y : short |
| - tileDirection : short |
| - meepleTileSide : short |
| - turn : short |
| - playerName : string |

3.4.4 Tolleranza ai guasti

Il sistema risulta tollerante ai guasti di tipo *crash*. Nel caso di un guasto di questo tipo sul nodo leader corrente, sarà il sistema ad invocare un'eccezione di tipo **RemoteException** verso i nodi interroganti, che lo elimineranno quindi dalla loro lista di interrogazione, riconfigurando l'anello.

Nel caso di un guasto di un nodo intermedio il risultato sarà il medesimo al momento di un'interrogazione da parte dei vari nodi dell'anello (e.g. quando il nodo crashato sarà di turno).

Questo genere di configurazione mantiene coerente lo stato locale delle diverse istanze, poichè ogni nodo aspetta la risoluzione delle varie mosse ad esso precedenti prima di essere interrogato a sua volta e poter agire.

Questo schema è banalmente possibile utilizzando una semplice rete di tipo token ring, ma è stato scelto di implementare il tutto come una sorta di cricca per l'aggiornamento automatico e la visualizzazione dei risultati con latenze brevi: se si fosse ponderato per una struttura completamente circolare (utilizzante lo stesso modello logico) gli aggiornamenti allo stato locale sarebbero applicati solamente dopo che il controllo (e quindi la leadership) fosse tornata al nodo richiedente, risultando in un'attesa pari ad **N-1** turni. Essendo il gioco in questione un gioco di logica non propriamente reattivo e con turni di gioco potenzialmente molto lunghi e riflessivi, abbiamo optato per un modello più pesante da un punto di vista di scambio di informazioni, ma, allo stesso tempo, più reattivo per tutti i client.

Le informazioni aggiuntive di numero di turno (clock logico) e nome del giocatore presenti nel diff vengono utilizzate insieme ad un id di gioco stabilito ad inizio partita per identificare alcuni problemi di consistenza: tutte le richieste verso il giocatore di turno specificano il clock e l'id della partita e vengono controllate da quest'ultimo prima di restituire il diff, allo stesso modo il diff deve contenere il nome del player che ha appena giocato e il clock logico del suo turno. Questo doppio controllo consente ai nodi di verificare con sufficiente confidenza che le informazioni contenute nel diff non siano frutto di un malfunzionamento e che i nodi richiedenti siano ancora sani.

