Tron

Progetto di Sistemi Distribuiti

Antonio Carbonara, Andrea Sghedoni, Orgest Shehaj

antonio.carbonara@studio.unibo.it
andrea.sghedoni4@studio.unibo.it
orgest.shehaj@studio.unibo.it

Università di Bologna

11 aprile 2016

Indice

1 Aspetti Progettuali **3**
- 1.1 Server . . . 3
- 1.2 I Peer . . . 4
- 1.3 Il Gioco . . . 5
- 1.4 I problemi affrontati . . . 6
- 1.5 Le soluzioni proposte . . . 7

2 Aspetti Implementativi **8**
- 2.1 Package network . . . 10
- 2.2 Package graphics . . . 11
- 2.3 Package crash . . . 12
- 2.4 Package registration . . . 13

3 Valutazioni e Conclusioni **14**

Sommario

Il progetto consiste nell'implementazione di una versione distribuita del gioco Tron, con particolare attenzione alla tolleranza ai guasti di tipo crash sui nodi distribuiti, che sono stati gestiti in modo appropriato per non compromettere la partita. Abbiamo apportato modifiche importanti rispetto la versione originale del gioco, per poterla adattare al nostro caso, come il numero massimo dei giocatori, le posizioni di partenza delle moto ed altro ancora. In questa relazione descriveremo l'architettura del progetto ed i problemi che abbiamo affrontato.

Introduzione

Come progetto per il corso di Sistemi Distribuiti abbiamo implementato il gioco *Tron*, Ovvero un videogioco arcade pubblicato da Bally Midway nel 1982, ispirato all'omonimo film. Il gioco consiste nell'evitare di scontrarsi contro le scie delle moto degli avversari, cercando di intrappolare gli avversari nella propria scia. Al gioco possono partecipare un massimo di 6 giocatori, a differenza del gioco originale, dove il massimo numero dei giocatori è limitato a 4. Essi si distinguono tra di loro sia per colori differenti di scie, sia per differenti posizioni che sono equidistanti, oltre al nome (username). Il goal che ci siamo imposti è consistito nel creare un gioco (con grafica minimale) che sia utilizzabile su più macchine interconnesse tra di loro e che debba evitare la totale interruzione del gioco nel caso uno dei giocatori coinvolti:

- perda
- chiuda la connessione
- la sua macchina vada in crash

1 Aspetti Progettuali

1.1 Server

Per cominciare il gioco, deve essere avviato per primo il server, il quale interroga l'utente su tutti i dettagli necessari per implementare la *Room* dove avverrà la competizione, i dettagli richiesti dal server sono:

- l'username dell'utente stesso che si comporta come amministratore
- il numero dei giocatori che parteciperanno alla gara

Figura 1: Schermata iniziale del Server

- viene inoltre mostrato l'indirizzo ip da comunicare agli altri giocatori che si dovranno connettere al server.

Una volta completati tutti i campi,il server, aspetta che tutti gli altri giocatori siano connessi, dopo di che il suo ruolo di server terminerà e la comunicazione sarà soltanto peer-to-peer; da qui in poi, il gioco continuerà soltanto tra i peer, e non vi sarà più la parte centralizzata.

1.2 I Peer

Quando comincia il gioco, se non si sceglie di fare il server, allora il giocatore si può comportare solamente come un peer, e i dati richiesti ad un peer sono:

- il suo username
- l'indirizzo ip del server

Una volta completati questi campi, si fa aspettare il giocatore finchè non si sono connessi anche tutti gli altri peer, dopo di che comincerà il gioco

vero e proprio. La posizione di partenza di ogni peer, viene decisa a seconda dell'ordine in cui il peer stesso si nnesso al server, e in questo modo anche il colore della moto di ogni giocatore; si parte dal giocatore che ha svolto il ruolo di server iniziale, al quale viene associato il colore rosso e la posizione in basso a sinistra, dopo di che viene assegnato una posizione e un colore anche ai giocatori rimanenti.

