

# Distributed Uno

Albana Gaba      Fabio Strozzi

## Sommario

Abbiamo realizzato la versione distribuita del gioco di carte Uno che abbiamo chiamato DistriUno. Lo scopo principale del progetto è stato quello di studiare i meccanismi di fault tolerance nei sistemi distribuiti. Nel caso specifico di DistriUno abbiamo fatto in modo che il gioco fosse tollerante al guasto di al più  $n - 2$  nodi. Questo articolo descrive l'architettura del sistema, gli aspetti progettuali della politica di fault tolerance e alcuni dettagli implementativi.

## 1 Introduzione

In DistriUno i giocatori sono tutti pari tra di loro, ovvero non esiste nessuna forma di centralizzazione fatta eccezione per la fase iniziale del gioco (chiamata *fase di bootstrap*) che comprende:

- registrazione dei giocatori
- distribuzione delle carte e dei mazzi
- comunicazione dei partecipanti a tutti i giocatori

Questa fase è gestita da un server dedicato che viene eseguito appositamente per avviare una partita.

Il gioco è a turni. La sequenza delle giocate non è definita staticamente ma può cambiare (e in particolare invertirsi) durante la partita in base alle carte giocate. Il numero massimo di giocatori è 10 e una partita può proseguire fintanto che rimangono almeno due partecipanti. Le regole del gioco possono essere consultate su Wikipedia [Wik].

Secondo le specifiche del progetto [Amo], si ipotizza che la rete sia affidabile (niente guasti di tipo *data omission*) e che gli unici guasti che si possono verificare siano del tipo *crash dei nodi*. Da un lato questo significa che, a meno di guasti dei peer, un messaggio inviato viene sicuramente ricevuto dal destinatario. Ciò non esclude tuttavia la possibilità che la rete abbia una elevata latenza, scarsa banda o

addirittura che sia congestionata. Il nostro progetto ha cercato di affrontare anche questo tipo di problematica (si veda il capitolo 3.1).

Durante lo sviluppo del progetto abbiamo cercato di mantenere l'architettura del sistema e il protocollo di comunicazione più semplici possibile. Il progetto, perciò offre una soluzione ad-hoc, strettamente dipendente dal gioco e dalle specifiche richieste. Ciò nonostante, laddove è stato possibile, abbiamo cercato di trovare soluzioni generali in modo da creare componenti riutilizzabili per altri problemi distribuiti (si vedano i capitoli 2.1 e 4).

## 2 Il modello

Il sistema creato può essere caratterizzato dal modello di *sistema distribuito asincrono*, in base al quale [CDK02] non sono definiti limiti:

- alla velocità di esecuzione dei processi;
- ai ritardi nella trasmissione dei messaggi;
- al *drift rate* degli orologi fisici.

In pratica, non si può affermare che un processo che non risponde entro un margine di tempo fissato ha certamente avuto un guasto. Ma facendo affidamento sulla affidabilità della rete (vedi le specifiche [Amo]) possiamo scoprire lo stato di un nodo spedendo con cadenza regolare un messaggio specifico (chiamato AYA, “are you alive”): per convenzione, si è deciso che se avviene un errore in risposta al messaggio AYA, si può concludere che il nodo ha fatto crash<sup>1</sup>. In generale, ogni volta che avviene un errore durante la consegna di un qualunque tipo di messaggio (con Java RMI questo si traduce in una `RemoteException`), il peer destinatario viene definito guasto ed escluso dal gioco.

Tra gli obiettivi del sistema vi è anche quello di informare i processi sull'avanzamento del gioco prima possibile. Per propagare i messaggi di gioco abbiamo implementato un protocollo di *broadcast*. I messaggi scambiati sono di due tipi:

**messaggi di gioco** : rappresentano le “mosse” che ha fatto un giocatore durante il suo turno (quante carte ha pescato, quale carta ha scartato se ha scelto di punire qualcuno, ecc.).

**messaggi di controllo** : servono per garantire le connessioni tra i peer e mantenere lo stato coerente; si distinguono in AYA e messaggi errore. Verranno descritti meglio nel capitolo 3.

