Bajnarola

Progetto di sistemi distribuiti

Davide Berardi 0000712698 | Matteo Martelli 0000702472 | Marco Melletti 0000699715

1 dicembre 2015

Sommario

Bajnarola e' un bel gioco

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 nello specifico 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 monastero [1]. Completando quindi più città, strade, ecc, 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, vedremo brevemente di seguito il funzionamento del gioco. All'inizio della partita, una singola tessera è posizionata sul tavolo, scoperta; le altre tessere sono posizionate 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

[1] Si fa riferimento alla prima versione del gioco in cui non sono presenti fiumi

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

A turno, i giocatori estraggono una tessera dal mazzo e la posizionano scoperta sul tavolo in contatto con le tessere già piazzate attraverso uno o più lati in modo coerente con le altre, in modo da proseguire eventuali strade, campi, o mura già presenti.

Dopo aver posizionato la tessera, il giocatore può decidere di piazzare una pedina detta *meeple* su di essa che reclama la proprietà di un elemento di terreno e non può essere piazzato su un elemento già reclamato da un altro meeple. Può accadere comunque che un elemento possegga più di un proprietario se diviene una congiunzione di due elementi dello stesso tipo non precedentemente adiacenti. 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. Il punteggio di un elemento è dipendente dal numero e dal tipo di tessere che compongono l'elemento.

Il gioco termina con il piazzamento dell'ultima tessera; vince il giocatore che 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. Quest'ultimi infatti

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

2.2 La comunicazione

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

3.3 L'interfaccia grafica

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, implementante diverse "stanze" di gioco; le cosiddette lobby.
Questo server aspettera' quindi la registrazione del numero specificato di partecipanti, inviando loro la lista dei giocatori per poi lasciare il pieno controllo ad essi, chiudendo la stanza.