Figura 2: Schermata iniziale del Peer

1.3 Il Gioco

Dopo la registrazione di ogni peer, viene allestita la Room (ovvero l'arena di gioco), posizionati i giocatori, ed il gioco comincia. Ogni giocatore parte a una distanza prestabilita dagli altri, avendo in questo modo il tempo per giocare al meglio la propria partita e usare la propria strategia in un tempo congruo con quello degli altri giocatori. Le posizioni di partenza sono diverse rispetto al gioco ufficiale dal quale abbiamo preso spunto, e anche il numero dei giocatori, in quanto prevediamo un numero massimo di 6 giocatori, mentre nel gioco ufficiale il numero di giocatori è limitato a 4; le posizioni iniziali dei giocatori sono differenti, in quanto, nel gioco ufficiale

tutti e 4 i giocatori partono dal centro e sono diretti verso gli angoli, mentre nella nostra versione i giocatori cominciano a giocare dagli angoli, poichè il numero massimo dei giocatori non permette di farli partire dal centro della Room, in quanto si scontrerebbero subito. Ogni giocatore "viaggia" con la sua moto a velocitstante, uguale per tutti i nodi. I giocatori perdono se:

- oltrepassano i limiti dell'arena contrassegnati da dei bordi che hanno colore uguale a quello della scia del giocatore stesso;
- sbattono contro la scia di un altro giocatore;
- sbattono contro la propria scia.

1.4 I problemi affrontati

I problemi incontrati in seguito ad alcune prove effettuate nel laboratorio Ercolani sono stati i seguenti:

- **Crash:** con quattro o più giocatori, nel caso uno di questi andava in crash il gioco non poteva proseguire in quanto l'anello non veniva riconfigurato e quindi ogni giocatore non aggiornava le informazioni dei nodi dell'anello e dunque il gioco veniva interrotto poichè non si trasmettevano le informazioni necessarie per poter proseguire.
- **Aggiornamento colore:** nel caso in cui due o più giocatori giocavano in maniera ipoteticamente perfetta, il colore di alcuni punti delle scie veniva visualizzato in modo errato in quanto le informazioni relative al posizionamento di ogni giocatore crescevano in maniera esponenziale e probabilmente il garbage collector di Java non aggiornava adeguatamente i colori dei punti delle scie a causa del tempo di aggiornamento che risultava essere troppo piccolo (tra i 20 ed i 50ms).

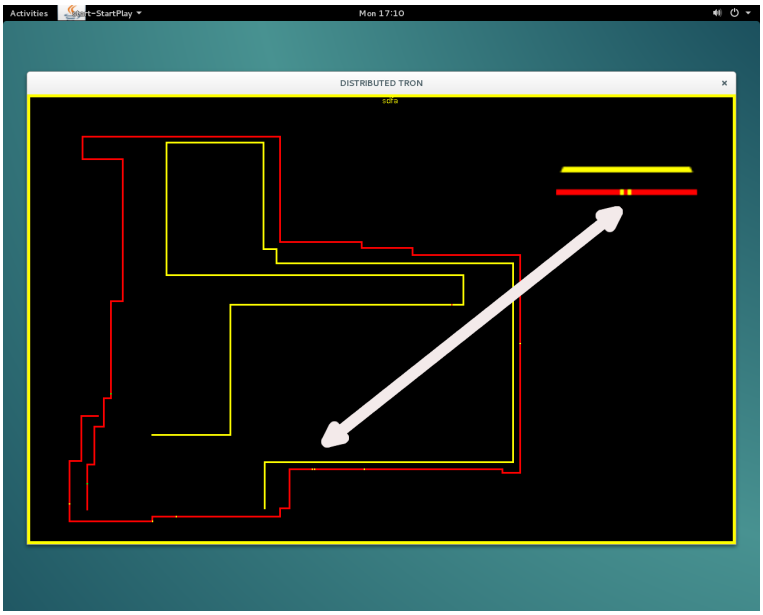

Figura 3: Schermata con l'errore

- **Aggiornamento scie:** Nel caso in cui uno dei giocatori, che partiva a ovest o ad est, perdeva, la sua scia non veniva cancellata completamente poichè l'algoritmo non cancellava adeguatamente i punti in quanto la posizione iniziale del giocatore stesso si trovava in una posizione non divisibile per la grandezza della moto, e veniva arrotondato per eccesso, mentre nella matrice si memorizzavano i punti arrotondati per difetto.

