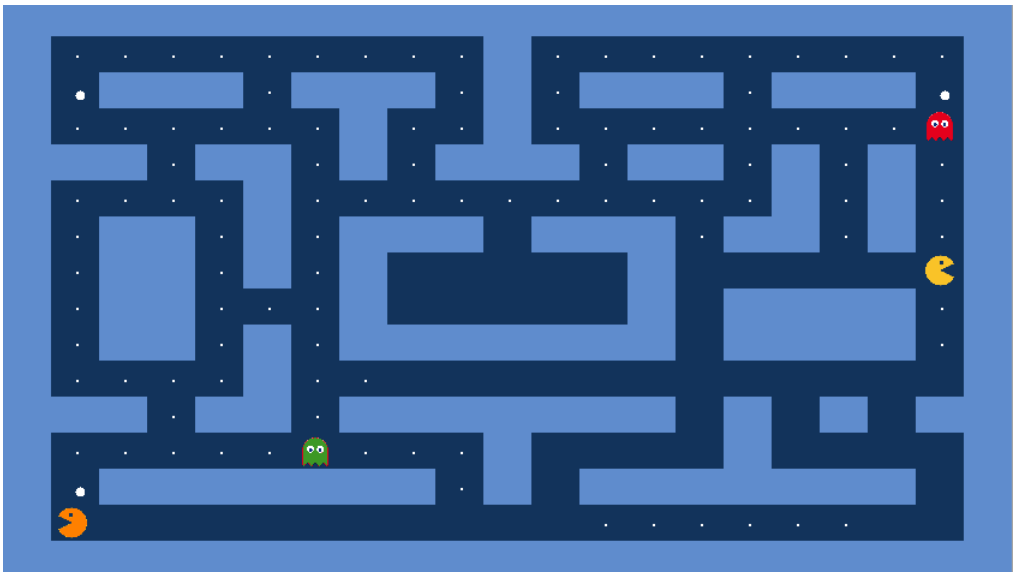
Progetto Sistemi Distribuiti
a.a. 2014/2015
Pac Man Multiplayer

Antonello Antonacci
Pierpaolo Del Coco
Ivan Heibi

Alma Mater,Università di Bologna



Sommario Con questo progetto si è realizzato un gioco su piattaforma distribuita, in particolare una versione del classico pacman riadattata per il multiplaying. Il sistema distribuito è stato progettato in modo da consentire l'interazione simultanea di più giocatori, risolvendo i principali problemi di coerenza risultanti dall'eliminazione delle centralità e dalla distribuzione del gioco su più nodi. Inoltre molta attenzione è stata posta sulle performance del gioco, incentrando la progettazione del sistema sui criteri di giocabilità e reattività in tempi accettabili, parametri fondamentali nella valutazione di questo genere di applicazioni.