---

<sup>1</sup>Da ora in poi distingueremo tra peer attivi (o vivi) e peer crashed: i primi sono quelli che fanno ancora parte del gioco, i secondi sono quelli ritenuti guasti e quindi esclusi dal gioco.

## 2.1 Topologia

Abbiamo deciso di adottare per la topologia della rete l'*anello bidirezionale*. Secondo questa configurazione, un nodo che desidera spedire in broadcast un suo messaggio deve comunicarlo ad entrambi i suoi vicini (che chiameremo *vicino destro* e *vicino sinistro*). Questi dovranno a loro volta inoltrare il messaggio ricevuto al vicino opposto; il vicino destro lo inoltrerà al proprio vicino destro mentre il vicino sinistro lo inoltrerà al proprio vicino sinistro. L'operazione si ripete fino a che il messaggio non è arrivato a tutti i nodi. Quando un nodo (per esempio quello diametralmente opposto al mittente) riceve il messaggio due volte (sia dal vicino destro che da quello sinistro), scarta la seconda copia senza inoltrarla.

Rispetto all'anello unidirezionale, nella migliore delle ipotesi (la rete dei nodi ha banda e latenza pressoché uniformi e non si verificano crash) un messaggio per arrivare a tutti i peer impiega metà tempo.

Considerato che il verso di gioco può cambiare durante la partita a seconda delle carte giocate, questa scelta risulta più adatta dell'anello unidirezionale con verso fissato staticamente. Infatti, è preferibile che chi deve prendere il turno venga notificato prima possibile.

Questa topologia si presta bene anche per sistemi distribuiti con un elevato numero di nodi (ad esempio dell'ordine delle migliaia): qualunque sia la dimensione della rete ogni nodo deve mantenere le comunicazioni solamente con altri due nodi.

## 2.2 Java RMI

Il nodi della rete comunicano tra di loro tramite un semplice protocollo realizzato grazie al paradigma delle "remote method invocations" di Java RMI. La scelta di RMI ha influenzato notevolmente non solo le scelte implementative, ma anche quelle progettuali, per cui vale la pena descrivere brevemente quali sono stati gli aspetti cruciali.

Le chiamate remote di RMI sono bloccanti, ovvero il chiamante rimane in attesa che si concluda l'esecuzione remota prima di poter riprendere il controllo. Inoltre, ogni chiamata remota comporta per il chiamato - l'host remoto - l'avvio di un nuovo thread dedicato all'esecuzione del metodo.

La prima conseguenza è che più invocazioni bloccanti annidate possono provocare una attesa molto lunga da parte del chiamante, che è obbligato ad aspettare che l'ultimo peer coinvolto termini l'esecuzione del metodo. A peggiorare la situazione possono contribuire la latenza e il congestionamento della rete. Ciò è ancor più svantaggioso se si ripercuote sull'interfaccia grafica. La soluzione che abbiamo adottato

è stata quella di creare un thread dedicato alla spedizione dei nuovi messaggi (si vedano i capitoli 3 e 4 per una descrizione più dettagliata).

La seconda conseguenza è che bisogna pensare l'esecuzione dei metodi remoti nell'ottica della programmazione concorrente: infatti, se più metodi vengono invocati su un peer, questo dovrà estire localmente la mutua esclusione per l'accesso alle proprie risorse dal momento che i metodi remoti possono essere eseguiti contemporaneamente. Per semplificare l'architettura abbiamo fatto in modo che i messaggi in arrivo (i parametri delle invocazioni remote) confluissero in un'unico buffer. Di volta in volta il gioco estrae i messaggi dal buffer e li processa.

## 2.3 Coerenza e stati globali

Lo stato del gioco è definito da:

- le carte del mazzo coperto;
- l'ultima carta giocata scoperta sul tavolo;
- il giocatore corrente;
- il numero di carte in mano ad ogni giocatore (fondamentale per determinare la chiusura del gioco).

Data la configurazione iniziale (in seguito alla fase di *bootstrap*), la logica di DistriUno è tale che ogni nodo può aggiornare autonomamente il proprio stato in base a:

- le proprie giocate;
- le giocate degli avversari;
- la conoscenza locale che si ha degli altri peer (quali sono attivi e quali crashed), si veda il capitolo 3.

