Progetto di Sistemi Distribuiti
"Almost-Othello" multiplayer

di G. Benvenuti, G. Ledonne, L. Leoni
{benvenut | ledonne | lleoni} @ cs.unibo.it

13 settembre 2011

Questo documento è la relazione del progetto ed è composto sostanzialmente da tre parti: introduzione al gioco e spiegazione dettagliata delle regole, scelte effettuate in fase di progettazione - p.e. stato dei giocatori, informazioni condivise, gestione dei guasti - e strategie implementative - strutturazione delle classi e librerie utilizzate. A queste se ne aggiunge una dedicata a class e sequence diagram UML. Il gioco che è stato scelto è la versione multiplayer dell'antico Reversi (gioco per due giocatori), leggermente riadattato per rendere quanto possibile paritaria la coesistenza di più di due giocatori.

1. Introduzione

Si vuole sviluppare un gioco multiplayer distribuito con $N \geq 3$ giocatori, tutti pari fra loro, che condividono uno stato globale rappresentato da una othelliera[1]. Il sistema deve tollerare almeno due guasti di tipo crash (sebbene il nostro progetto ne tolleri $N-1$). Si potrà contare su canali di comunicazine affidabili, e verrà utilizzato RMI, e non i socket, per la comunicazione fra i nodi. L'unica componente centralizzata sarà la registrazione iniziale al gioco, che a partita iniziata non sarà più necessaria.

2. Regole del gioco

Come accennato in precedenza, il gioco scelto si ispira molto a Othello, un classico per due giocatori.

Da regolamento Othello non è giocabile da un numero di giocatori diverso da due, quindi si è deciso di modificarlo applicando il seguente insieme di regole:

- Disposizione iniziale random delle pedine, tre per giocatore.
- Pedine identificate da colori diversi.
- Ogni pedina ha a disposizione due tipi di mosse: conquista e colonizza:

[1] Tale definizione è quella riconosciuta all'interno del regolamento ufficiale del gioco

· Conquista (c_1) consiste nel porre una pedina del proprio colore in uno degli spazi vuoti in modo che una o più pedine di uno o più avversari, situate perpendicolarmente, orizzontalmente, e/o diagonalmente, rimangano chiuse tra quest'ultima ed un'altra preesistente del proprio colore. Non devono essere presenti pedine vuote in mezzo al percorso.

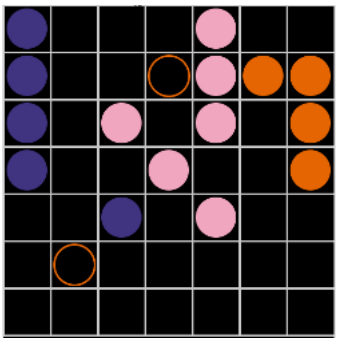

Figure 1: Esempio di mossa conquista

· Colonizza (c_2) è l'occupazione di una casella vuota adiacente a una occupata da una pedina del proprio colore.

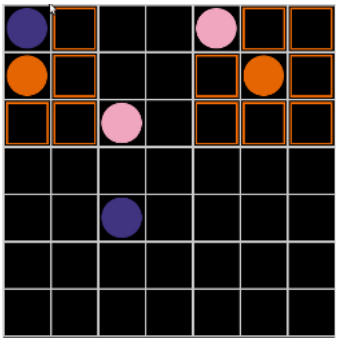

Figure 2: Esempio di mossa colonizza

- Priorità delle mosse: $c_1 > c_2$: quando è possibile fare una conquista è obbligatorio procedere in tale modo. Se invece, come spesso si verifica nelle fasi iniziali del gioco, i giocatori sono impossibilitati a conquistare, allora si può procedere con una mossa di colonizzazione.
- Il giocatore che non può effettuare né conquiste né colonizzazioni è costretto a cedere il proprio turno di gioco.
- Quando nessun giocatore potrà effettuare ulteriori mosse, il gioco si dirà terminato:

- Il vincitore è colui che avrà collezionato il maggior numero di pedine del proprio colore.
- Nel caso in cui il numero maggiore di pendine venga collezionato da due o più giocatori, allora il gioco terminerà con un pareggio.