1 Introduzione

L'obiettivo di questo progetto è la realizzazione del gioco Pacman in versione multiplayer e distribuita. A questo scopo sono state introdotte alcune modifiche alla versione classica per adattare il gioco a questo tipo di architettura. Infatti in questa variante, gli utenti possono scegliere di giocare sia con pacman che con un fantasmino e inoltre, possono essere attivi contemporaneamente più pacman e più fantasmini nella stessa partita, dando luogo ad una partita a due squadre rappresentate dalle due diverse fazioni.
In questo progetto, per gioco multiplayer è sempre inteso un ambiente in cui interagiscono più utenti attivi, i quali possono compiere azioni simultaneamente.
La modalità multiplayer delinea fortemente la progettazione del sistema distribuito su cui il gioco dev'essere eseguito. Infatti a differenza di ciò che accade in un sistema a partecipazione alternata, come un semplice gioco a turni, in un sistema in cui i partecipanti sono attivi contemporaneamente in ogni momento del gioco, è naturale pensare ad uno stato globale, condiviso tra i processi attivi, contenente il mondo in cui interagiscono i giocatori. In un'architettura client-server questo è intrinsecamente garantito dalla natura centralizzata del sistema: i giocatori inviano le azioni al server e quest'ultimo risponde a tutti i client con i conseguenti cambiamenti di stato. Ovviamente un sistema di questo tipo, nel fare affidamento su un singolo nodo, soffre dei ben noti problemi di robustezza e scalabilità. Nella progettazione di un sistema distribuito capace di supportare un gioco multiplayer, la sfida principale è quella di simulare lo stato globale, replicando il mondo in cui i giocatori interagiscono tra i diversi nodi di gioco e mantenendo la coerenza tra queste repliche locali. Un'ulteriore complicazione è dovuta alle performance di questo tipo di applicazioni, dal momento che ogni azione intrapresa da un nodo dev'essere comunicata a tutti gli altri nodi attivi in un tempo accettabile; questo si traduce in un aumento di traffico tra i nodi del sistema che occorre minimizzare per garantire la giocabilità.
In realtà entrambe le complicazioni sono due facce della stessa medaglia: per forzare la coerenza tra le repliche dello stato del mondo locali ai vari nodi, si tende ad aggiornare più frequentemente i nodi durante i cambi di stato, ottenendo appunto un maggiore traffico tra i nodi e quindi scarse performance.
Di seguito sono discusse e analizzate le problematiche riscontrate durante la progettazione del sistema. Nel paragrafo 2 si illustra in modo astratto l'architettura del sistema, focalizzando l'attenzione su problemi che riguardano in generale la classe di giochi multiplayer su sistemi distribuiti e riportando gli approcci risolutivi. Si illustra inoltre il sistema dal punto di vista dell'interazione con l'utente, descrivendo le varie fasi di gioco e i casi d'uso. Successivamente, nel paragrafo 3 si entra nel dettaglio implementativo degli approcci risolutivi, descrivendo gli algoritmi e gli schemi utilizzati, dapprima dal punto di vista delle interazioni tra i nodi del sistema e in secondo momento riportando le caratteristiche locali dei singoli nodi. Infine si illustrano i metodi di valutazione effettuati sul sistema riportandone i risultati ottenuti.

2 Architettura del sistema

L'architettura del sistema, con l'eccezione di alcuni casi particolari, è normalmente rappresentata da un grafo completamente connesso, in cui ogni nodo comunica con tutti gli altri. Questa topologia è particolarmente consona ad un gioco multiplayer distribuito, poiché consente la naturale propagazione del risultato di un'azione da un ambiente di gioco locale verso tutti gli altri nodi del sistema. Come spiegato successivamente il sistema è però in grado di riconfigurarsi in una diversa topologia al fine di garantire la coerenza tra i nodi di gioco in casi particolari.

2.1 Comportamento e interazioni regolari

Per costruire un sistema distribuito adatto all'esecuzione di un gioco in modalità multiplayer, occorre eliminare ogni centralità e replicare l'ambiente di gioco tra i

vari nodi, simulando di fatto uno stato globale condiviso tra i giocatori. Come già spiegato, è fondamentale mantenere la coerenza tra gli stati del mondo, prestando però molta attenzione alle performance. La soluzione adottata a tal fine, è quella di minimizzare quanto più possibile il traffico di dati tra i nodi. Questo è possibile traendo vantaggio dalla semplice dinamica del gioco realizzato. Osservando il gioco infatti, ogni personaggio, che sia un pacmac o un fantasmino, si muove tra le strade del labirinto in maniera del tutto elementare, continuando a viaggiare lungo la stessa direzione fino a quando non raggiunge un muro, oppure il cambio di direzione non è espressamente indicato dall'utente. Partendo da questo presupposto è possibile usare un protocollo in cui ogni nodo non comunica continuamente la propria posizione, ma invia a tutti gli altri un messaggio solo dopo un cambio di direzione. Il messaggio spedito contiene la posizione del personaggio e la nuova direzione impressa a partire da quel punto; per tanto è compito dei nodi riceventi continuare a spostare localmente il personaggio lungo quella direzione con una velocità costante nota tutti i nodi. Un approccio del genere soffre dei problemi dovuti ad eventuali ritardi di comunicazione tra i nodi della rete; al tempo stesso però, permette a un nodo di imprimere una coerenza tra il personaggio comandato da un giocatore e il corrispondente nel suo mondo, spostando "brutalmente" il personaggio nella giusta posizione successivamente alla ricezione del relativo messaggio. Un maggiore ritardo nella ricezione del messaggio corrisponde a uno spostamento più brutale che genera il fenomeno noto come **Latency Gap** (**Lag**), estremamente fastidioso durante le partite multiplayer in generale.
Questo approccio quindi preserva la coerenza tra i nodi di gioco a costo di un lag proporzionale al ritardo di ricezione. Il risultato, constatato in fase di test, è che in assenza di ritardi importanti è normalmente garantita una buona giocabilità: l'utente infatti non percepisce un cambiamento di posizione dei personaggi evidentemente irregolare rispetto alla naturale corsa.

