

Bajnarola

Progetto di sistemi distribuiti

Davide Berardi Matteo Martelli Marco Melletti
0000712698 0000702472 0000699715

3 dicembre 2015

Sommario

In questo documento verrà descritto *Bajnarola*, la versione videogame distribuita del gioco da tavolo *Carcassonne*. Particolare attenzione verrà attribuita alla progettazione e all'implementazione del sistema di comunicazione fra gli utenti, essendo uno degli aspetti principali di un software interattivo multiutente e con architettura distribuita. 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 però sono ancora rare implementazioni distribuite di giochi multiplayer le quali spesso sono invece basate su un architettura client server. In questo documento descriveremo il nostro lavoro di progettazione ed implementazione della versione software di un gioco da tavolo con particolare interesse riguardo l'architettura di rete distribuita utilizzata nella modalità multiplayer.

1.1 Carcassone

In particolare è stato nostro interesse occuparci del remake del gioco da tavolo Carcassonne. Quest'ultimo è un boardgame dei primi anni 2000 basato su tessere 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 o un monastero¹. Completando più città, strade o monasteri attraverso tali tessere, i giocatori (previsti da 2 a 5) accumulano i punti necessari a vincere la partita. Al fine di comprendere al meglio le prossime sezione di questo documento, verrà illustrato di seguito il funzionamento del gioco.

All'inizio della partita, una specifica tessera è posizionata sul tavolo, scoperta; le altre tessere sono mescolate e rimangono coperte nel mazzo e mischiate. 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