- Per ogni giocatore che abbandona il gioco, le pedine possedute resteranno sul campo (diventando di colore grigio) e potranno essere conquistate in ossequio alle regole precedentemente introdotte.

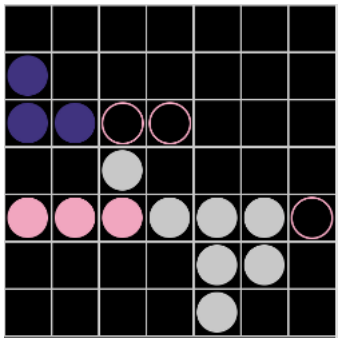

Figure 3: Esempio della board qualora vi siano pedine di un giocatore che ha abbandonato la partita

- In caso di $N-1$ abbandoni il vincitore sarà il giocatore rimasto.
- Il vincitore paga da bere a tutti.

Sono necessarie alcune precisazioni:

- Una conquista può avere la conseguenza di far perdere tutte le pedine ad un giocatore, che sarà dichiarato sconfitto.
- La conquista riguarda esclusivamente le pedine imprigionate *direttamente* fra la pedina messa sulla scacchiera e la prima pedina del medesimo colore posta sulla stessa riga, colonna e/o diagonale.
- Questo insieme di regole non garantisce la fairness del gioco, poiché la disposizione iniziale random potrebbe favorire alcuni giocatori a discapito di altri.[2]

3. Scelte progettuali

Il sistema è stato realizzato implementando due paradigmi di comunicazione: client-server e anello unidirezionale.

[2] Al contrario la fairness di sistema è garantita dal fatto che tutti i nodi godono di un trattamento paritario.

Il primo è stato utilizzato unicamente per realizzare il servizio centralizzato di registrazione al gioco. Il secondo, invece, per realizzare le comunicazioni e il sistema di recovery dai guasti.

Come detto in precedenza, la registrazione è stata realizzata sfruttando il modello client-server. Il server è rappresentato dal servizio che raccoglie le richieste di partecipazione e provvede alla consegna della lista ordinata dei partecipanti una volta raggiunto il numero di giocatori stabilito. I client, di conseguenza, sono tutti coloro che intendono partecipare al gioco. Il server non è un nodo a se stante, bensì viene inizializzato assieme a una delle macchine partecipanti e il suo indirizzo dovrà quindi essere fornito a tutti gli altri a runtime.

Successivamente a questa fase il sistema funziona secondo il modello ad anello unidirezionale. Ogni nodo, infatti, può comunicare solo e soltanto con il suo successore nella lista e riceve messaggi esclusivamente dal suo predecessore (minimal "a priori" knowledge). I nodi condividono uno stato globale, la othelliera, e annesse informazioni (caselle libere, caselle occupate, proprietari delle caselle, ecc). Affinché lo stato globale sia sempre coerente, le operazioni di update sono consentite solo al giocatore che possiede il token in quel momento - il giocatore a cui tocca effettuare la mossa.
Il nodo che effettua una mossa inoltra un messaggio e attende che questo gli venga riconsegnato. Questo messaggio, effettuando un giro completo dell'anello, sarà visibile a tutti i nodi intermedi, i quali potranno leggerne e processarne il contenuto al fine di mantenere coerente lo stato globale. Al termine del turno di gioco il nodo passerà il token al suo successore.

Il sistema di recovery dai guasti è stato realizzato al fine di poter rilevare i crash attraverso l'esecuzione di due task: controllare che tutte le comunicazioni da un nodo verso il successivo vengano effettuate con successo e verificare che lo stesso non si andato in crash anche in situazioni nelle quali non sono previste comunicazioni inerenti alle fasi di gioco.