2.2 Casi Particolari

Esistono eventi del gioco in cui è necessaria una forma di sincronizzazione più forte. In tali circostanze infatti la modifica del mondo da parte di un giocatore risulta critica poiché determina lo stato della partita; queste possono essere ricondotte inizialmente a due eventi specifici che avvengono quando:

- un pacman mangia un *powerdot*;
- un pacman e un fantasmino si scontrano.

Il primo evento determina un momentaneo cambiamento di stato nel mondo, che viene etichettato come *reverse*; questo stato rimane attivo tra i nodi un numero di secondi costante e noto a priori.
L'esito del secondo evento dipende dal primo: normalmente il pacman sarà mangiato dal fantasmino e perderà la partita, ma se è attivo un *reverse* avverrà il contrario. Un problema in questo caso potrebbe essere quello di determinare l'esito dello scontro se lo stato *reverse* è vero in uno solo dei mondi dei due rispettivi giocatori. Il secondo evento è problematico anche da un altro punto di vista: al momento dell'evento infatti, i due personaggi potrebbero trovarsi, ad esempio a causa di un ritardo, in posizioni differenti nei mondi dei due rispettivi giocatori; se le posizioni sono tali per cui i personaggi si scontrano in uno solo dei due mondi, è evidente un'incoerenza che porterà ad avere un giocatore perdente in un mondo, ma non nell'altro.
L'astrazione di questa classe di problemi così frequenti nei sistemi distribuiti è generalmente nota in letteratura come il **problema del consenso**; quest'ultimo si presenta tutte quelle volte in cui i processi partendo da opinioni differenti devono accordarsi su un'opinione comune. Esistono diverse soluzioni al problema del consenso. Nel caso specifico tale problema è stato risolto tramite l'elezione di un **leader**, il quale prende una decisione univoca e la comunica a tutti gli altri processi.
Esiste infine un terzo evento, non citato prima, che necessita particolare accortezza:
La scelta del personaggio con cui gli utenti giocheranno la partita.
In questo caso per garantire una forma di fair-play, è necessario dare all'utente

che inizializza la partita sul server la possibilità di scegliere il numero di pacman e fantasmini che crede sia più consono ai fini della giocabilità. Occorre quindi instanziare una mutua esclusione sulla lista dei personaggi disponibili durante tale scelta; ciò è realizzato mediante l'uso di uno schema **Token-Ring**, che impone ad ogni giocatore di effettuare una scelta rispettando il proprio turno.