- **Scontro tra i giocatori:** nel caso in cui due giocatori erano troppo vicini tra di loro e si scontravano, le loro scie si attraversavano in quanto le informazioni delle posizioni dell'avversario non venivano aggiornate in tempo, ovvero le posizioni dell'avversario non facevano il giro dell'anello in tempo.

1.5 Le soluzioni proposte

- **Crash:** abbiamo implementato un algoritmo più intelligente consistente nell'inviare una nuova configurazione dell'anello al nodo successivo (che non sarà in crash) una volta scoperto il nodo che è andato in crash grazie all'eccezione RemoteException generata dalla chiusura

dell'interfaccia rmi del nodo che è andato in crash. Quest'ultimo viene eliminato dall'anello.

- **Aggiornamento colore:** per risolvere questo specifico problema abbiamo dovuto cambiare il modo di memorizzazione delle coordinate delle scie di ogni moto, ovvero abbiamo implementato una matrice statica con coordinate prefissate, in cui ogni cella corrisponde alla dimensione della moto, e la dimensione totale della matrice deve essere divisibile per la dimensione della moto per creare delle celle uniformi.
- **Aggiornamento scie:** questo problema è stato banalmente risolto cambiando le coordinate dei giocatori che cominciavano il gioco ad est od ovest
- **Scontro tra i giocatori:** anche questo problema è stato di semplice soluzione in quanto bastava aumentare il tempo di aggiornamento dell'invio dei messaggi.

2 Aspetti Implementativi

Per l'implementazione del progetto si elto di utilizzare il linguaggio di programmazione basato sugli oggetti Java, che tramite il framework RMI fornisce API specificamente progettate per consentire la realizzazione di applicazioni distribuite in modo efficace. Il software ato implementato seguendo il pattern **Singleton**, uno dei pattern fondamentali descritti dalla "Gang of four", ovvero un pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza. Per implementare il gioco distribuito di Tron abbiamo scelto una struttura di comunicazione ad anello unidirezionale, dove vi è una prima parte centralizzata in cui viene eseguito per primo il server di registrazione. Il server di registrazione ha lo scopo di fornire un unico punto di accesso, in maniera da consentire al server stesso di conoscere tutti i dettagli necessari per organizzare la Room dove avverra la competizione. Una volta completato il suo compito, il server, torna ad essere allo stesso livello degli altri nodi, ovvero soltanto un peer. Ogni nodo, conosce le coordinate della scia della propria moto, e anche le coordinate di ogni moto ancora in competizione, ogni volta che un nodo esplora delle coordinate nuove, invia queste ultime al suo vicino, e le informazioni fanno il giro completo dell'anello fino a ritornare al mandante stesso, aggiornando cosi le informazioni di ogni nodo. Qui di seguito descriveremo le singole

classi implementate (solamente quelle più importanti), suddivise nei seguenti pacchetti: **network** (classi che gestiscono la comunicazione di rete tramite le librerie java.rmi) , **crash** (vi è un'unica classe che si occupa della gestione dei crash), **graphics** (classi che realizzano e gestiscono la resa grafica del gioco grazie alle librerie java.awt e/o java.swing).