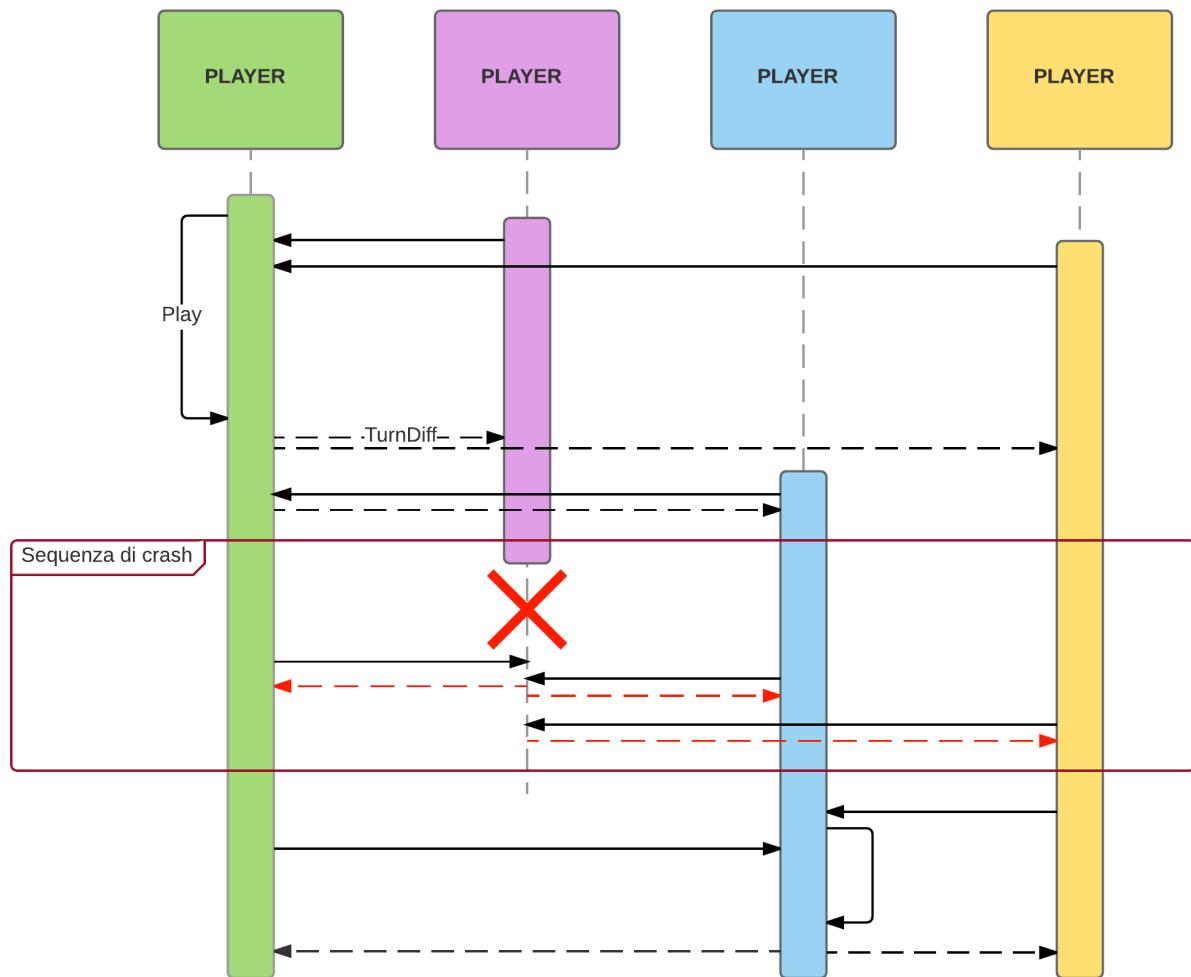


Figura 9: Comunicazione in una partita d'esempio a 4 giocatori

3.4.5 Gestione della critical section

Il modello presenta una critical section ad ogni interrogazione di un leader (ogni nodo interro-gante dovrà aspettare l'effettivo turno di gioco del leader), questa critical section è implementata come un **Lock** a barriera utilizzando la classe **ReentrantLock**.

Ogni nodo verrà bloccato fino all'avvenuto piazzamento della tessera e alla relativa creazione della struttura **TurnDiff**.

Questo modello di critical section locale non invalida il meccanismo di tolleranza ai guasti poichè se il processo leader subisce un **crash**, i vari richiedenti verranno liberati dalla critical section e gli verrà segnalato dal sistema di comunicazione il guasto.