¹Si 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

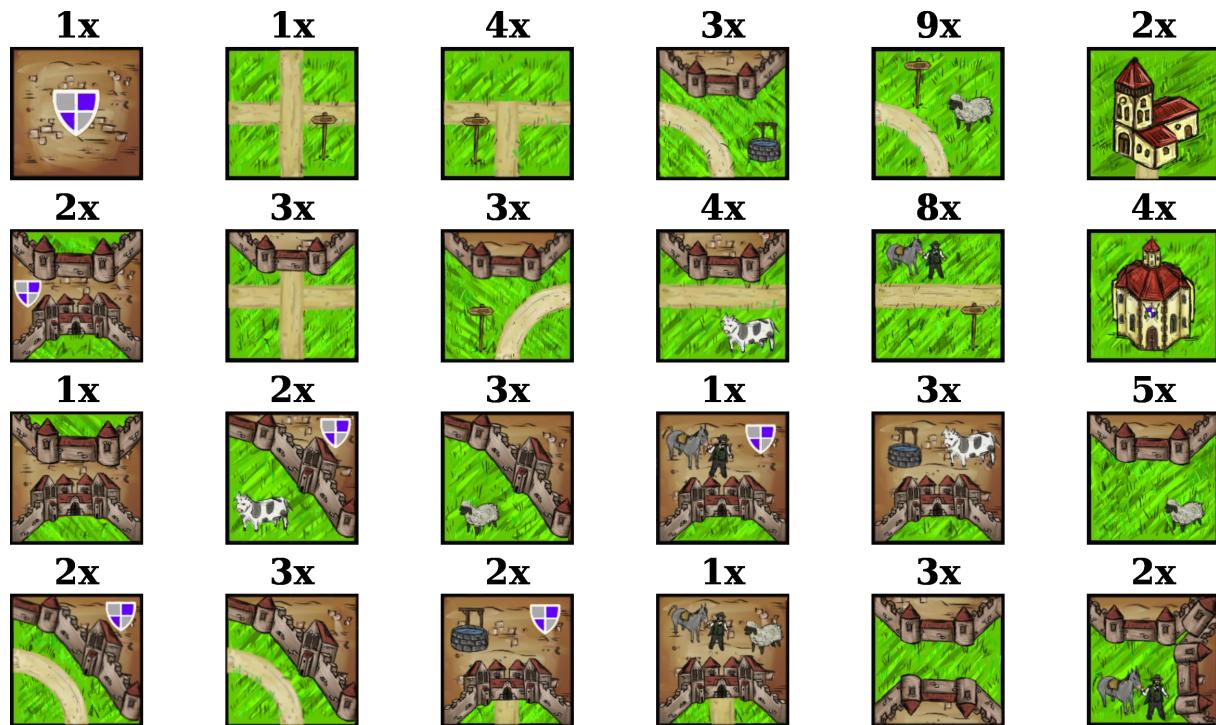


Figura 1: Elenco delle tessere presenti nel gioco

A turno, un giocatore per volta estrae una tessera dal mazzo e la posiziona scoperta sul tavolo a contatto con almeno una tessera già piazzata su uno o più lati, 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 *meeples* su un elemento del paesaggio della tessera appena posizionata, reclamandone la proprietà a patto che non sia già stato reclamato da un altro giocatore. Può accadere comunque che un elemento abbia più di un proprietario se diviene una congiunzione di due elementi dello stesso tipo non precedentemente adiacenti, in questo caso chi ha il numero maggiore di pedine sull'elemento rimane il proprietario, in caso di pareggio entrambi 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 chiuse, il proprietario di quell'elemento acquisisce i relativi punti e tutte le pedine che erano su quell'elemento vengono restituite ai relativi proprietari. Il punteggio di un elemento è dipendente dal numero e dal tipo di tessere che lo compongono.

Unica eccezione viene fatta per i poderi, che non vengono mai completati fino al termine della partita e sono soggetti ad un conteggio speciale.

Il gioco termina con il piazzamento dell'ultima tessera, al termine viene effettuato un conteggio finale dei punteggi derivanti dagli elementi non completati ma posseduti dai giocatori che avranno

un valore differente in questa fase; vince il giocatore che, dopo il conteggio finale, ha totalizzato più punti.

Rimandiamo alla documentazione ufficiale di Carcassonne per ulteriori dettagli.

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 hanno 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).

Vedremo nelle prossime sezioni i dettagli progettuali ed implementativi. In ultimo forniremo una analisi valutativa dei risultati ottenuti dai test con le dovute considerazioni finali.

2 Aspetti progettuali

Il gioco in questione è stato progettato nell'ottica di un sistema resistente ai guasti, come richiesto da progetto.

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 realtime.

2.1 Il gioco

Le regole del gioco originale sono state vagamente 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 quindi nelle due fasi del turno di un giocatore: il primo controlla che sia possibile piazzare la tessera pescata dove desidera il giocatore garantendo che non venga violata la continuità del paesaggio, il secondo controlla la proprietà dei vari elementi del paesaggio della tessera appena piazzata e permette o meno il piazzamento dei meeple, dopo di che 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, e' 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 che contengono un incrocio o una strada che finisce in una città, quando queste sono 2 l'elemento è completo.

Città

sono complete quando tutto il muro di cinta e' chiuso, viene tenuto quindi il conto di quanti lati aperti sono rimasti su tutto il confine della città, quando questo numero scende a zero l'elemento e' completo.

2.2 La comunicazione

L'aspetto principale dell'intero sistema risulta l'architettura paritaria dell'intera rete, questa e' stata progettata utilizzando una topologia token ring, questo tipo di topologia fa in modo che il gioco possieda un **ordine** tra i vari giocatori in base ad un tiro di dado come si farebbe in una partita reale.

Come appena introdotto, la rete e' stata pensata come un **token ring**, il problema di questo approccio pero' risulta nell'aggiornamento dello schema di gioco; se la rete fosse completamente token ring i pacchetti dovrebbero percorrere l'intero anello 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 e' stata scelta una topologia **completamente connessa**: anche se lo schema di gioco non presenta particolari requisiti di latenza, ogni nodo contattera' ogni volta il nodo possedente il turno, il quale terminata la sua azione restituira' il controllo al nodo richiedente che potra' aggiornare il suo stato locale.