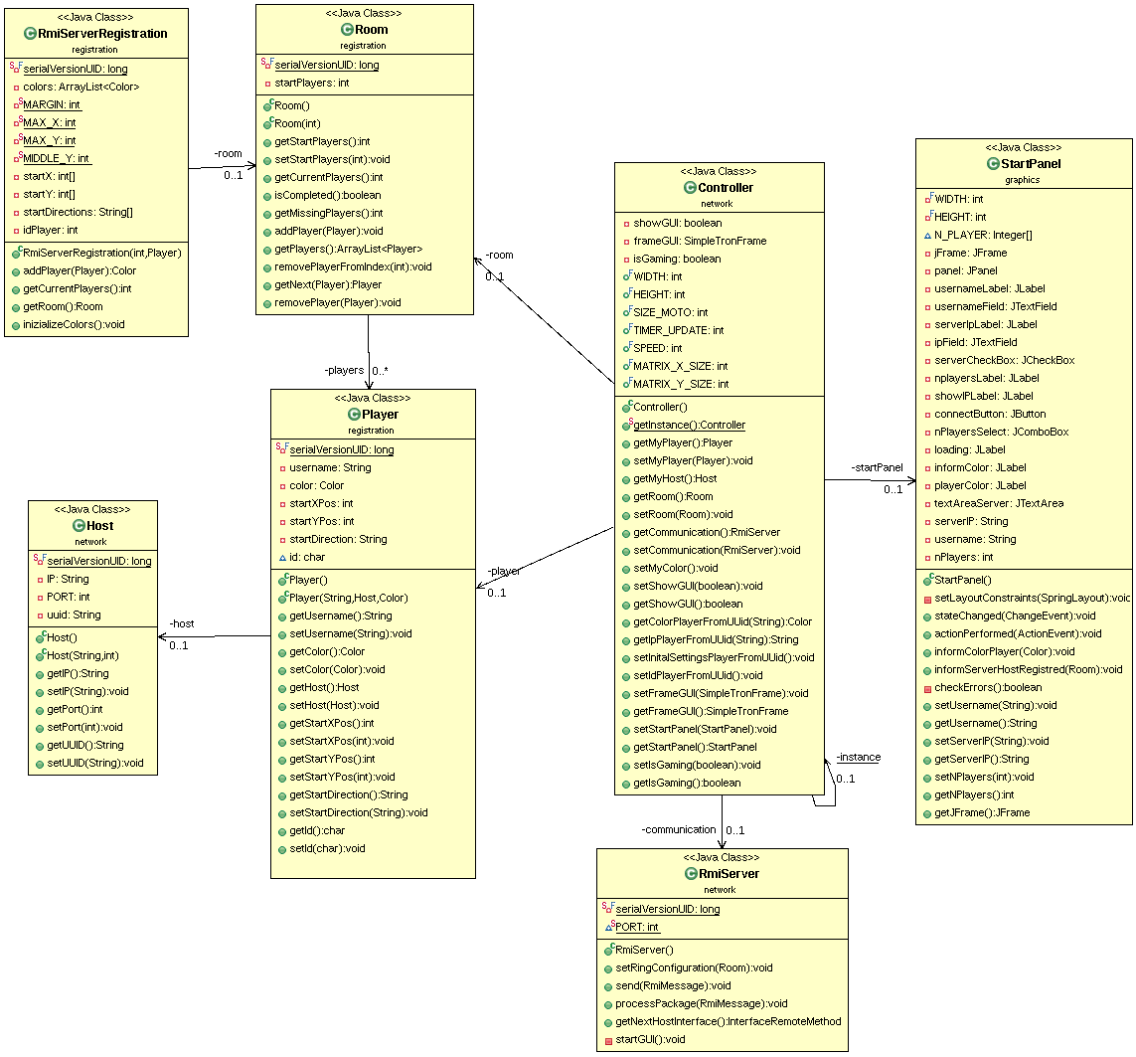

Figura 4: Diagramma delle classi

2.1 Package network

Figura 5: Diagramma delle classi del Network

- `Controller`: implementato col pattern **Singleton**, si occupa di gestire tutti quelli aspetti legati alla configurazione della rete (gestione dell'anello) e di tenere traccia degli elementi più importanti e che potrebbero essere richiamati successivamente nel codice come ad es. la Room, i dati del giocatore o host, ...