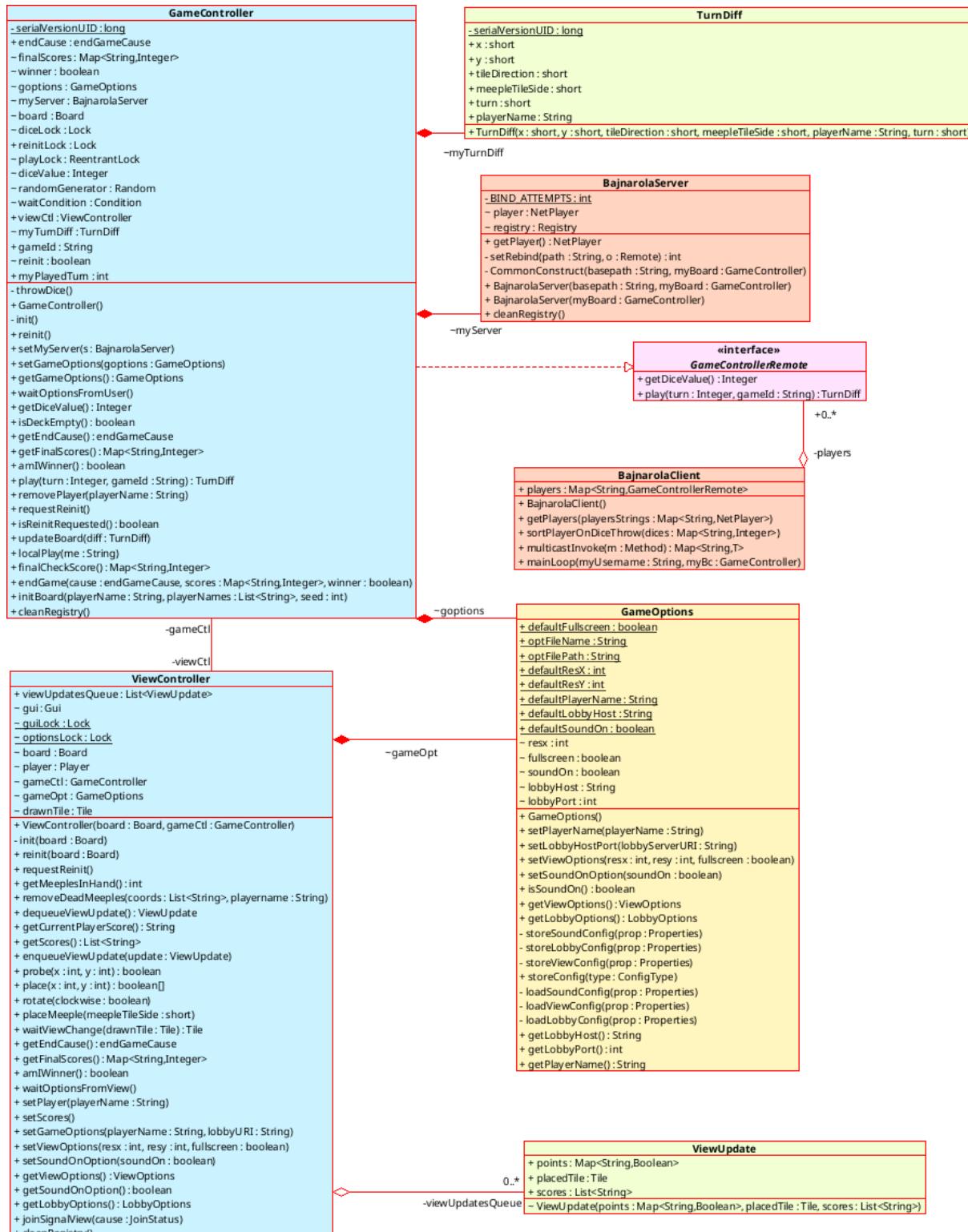


Figura 10: UML: Controller

4 Valutazioni Finali e Conclusioni

Al termine della fase di sviluppo abbiamo condotto diversi test al fine di valutare l'esperienza di gioco. In questa fase abbiamo appreso appieno l'importanza dell'utilizzo dei design pattern e dell'accurata strutturazione gerarchica del modello delle classi. In particolare il pattern MVC ci ha permesso di correggere facilmente gli errori di implementazione.

Ci siamo concentrati sulla comprensione della correttezza del comportamento delle comunicazioni tra i nodi della rete.

Dopo gli ultimi bugfix, non abbiamo più riscontrato problemi implementativi nella fase finale di test e le interazioni fra i giocatori funzionano come da aspettative. Inoltre l'esperienza di gioco risulta essere particolarmente fluida e piacevole grazie alle ottimizzazioni introdotte nell'interfaccia utente.

Per quanto riguarda l'affidabilità, la nostra implementazione è tollerante ai guasti di tipo crash. La rete logica in Bajnarola infatti si riorganizza automaticamente nel caso in cui uno o più nodi subiscano un crash, permettendo agli utenti ancora in gioco di concludere la partita ed essere notificati dei guasti.

In conclusione tramite questo lavoro abbiamo avuto l'occasione di approfondire il mondo dei sistemi distribuiti e allo stesso tempo siamo riusciti a raggiungere un risultato implementativo molto soddisfacente che verrà sicuramente arricchito da sviluppi futuri.