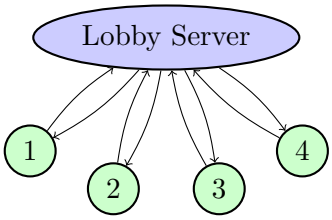


Figura 1: Registrazione presso il server lobby

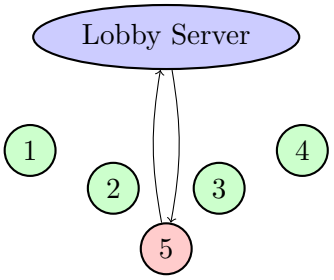
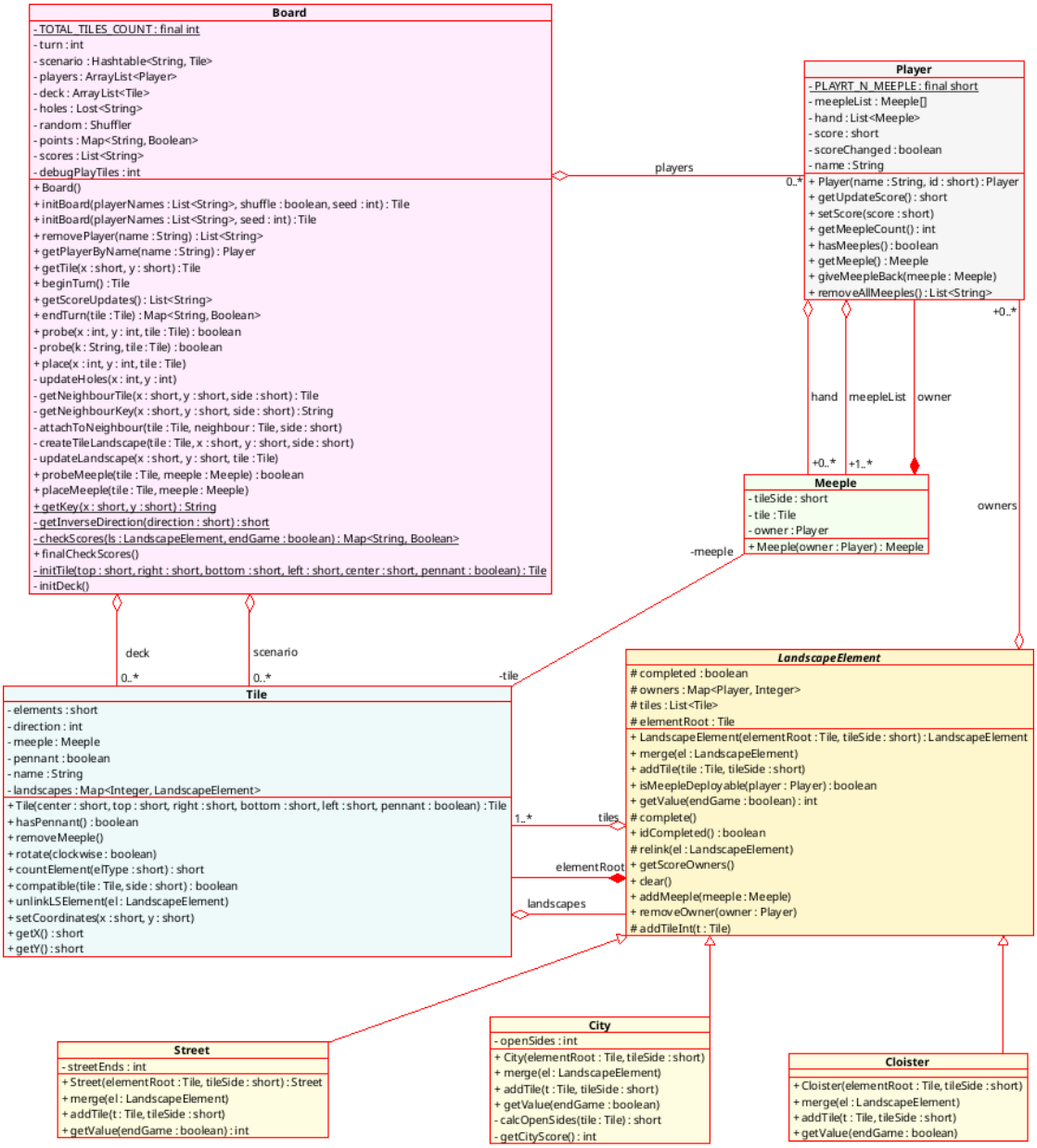
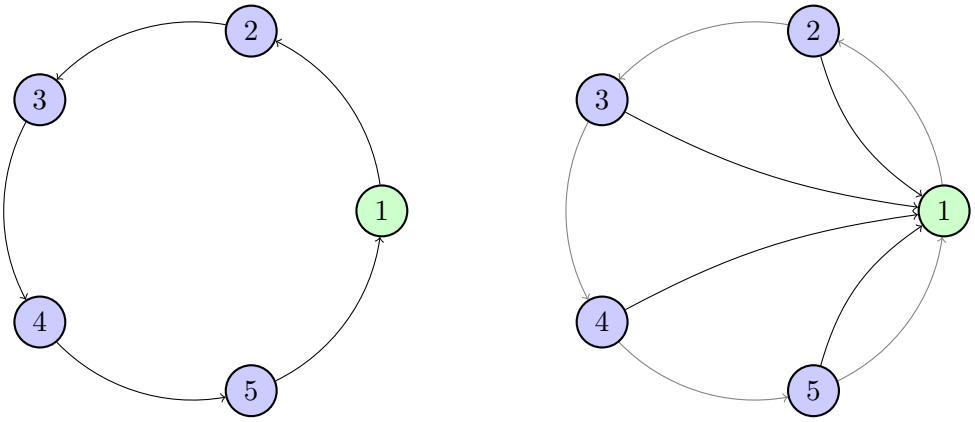


Figura 2: Eccezione del server alla richiesta di una lobby chiusa

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), questo lancio viene quindi distribuito dai vari player ad ogni giocatore, creando un **ordine** di gioco, il giocatore con punteggio maggiore verra' quindi dichiarato come leader corrente, e da li a decrescere.
La topologia risulta quindi una rete monodirezionale con passaggio del testimone (token ring) per la decisione del leader corrente.