2.3 Interazione con l'utente e casi d'uso

Prima di iniziare una partita occorre innanzitutto avviare il server specificando il numero di pacman e fantasmini desiderati. A questo punto, ogni giocatore dal proprio client si registra alla partita contattando il server e rimane in attesa dell'avvio del gioco che avviene non appena si raggiunge il numero di giocatori specificato in precedenza.
All'avvio del gioco, il sistema chiede ad ogni giocatore a turno di scegliere il personaggio con cui vorrà giocare la partita. Solo a questo punto ha inizio la partita vera e propria. L'interfaccia di utilizzo è molto semplice: l'utente può spostare il suo personaggio attraverso le frecce direzionali sulla tastiera. Altrettanto semplici sono i casi d'uso che si prospettano all'utente, che riprendono la versione classica del gioco pacman e che sono riportati nella figura sottostante.
Il diagramma mostra le due tipologie di utente e le relative operazioni che possono eseguire dopo aver dato il via alla partita, alcune di esse possono essere vincolate al risultato di azioni precedenti, mentre altre implicano l'esecuzione di altri metodi per il cambiamento dello stato globale del gioco. Come descritto dalle dipendenze in figura, l'inizio del gioco è strettamente legato ad una precedente registrazione al sistema stesso, così come la modifica dello stato `reverse` avviene quando un `pacman` mangia un `powerdot`; la `EatGhost` o la `EatPac` sono invece operazioni complementari possibili solo in presenza di un particolare valore del `reverse`, in base al quale si decide quale delle due è lecita e quale no.

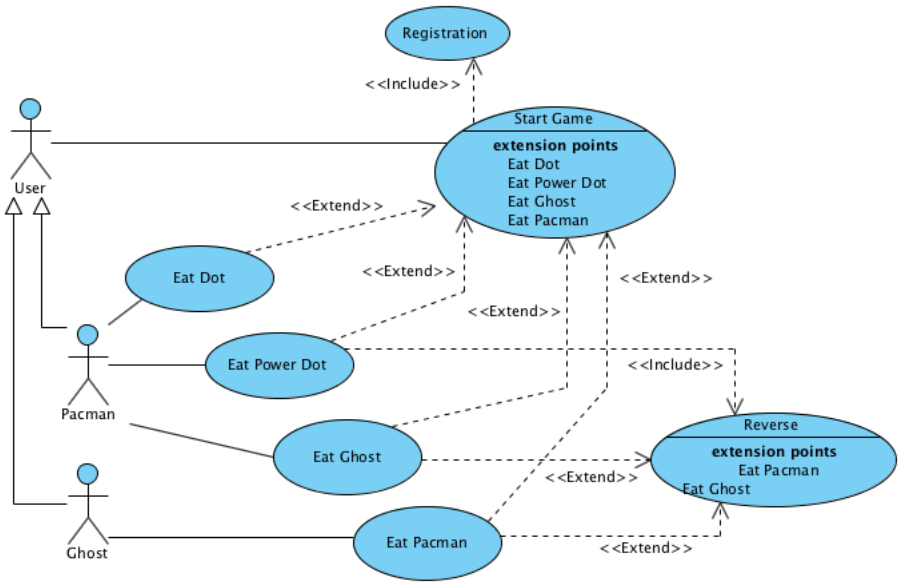


Figura 1: `Diagramma dei casi d'uso`

3 Aspetti implementativi

In questo paragrafo sono analizzati gli aspetti implementativi che coinvolgono gli schemi di comunicazione, la gestione dei crash nelle varie configurazioni del sistema, l'elezione del leader e in generale la dinamica del sistema nelle varie fasi di gioco, dalla registrazione alla fine della partita. Si descrivono inizialmente i comportamenti globali per poi scendere nel dettaglio analizzando la logica locale del singolo nodo.

3.1 Architettura del sistema nelle fasi di gioco

Come accennato precedentemente l'architettura del sistema durante il gioco è rappresentata da un **grafo completamente connesso**, in cui ciascun nodo comunica con tutti gli altri. Questa configurazione è raggiunta consecutivamente a una prima fase di registrazione, schematizzata di seguito.