4. Strategie implementative

Il gioco è state realizzato mediante l'impementazione di una rete ad anello unidirezionale. Tale scelta rappresenta l'opzione più appropriata per la realizzazione di un gioco a turni, nonostante non sia allo stesso tempo la scelta più consona per la realizzazione di un sistema distribuito.

Affinchè i nodi possano comunicare seguendo tale paradigma, è necessario che condividano le informazioni riguardo gli altri nodi del sistema: la classe PlayerList risponde a tale esigenza.

PlayerList rappresenta la lista ordinata dei giocatori, ed è restituita a tutti i giocatori al termine della fase di registrazione. Al suo interno sono presenti un numero variabile di oggetti di tipo Player. Ogni oggetto Player rappresenta la carta d'identità del singolo nodo, poiché al suo interno sono presenti tutti i dati che lo rappresentano. Infatti abbiamo informazioni presentazionali come il nome, di comunicazione come indirizzo IP e porta, nonché uno UUID per la sua identificazione univoca. All'interno di PlayerList, realizzata come estensione

della classe LinkedList, sono presenti un numero considerevole di metodi di ricerca/aggiornamento della suddetta lista.

Il modello di messaging prevede due famiglie di messaggi:

- Cambiamento dello stato di gioco: sono i messaggi di comunicazione della mossa e di passaggio del token.
- Gestione dei guasti: sono i messaggi di ping generati periodicamente e quello di notifica dei crash.

Cambiamento stato gioco.

Prima di introdurre le classi e i metodi che si occupano delle comunicazioni inerenti al cambimento dello stato di gioco, è necessario effettuare una breve introduzione su altre classi, le quali hanno il compito di garantire la corretta operabilità funzionale del sistema (anche dal punto di vista del regolamento). La prima di questa classi è Othello: come si evince dal nome questa classe è fondamentale perchè si occupa della gestione della grafica, esegue il controllo di flusso, ossia quell'insieme di operazioni che consente di stabilire quando il giocatore può effetuare le proprie mosse, la verifica delle condizioni di fine gioco e la modifica dello stato (ossia la modifica dell'oggetto singleton Board) secondo le regole del gioco. Tali regole sono state separate dalla Board e sono state implementate nella classe BoardLogic in ossequio al principio di separazione della politica dai meccanismi che garantisce la giusta separazione fra la logica del gioco e lo stato condiviso. Al termine di tale processo all'interno di Othello verrà prima invocata la notifica del cambiamento dello stato e poi l'eventuale passaggio del token.

All'interno della classe Node sono state implementate tutte le funzioni inerenti alla comunicazione del cambiamento dello stato. Questa classe è stata strutturata in maniera molto semplice e possiamo sostanzialmente distinguere tre operazioni principali:

- Inizializzazione del nodo: Questo task viene svolto dal metodo initializeNode() e si tratta dell'inizializzazione del nodo con i dati relativi alle informazioni del giocatore e della successiva attivazione dei metodi RMI della classe. Nel caso in cui il nodo in questione debba anche fungere da servizio di registrazione, tale servizio viene attivato in questo metodo.
- Registrazione al gioco: Questo task viene svolto dal metodo registerToGame() e consiste in una interazione con il servizio di registrazione al fine di ottenere la lista ordinata dei giocatori.
- Comunicazione: Questo task viene svolto principalmente dal metodo send(), il quale a seconda della tipologia di parametro, effettua la relativa gestione della comunicazione. Al fine di rendere più corretta la struttura delle direttive di comunicazione, sono stati previsti 3 metodi che nei fatti sfruttano send(): startGame(), usata per condividere con tutti gli altri nodi lo stato iniziale, sendMove(), usata per comunicare la mossa effettuata e sendToken(), usata per passare il token al nodo successivo e quindi rendere operativo il cambio di turno.