Board
- TOTAL_TILES_COUNT : final int
- turn : int
- scenario : Hashtable<String, Tile>
- players : ArrayList<Player>
- deck : ArrayList<Tile>
- holes : Lost<String>
- random : Shuffler
- points : Map<String, Boolean>
- scores : List<String>
- debugPlayTiles : int
+ Board()
+ initBoard(playerNames : List<String>, shuffle : boolean, seed : int) : Tile
+ initBoard(playerNames : List<String>, seed : int) : Tile
+ removePlayer(name : String) : List<String>
+ getPlayerByName(name : String) : Player
+ getTile(x : short, y : short) : Tile
+ beginTurn() : Tile
+ getScoreUpdates() : List<String>
+ endTurn(tile : Tile) : Map<String, Boolean>
+ probe(x : int, y : int, tile : Tile) : boolean
- probe(k : String, tile : Tile) : boolean
+ place(x : int, y : int, tile : Tile)
- updateHoles(x : int, y : int)
- getNeighbourTile(x : short, y : short, side : short) : Tile
- getNeighbourKey(x : short, y : short, side : short) : String
- attachToNeighbour(tile : Tile, neighbour : Tile, side : short)
- createTileLandscape(tile : Tile, x : short, y : short, side : short)
- updateLandscape(x : short, y : short, tile : Tile)
+ probeMeeple(tile : Tile, meeple : Meeple) : boolean
+ placeMeeple(tile : Tile, meeple : Meeple)
+ getKey(x : short, y : short) : String
- getInverseDirection(direction : short) : short
- checkScores(ls : LandscapeElement, endGame : boolean) : Map<String, Boolean>
+ finalCheckScores()
- initTile(top : short, right : short, bottom : short, left : short, center : short, pennant : boolean) : Tile
- initDeck()
players
0..*
Player
- PLAYRT_N_MEEPLE : final short
- meepleList : Meeple[]
- hand : List<Meeple>
- score : short
- scoreChanged : boolean
- name : String
+ Player(name : String, id : short) : Player
+ getUpdateScore() : short
+ setScore(score : short)
+ getMeepleCount() : int
+ hasMeeples() : boolean
+ getMeeple() : Meeple
+ giveMeepleBack(meeple : Meeple)
+ removeAllMeeples() : List<String>
+0..*
hand
meepleList
owner
+0..*
+1..*
Meeple
- tileSide : short
- tile : Tile
- owner : Player
+ Meeple(owner : Player) : Meeple
owners
-meeple
deck
0..*
scenario
0..*
-tile
Tile
- elements : short
- direction : int
- meeple : Meeple
- pennant : boolean
- name : String
- landscapes : Map<Integer, LandscapeElement>
+ Tile(center : short, top : short, right : short, bottom : short, left : short, pennant : boolean) : Tile
+ hasPennant() : boolean
+ removeMeeple()
+ rotate(clockwise : boolean)
+ countElement(elType : short) : short
+ compatible(tile : Tile, side : short) : boolean
+ unlinkLSElement(el : LandscapeElement)
+ setCoordinates(x : short, y : short)
+ getX() : short
+ getY() : short
1..*
tiles
elementRoot
landscapes
LandscapeElement
completed : boolean
owners : Map<Player, Integer>
tiles : List<Tile>
elementRoot : Tile
+ LandscapeElement(elementRoot : Tile, tileSide : short) : LandscapeElement
+ merge(el : LandscapeElement)
+ addTile(tile : Tile, tileSide : short)
+ isMeepleDeployable(player : Player) : boolean
+ getValue(endGame : boolean) : int
# complete()
+ idCompleted() : boolean
# relink(el : LandscapeElement)
+ getScoreOwners()
+ clear()
+ addMeeple(meeple : Meeple)
+ removeOwner(owner : Player)
# addTileInt(t : Tile)
Street
- streetEnds : int
+ Street(elementRoot : Tile, tileSide : short) : Street
+ merge(el : LandscapeElement)
+ addTile(t : Tile, tileSide : short)
+ getValue(endGame : boolean) : int
City
- openSides : int
+ City(elementRoot : Tile, tileSide : short)
+ merge(el : LandscapeElement)
+ addTile(t : Tile, tileSide : short)
+ getValue(endGame : boolean)
- calcOpenSides(tile : Tile) : short
- getCityScore() : int
Cloister
+ Cloister(elementRoot : Tile, tileSide : short)
+ merge(el : LandscapeElement)
+ addTile(t : Tile, tileSide : short)
+ getValue(endGame : boolean)



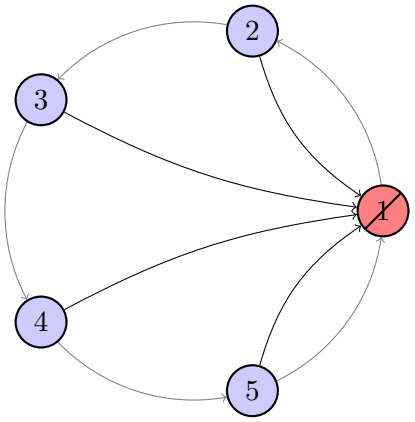
3.4.3 Aggiornamenti allo stato

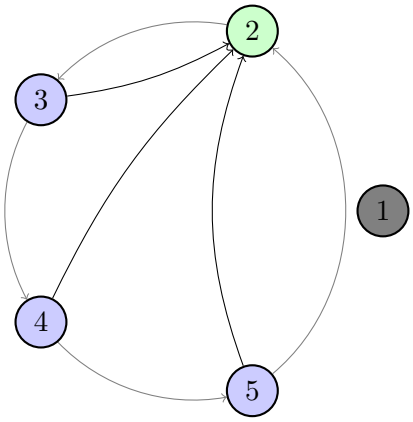
Uno degli aspetti piu' importanti del sistema di comunicazione del gioco e' la presenza di classi di differenza (classe **TurnDiff**), le suddette classi rendono la comunicazione il piu' leggero possibile essendo composte di:

- La rotazione della tessera posizionata;
- le coordinate della tessera;
- le posizioni relative del meeple se presente.

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.
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 e' stato optato per un modello piu' pesante da un punto di vista di scambio di informazioni che da un punto di vista piu' leggero come comunicazioni ma, allo stesso tempo, meno reattivo per tutti i client.





4 Valutazione

5 Conclusioni