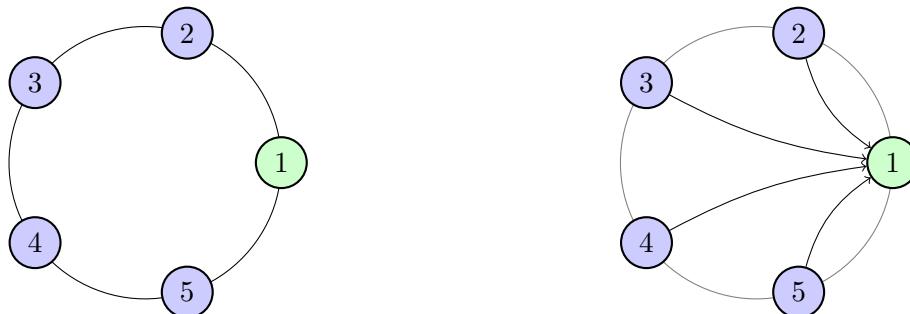


Figura 2: Tipologia token ring (sinistra) e tipologia completamente connessa (destra).

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

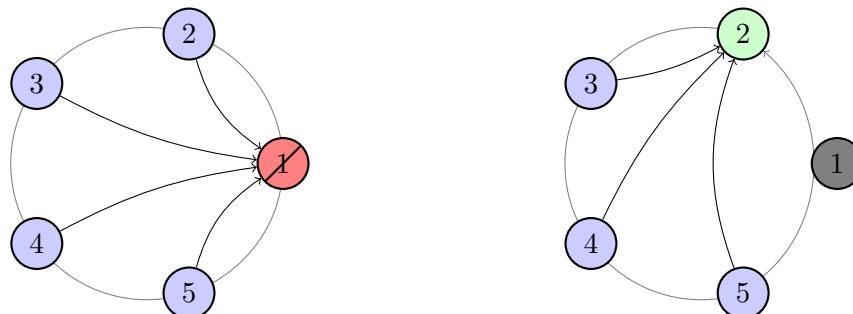


Figura 3: Riconfigurazione dei client dovuta ad un crash

3 Aspetti implementativi

3.1 Il desing pattern MVC

Il sistema e' 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'entita' denominata **controller**, nello scenario del progetto questa entita' e' 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.

Nello specifico il model contiene tutte le funzionalità che permettono il collegamento logico delle tessere fra di loro. Inoltre la logica, come anticipato dalla fase progettuale, implementa i meccanismi per rilevare il completamento degli elementi (città, strade, monasteri) ed assegnare i punti ai loro rispettivi proprietari.

3.3 L'interfaccia grafica

L'interfaccia grafica e' basata sul motore Slick2D??, questo mette a disposizione una libreria abbastanza semplice per realizzare interfacce grafiche anche complesse prive di rendering 3D.

L'intera esecuzione dell'interfaccia grafica e' 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 alla frequenza impostata, la classe principale delega dunque la composizione della schermata alla scena attualmente caricata. In parallelo viene anche lanciata (con cadenza meno regolare) una funzione di aggiornamento (`void update()`) che interroga il controller e recupera nel thread della grafica gli aggiornamenti pendenti, se ce ne sono (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 e' mostrata una schermata di gioco, la parte centrale e' 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 puo' 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.