- `RmiServer`: si occupa di implementare la primitiva `send()` dell'interfaccia `InterfaceRemotMethod` in maniera tale che faccia il giro completo dell'anello, ovvero ciascun `Host` verifica che l'`Uuid` sia uguale al proprio altrimenti passa il messaggio al successivo `Host` dell'anello. Nel caso l'`Uuid` sia uguale, processa il messaggio e verifica il tipo del messaggio: se di tipo `Rectangle` aggiorna l'interfaccia grafica del gioco tramite una `repaint()`; se di tipo `Room` vuol dire che un `Player` è

andato in crash ed allora bisogna settare una nuova configurazione dell'anello.

2.2 Package graphics

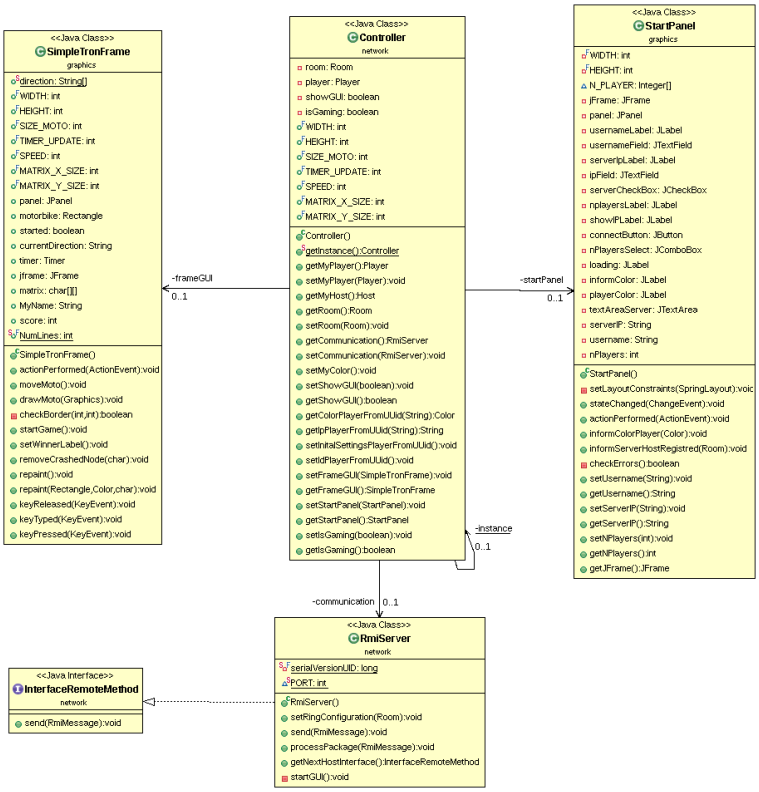

Figura 6: Diagramma delle classi del Graphics

- `StartPanel`: Si occupa di impostare il posizionamento degli elementi grafici nel campo da gioco, come le scie dei giocatori e i bordi del campo .
- `SimpleTronFrame`: si occupa di disegnare le scie delle varie moto ogni volta che riceve le coordinate, gestisce anche la direzione della moto del giocatore corrente ed elimina la scia del giocatore che ha perso o è andato in crash.

2.3 Package crash

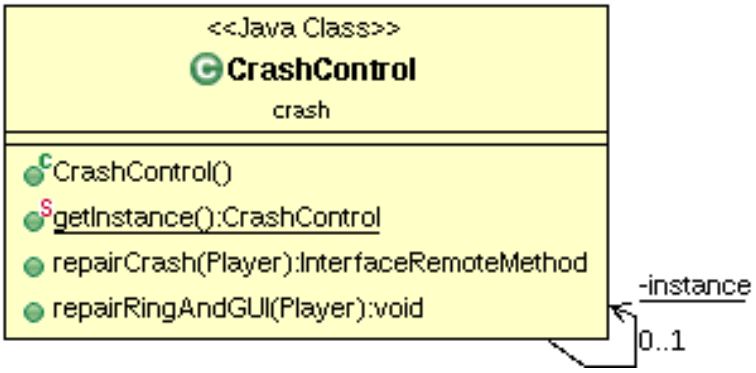

Figura 7: Diagramma delle classi del Crash

- `CrashControl`: crea l'istanza **Singleton**, prende in input il nodo crashato e si occupa di rimuovere il `Player` dalla `Room` e riconfigurare l'anello nel modo appropriato.