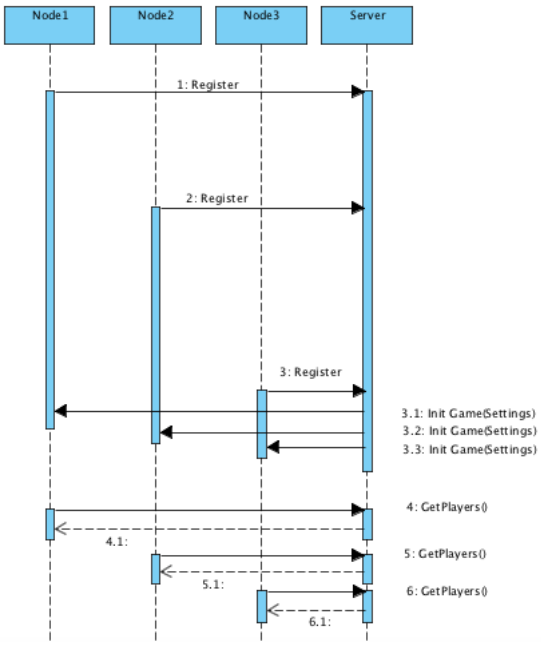


Figura 2: `Registrazione`

Inizialmente, ogni nodo che voglia entrare nel gioco, contatta un server attivo presso un indirizzo noto ed effettua una richiesta di registrazione. Il server, a sua volta, salva l'indirizzo del client registrante in una lista e si pone in attesa di nuove registrazioni. Una volta raggiunto il numero di partecipanti prefissato al momento della sua esecuzione, il server risveglia i client in attesa, dando inizio al gioco. Ogni nodo a questo punto recupera dal server la lista degli altri nodi e instaura con essi le connessioni che manterrà durante l'intera partita.
Il modello di comunicazione adottato nel sistema si basa interamente sulla tecnologia **RMI** (Remote Method Invocation), tramite la quale una classe esibisce un'interfaccia contenente un insieme di metodi che sono invocabili dai client remoti direttamente su una sua istanza locale. In questo modello dunque, la comunicazione da client a server avviene tramite l'esecuzione da parte del client di un metodo su un'istanza di un oggetto RMI locale al server. Viceversa, la comunicazione da

server a client avviene tramite *callback*: quando il client esegue un metodo sul server, passa come parametro un riferimento ad una sua istanza RMI, i cui metodi possono essere invocati dal server come risposta.
Successivamente alla fase di registrazione, ogni nodo si configura sia come server, ponendosi in attesa di una connessione dagli altri nodi, sia come client, instaurando una connessione con ciascuno di essi.
Va chiarito che da ora in poi i nodi costituiranno le componenti del sistema distribuito e saranno dunque completamente indipendenti dal server di registrazione. Sebbene il modello di comunicazione assuma per la maggior parte del tempo lo schema di un semplice grafo completo, in cui ogni nodo comunica con tutti gli altri, tale modello ha la capacità di riconfigurarsi per offrire più garanzie in corrispondenza di situazioni particolari. Come già spiegato in precedenza, una riconfigurazione della rete in Token-Ring è necessaria già all'inizio della partita, durante la scelta del personaggio, per assicurare la mutua esclusione sulla lista dei personaggi disponibili. Questo schema assicura che l'utente possa scegliere un personaggio solo quando entra in possesso di un token. Il token è un messaggio $\langle X,Y \rangle$ contenente il numero di pacman e di fantasmini disponibili; quando un nodo riceve il token sceglie uno tra i personaggi rimasti e decrementa di uno il relativo contatore, quindi passa il token al nodo successivo nell'anello. In questo modo le opzioni per i giocatori successivi sono limitate dalle scelte precedenti.

Gestione dei crash Se da un lato il Token-Ring rappresenta una soluzione in grado di garantire la mutua esclusione sulla scelta dei personaggi, dall'altro però rende molto delicata la gestione dei crash. Normalmente infatti il crash di un nodo è rilevato, grazie all'infrastruttura RMI, da un qualsiasi nodo che tenta di invocare un metodo sul nodo in crash, generando un'eccezione. Nella topologia a grafo completo prima o poi tutti i nodi rileveranno naturalmente un crash e rimuoveranno il nodo in questione dalla lista dei nodi attivi. In un token ring invece, un nodo in crash può essere identificato solo dal nodo attivo ad esso immediatamente precedente, al momento dell'invio del token. Quando ciò si verifica il nodo che rileva il crash provvede a rimuovere il nodo in crash dalla lista e a consegnare il token al nodo immediatamente successivo, come mostrato in figura.