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

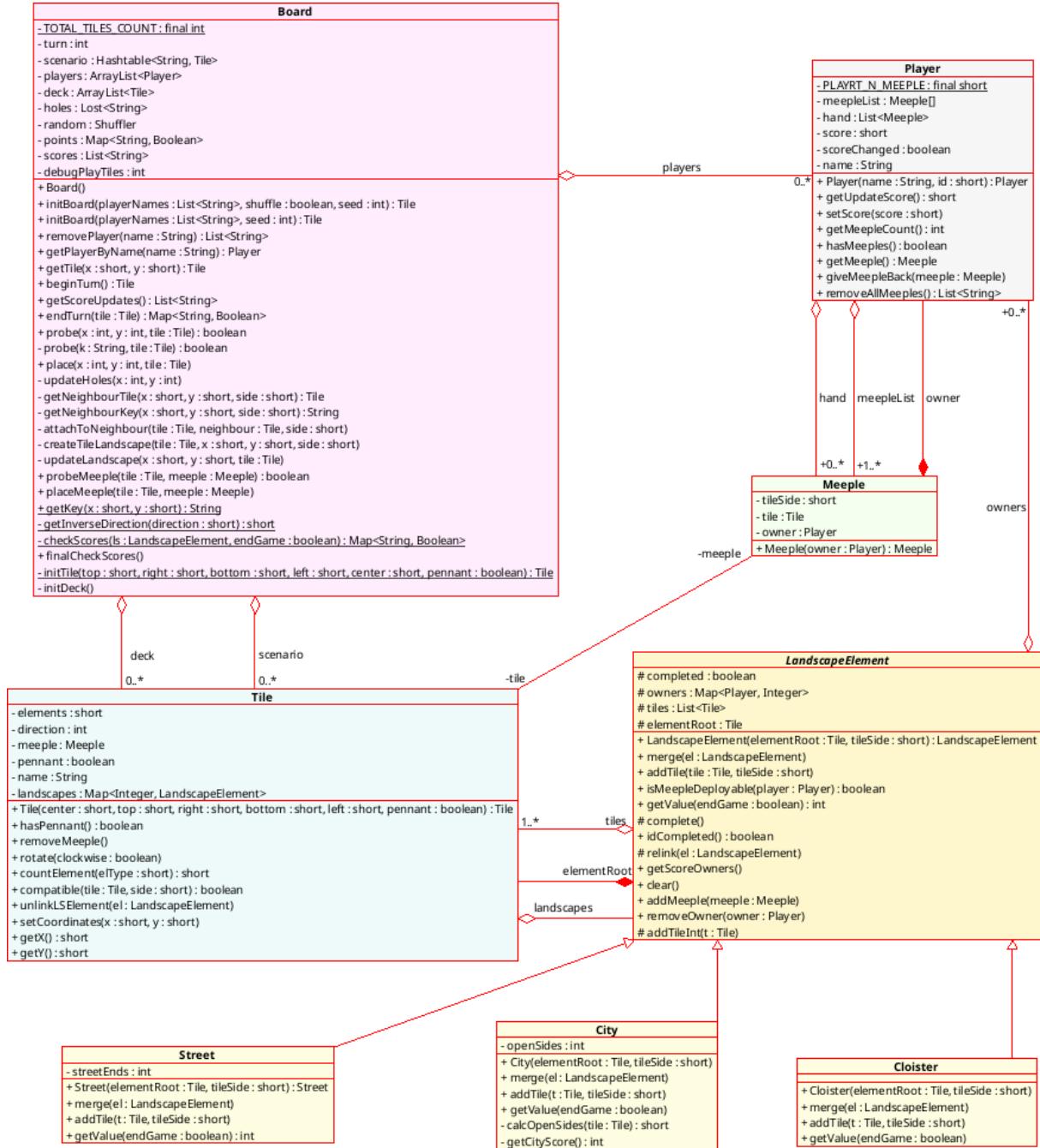


Figura 4: Diagramma delle classi del modello di gioco



Figura 5: Schermata di gioco

- 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 e' 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, puo vedere un contorno nero che indica la possibilita' di tentare un piazzamento. Un click sinistro del mouse effettuera' il tentativo, mostrando la tessera in trasparenza se il piazzamento e' possibile o un effetto rosso in caso contrario. Utilizzando la rotellina del mouse o il click destro e' possibile ruotare la tessera.

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

Al termine di un turno ogni giocatore vedra' 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 agli stessi, come anche il piazzamento o la restituzione dei 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 e' la registrazione presso un server centralizzato comune, che implementa una o piu' "stanze" di gioco; le cosiddette lobby.