2.4 Package registration

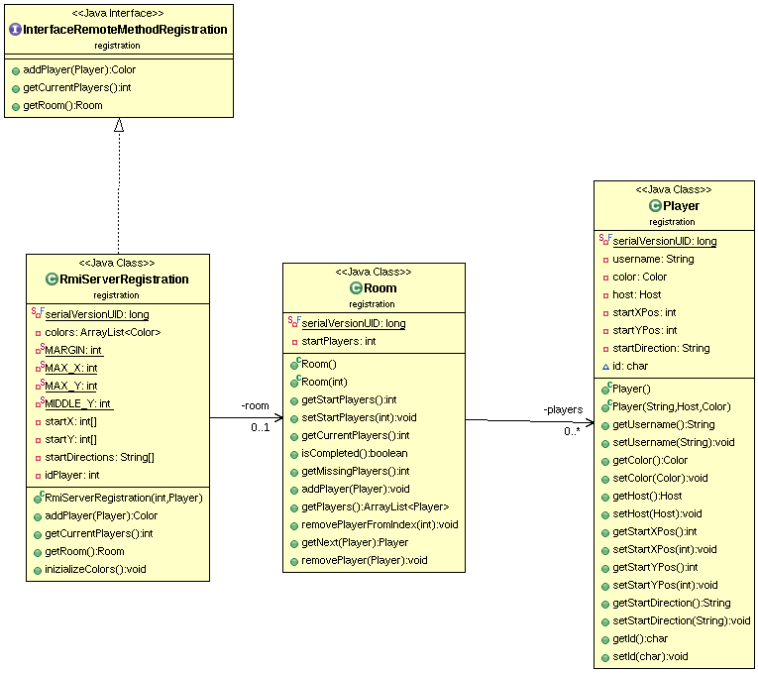

Figura 8: Diagramma delle classi della Registrazione

- `Room`: Classe che tiene traccia dei giocatori che partecipano al gioco, e di tutti i loro dettagli.
- `RmiServerRegistration`: classe che si occupa di implementare un servizio base di registrazione centralizzato remoto. Si occupa di aggiungere o rimuovere i host che partecipano al gioco.
- `Player`: La classe `Player` rappresenta il giocatore; tiene traccia di tutti i suoi dettagli, come l'username, il colore della moto, le posizioni di partenza e la direzione.

3 Valutazioni e Conclusioni

Dopo aver fatto tutti i test appropriati siamo giunti alla conclusione che l'applicazione da noi creata sia in grado di gestire guasti di tipo crash su uno qualsiasi dei nodi durante il gioco. Anche se il body dei messaggi scambiati tra i vari peer è molto grande e il tempo di aggiornamento delle posizioni di ogni peer avviene ogni 50 millisecondi, ciò non crea alcun tipo di rallentamento, ma consente un gioco fluido e consistente. Per aumentare la giocabilità di Tron si potrebbe migliorare la grafica e aggiungere una classifica generale. Tale risultato è stato raggiunto cambiando la logica iniziale del progetto, ovvero in un primo momento si era deciso di memorizzare per ogni `Player` anche le proprie coordinate di gioco in un `ArrayList` e quindi ogni giocatore, ogni volta che si muoveva, doveva comunicare le proprie coordinate, cosa che causava rallentamenti nel gioco o problemi di colorazione sbagliata delle scie. Il tutto si è risolto realizzando l'arena di gioco (o *Room*) tramite una matrice di coordinate prefissate in maniera tale che ogni giocatore poteva accorgersi della sconfitta di un altro semplicemente scandendo tale matrice e non tutte le liste di tutti i giocatori che causavano problemi con l'aggiornamento delle coordinate.