Quanto descritto fin'ora non è sufficiente a completare il progetto poichè le specifiche impongono la gestione dei guasti di tipo crash. Di ciò si occupa la classe CrashManager. Questa classe è stata pensata per potersi accorgere di un guasto in due differenti contesti operativi. Quello più triviale è la rilevazione di un crash in seguito alla cattura di una RemoteException in tutti i metodi della classe Node deputati alla comunicazione: verrà dunque catturato un errore di comunicazione con il nodo succcessivo. In tali metodi è stata sempre utilizzata le seguente struttura:

```
public void direttivaComunicazione(){
        try {
                InvocazioneDirettivaRMI();
        } catch (RemoteException e) {
                MyCrashManager.repairAndBroadcastPlayerList();
                this.direttivaComunicazione();
        }
}
```

Ossia ogni problema di comunicazione viene intercettato e gestito da parte di un oggetto MyCrashManager di tipo CrashManager attraverso l'utilizzo del metodo repairAndBroadcastPlayerList(). Tale metodo infatti effettua due semplici operazioni:

- Rimuove dalla PlayerList l'end-point andato in crash (ossia il suo successore in PlayerList).
- Comunica l'ID del giocatore rimosso a tutti gli altri nodi attivi.

In questo modo abbiamo la garanzia che tutti i nodi condivideranno lo stesso nuovo anello di comunicazione, poiché ogni nodo alla ricezione di tale identificatore provvederà alla rimozione dell'annesso nodo dalla PlayerList. Al termine di tale operazione la direttiva di comunicazione desiderata viene reinvocata ricorsivamente in modo tale da poter garantire la corretta operabilità del sistema anche nel caso in cui il nuovo successore vada in crash.

Tuttavia tale strategia non è sufficiente a garantire una gestione completa dei crash, poiché non consente in nessun modo di poter gestire un caso reale molto semplice: il regolamento del gioco non dà nessun limite temporale al giocatore che detiene il turno. Questa situazione dal punto di vista implementativo significa che (usando solo la soluzione descritta in precedenza) è impossibile stabilire se il nodo che detiene il token è vivo pur non effettuando comunicazioni o se è andato in crash. Per gestire tale situazione è stato implementato il metodo ping(), la cui esecuzione a intervalli prestabiliti è data dall'uso dell'oggetto Timer all'interno di startTimedController(). Una mancata risposta al ping() implicherà una gestione dei guasti effettuata con lo stesso modus operandi descritto in precedenza. Al fine di poter rendere completa la gestione dei crash, l'uso del ping è stato esteso a tutti i nodi dell'anello, indipendentemente dal fatto che questi detengano il token.

Dal momento che i crash sono impredicibili, è assolutamente plausibile che un nodo possa andare in crash durante un operazione del CrashManager: per questo motivo anche le direttive di comunicazione inter-nodo della classe CrashManager utilizzano lo schema di codice descritto per i metodi di comunicazione della classe Nodo, ossia viene aggiornata la PlayerList, viene comunicato l'aggiornamento da effettuare, e infine la direttiva viene reinvocata ricorsivamente.

Un ultima doverosa spiegazione del meccanismo di gestione dei crash, riguarda l'assenza di assunzioni su intervalli di time-out: tale mancanza non è dovuta a dimenticanze, bensì alla precisa scelta di voler sfruttare il fatto che RMI fornisce nativamente tali feature, ivi compresa la configurazione manuale di tale intervallo.

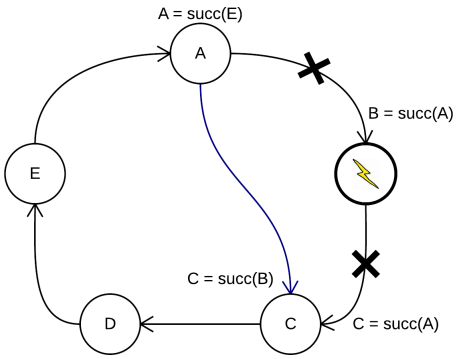

Figure 4: Diagramma riassuntivo del funzionamento della classe CrashManager

5. Diagrammi

In questa sezione verranno illustrati i seguenti diagrammi UML:

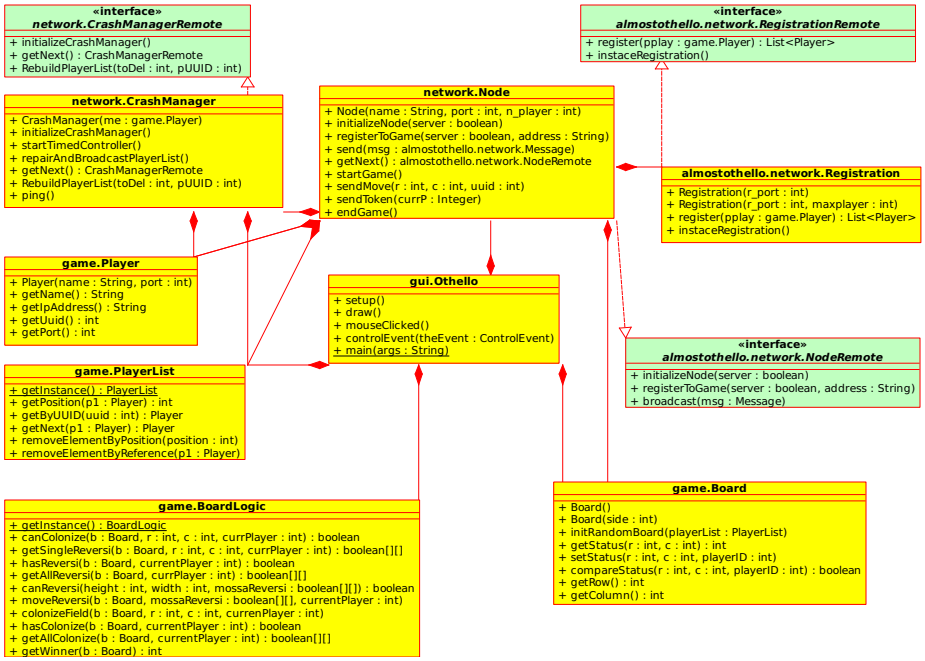

Figure 5: Diagramma delle classi

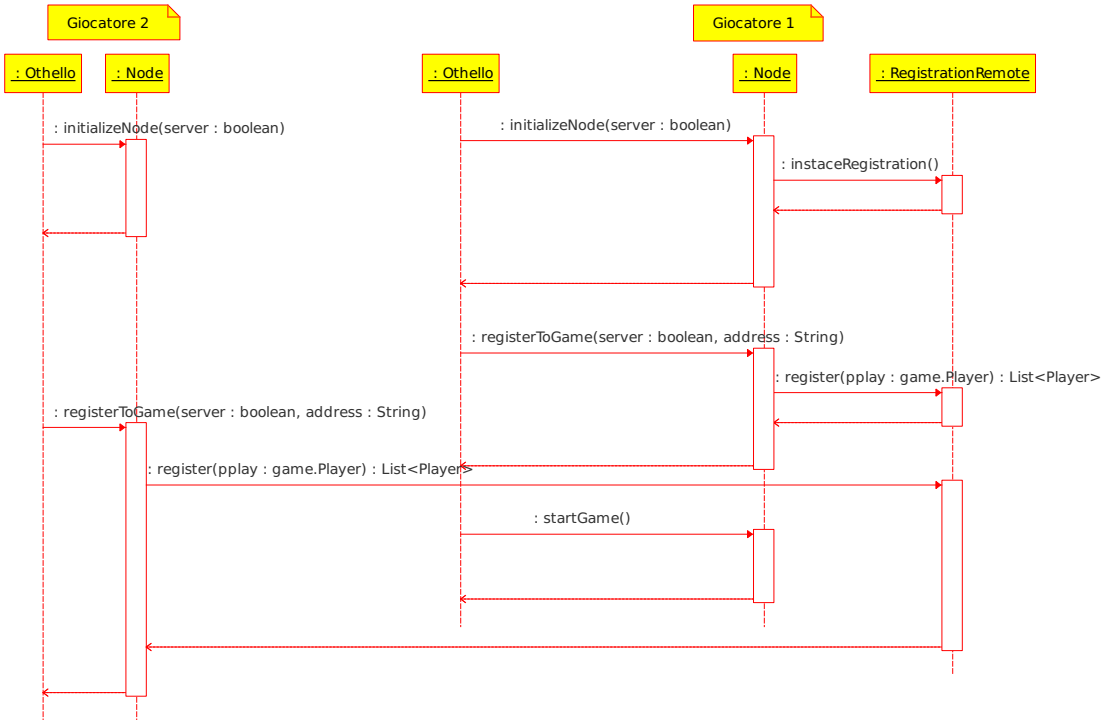

Figure 6: Diagramma di sequenza della fase di registrazione al gioco

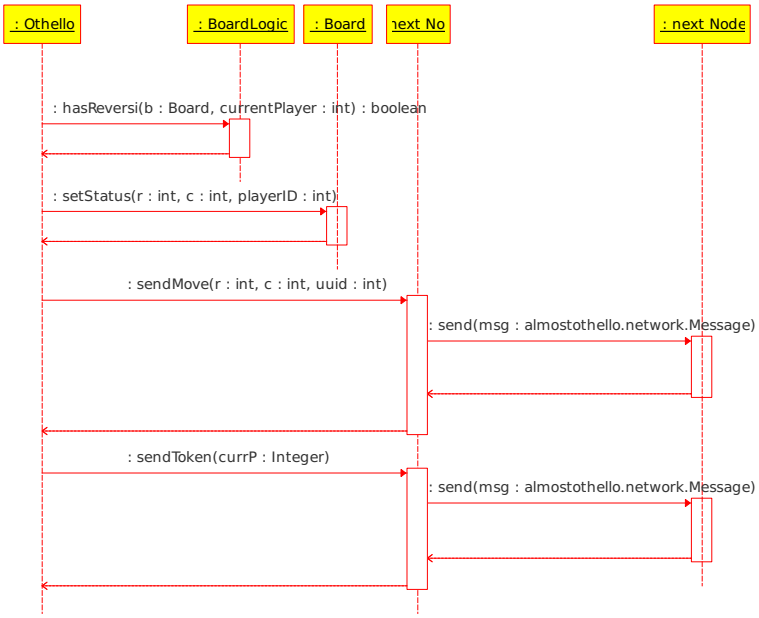

Figure 7: Diagramma di sequenza relativo a una mossa di gioco di tipo reversi

6. Conclusioni

Come espresso al capitolo 1 lo scopo era realizzare un gioco con almeno 3 giocatori, con uno stato globale condiviso e che tollerasse $N-1$ guasti di tipo crash, attraverso l'utilizzo di RMI come paradigma di comunicazione. Si può affermare che il sistema realizzato rispetta le specifiche imposte per la realizzazione dell'applicazione. L'utilizzo di RMI ha facilitato la realizzazione di un modello di messaging molto semplice ma assolutamente efficace ai fini dei requisiti e alle regole del gioco. La flessibilità di RMI è stata un fattore determinante anche per quanto concerne la realizzazione del sistema di recovery dai crash. In questo contesto è stata riscontrata la difficoltà progettuale maggiore, poichè si è resa necessaria l'adozione di un oggetto di tipo Timer per realizzare un ping temporizzato che potesse essere eseguito in maniera indipendente. E' opportuno sottolineare che la difficoltà riscontrata è essenzialmente imputabile alle scarse esperienze precedenti in termini di programmazione multithread in linguaggio Java.

I possibili sviluppi futuri dell'applicazione riguardano l'adozione della possibilità di poter effettuare nuove partite senza dover chiudere ogni volta l'applicazione. Inoltre sarebbe molto interessate implementare un sistema di locate automatico del server di registrazione. Dal punto di vista dell'interfaccia grafica si potrebbe dare la possibilità ai giocatori di poter scegliere il proprio colore o addirittura il proprio avatar.

Tuttavia questi aspetti sono a parere degli autori secondari rispetto al design

architetturale, che soddisfa invece le speciche iniziali.