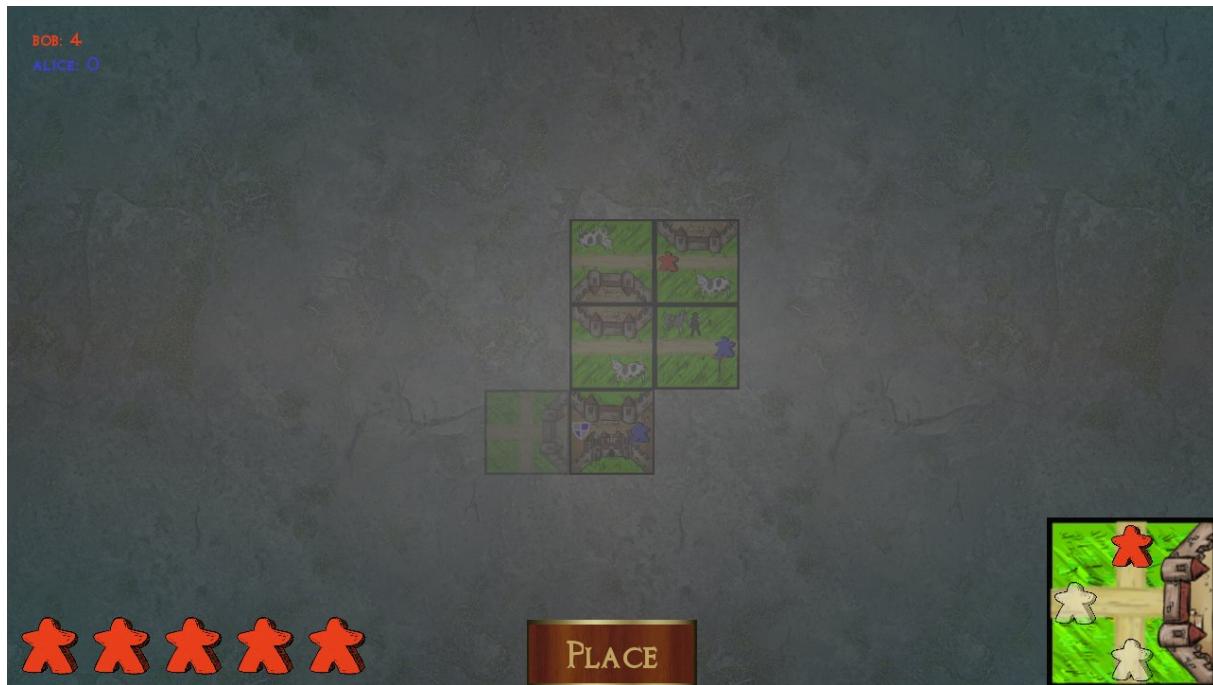


Figura 6: Schermata di gioco: piazzamento meeple

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.

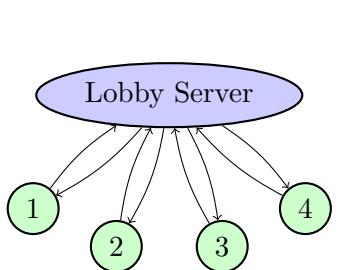


Figura 7: Registrazione presso il server lobby

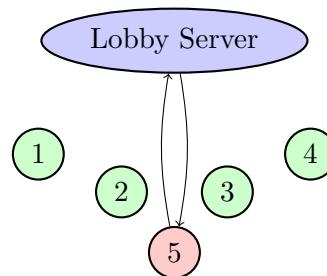


Figura 8: Eccezione del server alla richiesta di una lobby chiusa

3.4.2 Lo schema con leader dinamico.

Lo schema di distribuzione si basa sull'elezione di un leader "dinamico", questo leader e' deciso in base ad un lancio di un dado iniziale (nella realta' implementato come un random intero a 32 bit, per evitare lanci ripetuti). Il lancio viene distribuito tra i vari player, il giocatore con punteggio maggiore viene dichiarato come leader corrente, e da li 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 piu' importanti del sistema di comunicazione del gioco e' la presenza di classi di differenza (classe **TurnDiff**), le suddette classi rendono la comunicazione il piu' leggero possibile essendo composte da:

- Le coordinate della tessera;
- la rotazione della tessera posizionata;
- le posizioni relative del meeple se presente;
- il turno e il giocatore correnti.



3.4.4 Tolleranza ai guasti

Il sistema risulta tollerante ai guasti di tipo crash. Nel caso di un guasto di tipo crash sul nodo leader corrente, sara' 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 sara' il medesimo al momento di un'interrogazione da parte dei vari nodi dell'anello (e.g. quando il nodo crashato sara' di turno). Questo genere di configurazione mantiene coerente lo stato locale delle diverse istanze, poiche' ogni nodo aspetta la risoluzione delle varie mosse ad esso precedenti prima di essere interrogato a sua volta e poter agire.

Questo schema e' banalmente possibile utilizzando una semplice rete di tipo token ring, ma e' 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 piu' pesante da un punto di vista di scambio di informazioni, ma, allo stesso tempo, piu' reattivo per tutti i client, che comunque non e' pesante per la rete.