Se ogni giocatore comunica con un messaggio di gioco le proprie scelte a tutti gli altri, in ogni istante ognuno è in grado di stabilire autonomamente a chi appartiene il turno di gioco. Di conseguenza non c'è bisogno di scambiare un *token* per accedere alla sezione critica (in questo caso lo stato globale, modificabile dai messaggi di gioco).

### Creazione e ricezione dei messaggi

Affinché gli stati locali siano coerenti è cruciale il modo con cui vengono creati e processati i messaggi<sup>2</sup>. A tal fine si è stabilito che:

---

<sup>2</sup>Con “processare un messaggio” intendiamo l'operazione che aggiorna lo stato locale del gioco in base al contenuto del messaggio.

1. ogni peer invia un unico messaggio di gioco e solo durante il suo turno (fanno eccezione i messaggi di controllo, si veda il capitolo 3);
2. i messaggi di gioco vengono processati dai peer nello stesso ordine e con la stessa logica;
3. i messaggi di gioco, una volta spediti, o vengono processati da tutti i peer attivi, oppure non vengono processati da nessuno<sup>3</sup>.

Fatte queste premesse, si è certi che lo stato globale del gioco in qualunque istante coincide con lo stato del peer che ha lo stato locale più avanzato (condizione utile in casi particolari di crash e verrà descritta meglio in 3).

Per garantire la seconda condizione si deve marcare ogni messaggio con un numero sequenziale che svolge il ruolo di identificatore univo. Il peer col turno incrementa questo contatore di una unità e marca il proprio messaggio di gioco col nuovo valore: una sequenza di messaggi validi successivi, per esempio, può essere identificata da 8, 9, 10 e 11 ma non da 8, 10, 11, né da 8, 9, 9, 10, 11. In ricezione può succedere che i messaggi arrivino disordinati, ad esempio 8, 11, 10, 9. Ogni peer processa solo sequenze strettamente ordinate e crescenti: quindi, prima si ordinano i messaggi arrivati e poi si processano solo quelli per cui la sequenza è completa (cioè senza lasciare “buchi”). Nell’esempio presente, il peer avrebbe processato 8 al suo arrivo ma poi avrebbe messo 11 e 10 in una “pila ordinata” in modo da processarli solo dopo aver ricevuto e processato 9 (l’unico messaggio che mi aspetto di ricevere dopo 8).

## 3 Fault tolerance

### 3.1 Scenari possibili

## 4 Architettura

L’architettura è strutturata su tre livelli. Alla base vi è un livello dedicato alla rete ed è responsabile dei collegamenti tra i nodi, dello scambio di messaggi e attua le politiche di tolleranza ai guasti. Il livello intermedio implementa le regole del gioco: insieme al livello di rete contribuisce a mantenere coerente lo stato del gioco. Al più alto livello si trova l’interfaccia grafica.

---

<sup>3</sup>L’alternativa era implementare una operazione di *rollback* capace di ripristinare uno stato precedente del gioco, ma questo avrebbe complicato notevolmente l’architettura del sistema.

Per quanto riguarda il livello di rete, abbiamo cercato di proporre una soluzione generale. Essa, infatti, può essere riutilizzata per qualunque altro gioco distribuito a turni i cui nodi sono organizzati su una rete ad anello bidirezionale. L'unico vincolo è che ad ogni turno venga spedito un singolo messaggio di gioco.

## 5 Conclusioni e sviluppi futuri

### Riferimenti bibliografici

- [Amo] Amoroso Alessandro. Specifiche del progetto di Sistemi Distribuiti. <http://www.cs.unibo.it/~amoroso/sistdisdoc.html>.
- [CDK02] Coulouris George, Dollimore Jean, and Kindberg Tim. *Distributed Systems, concepts and design*. Addison-Wesley, third edition, 2002.
- [Wik] Wikipedia. Uno game rules. [http://en.wikipedia.org/wiki/UNO\\_%28game%29](http://en.wikipedia.org/wiki/UNO_%28game%29).