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 guasti arbitrari: 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.

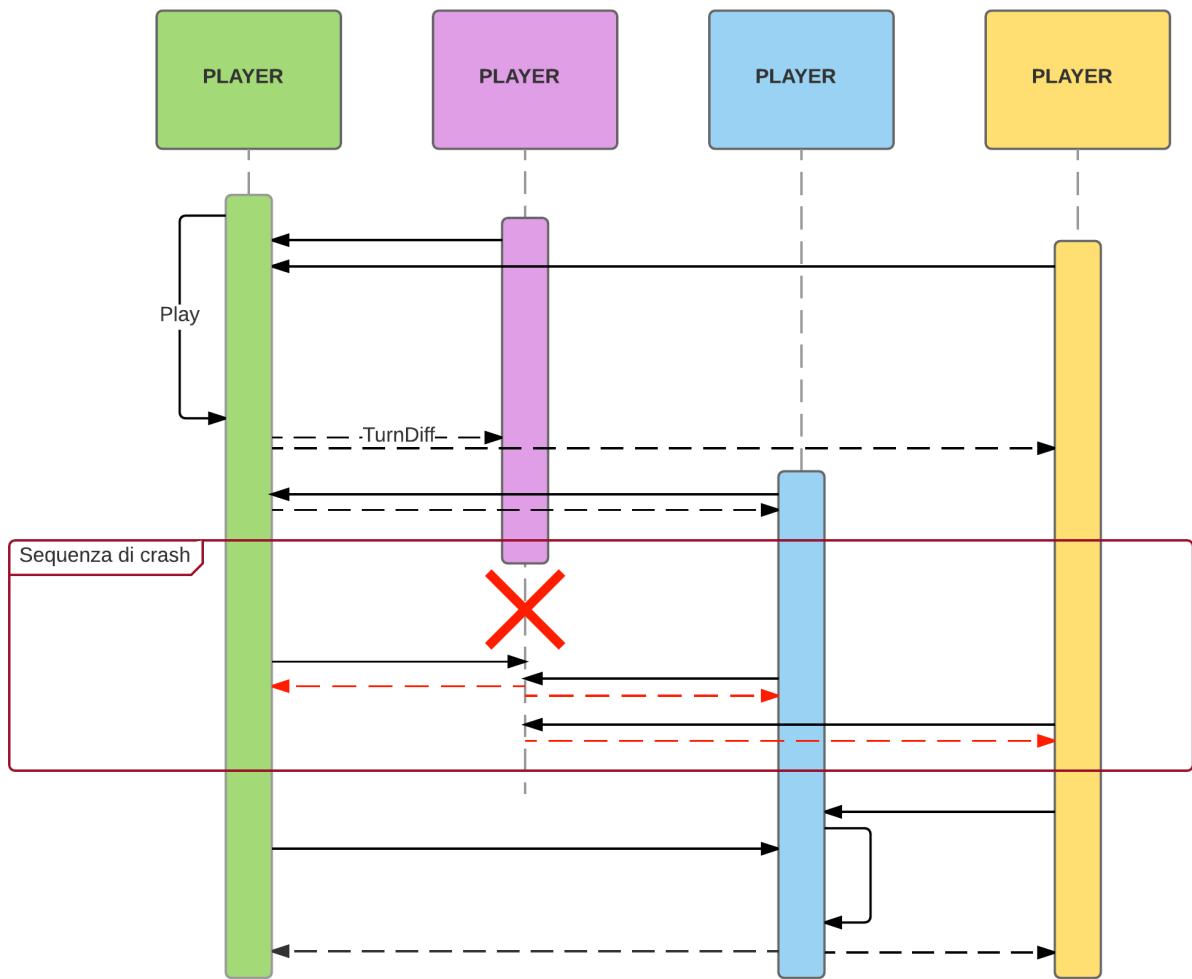


Figura 9: Comunicazione in una partita d'esempio a 4 giocatori

3.4.5 Gestione della critical section

Il modello presenta una critical section ad ogni interrogazione di un leader (ogni nodo interrogante 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 poiché 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.

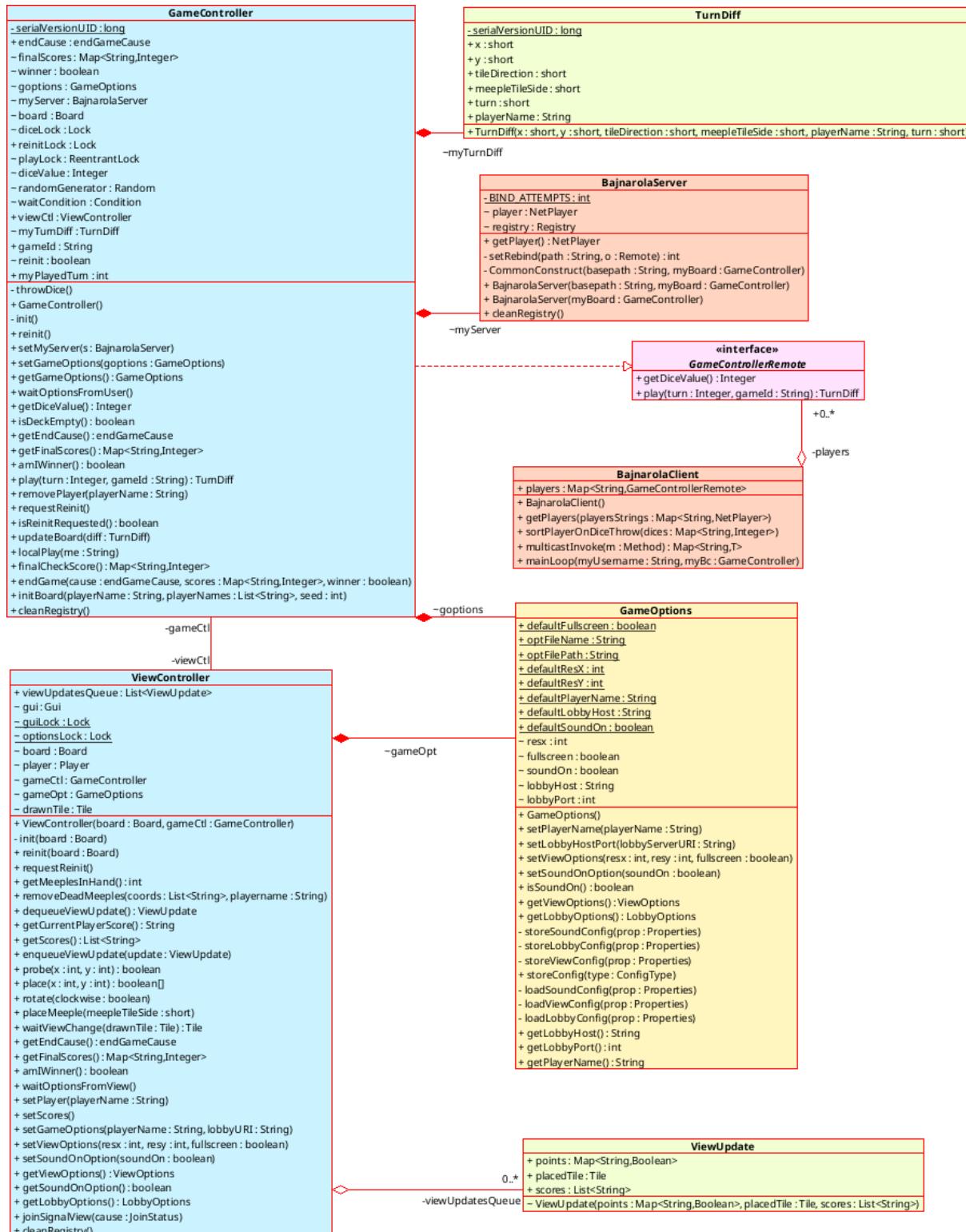


Figura 10: UML: Controller

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.

Nello specifico ci siamo concentrati principalmente nel comprendere se il comportamento delle comunicazioni tra i nodi della rete fosse corretto.

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 terminare la partita o di concluderla nel caso rimanga un solo giocatore.

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.