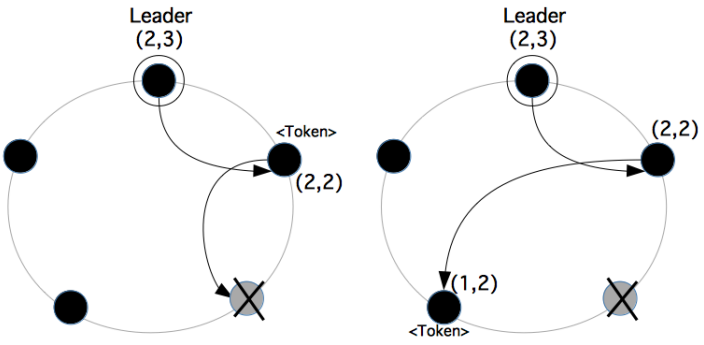


Figura 3: `Scelta dei personaggi: crash di un nodo`

Il caso è più complesso quando il crash si verifica su un nodo in possesso del token. In questo caso infatti, poiché il nodo precedente ha già consegnato il token non può più rilevare crash in posizioni successive; il nodo che segue il nodo in crash si pone a sua volta in un'attesa infinita che porta il sistema in deadlock. Per far fronte a questo problema è necessario introdurre un'ulteriore comunicazione, in senso inverso, in modo tale che il nodo in attesa del token possa rilevare un crash nei nodi precedenti. Questa tecnica è implementata tramite dei messaggi periodici

chiamati "TokenRequest", con cui un nodo si accerta che il suo predecessore sia ancora attivo: se un crash è rilevato, il nodo in questione può essere eliminato e le richieste indirizzate verso il nodo ancora precedente, come si può vedere nelle immagini sottostanti, finchè il token non è ricevuto. Il nodo che riceve una richiesta TokenRequest rispedisce il token al nodo richiedente, senza preoccuparsi in questo momento di eliminare i nodi in crash: lo farà subito dopo, non appena proverà a comunicare con questi durante il gioco.

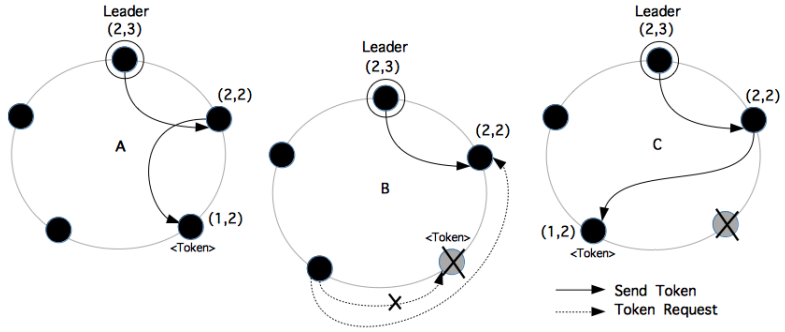


Figura 4: `Scelta dei personaggi: crash del nodo in possesso del token`

Leader Per assicurare la coerenza tra i mondi dei diversi nodi di gioco, si è optato, come già accennato, per l'assegnazione del ruolo di leader ad uno dei nodi. L'algoritmo di elezione del leader utilizzato è molto semplice. Ogni nodo condivide una copia della lista di tutti nodi attivi, sè compreso, creata dal server in fase di registrazione: il nodo in posizione minore è il leader. Ogni volta che i nodi rilevano un crash, aggiornano la lista ed eleggono eventualmente il nuovo leader, se è quest'ultimo ad aver subito un crash.
La presenza di un leader tra i nodi del sistema è quindi sempre garantita, sebbene esso entri in gioco solo nei casi necessari. Ad esempio, è il leader che genera il token nello schema precedente e lo invia al nodo successivo con i parametri settati in base alla sua scelta; quando riceve nuovamente il token deduce che tutti i giocatori hanno effettuato la loro scelta e quindi decrata l'inizio della partita, invocando il metodo *startgame()* su tutti i nodi attivi.
Il leader ha inoltre il compito di risolvere le circostanze di indecisione che scaturiscono dal noto problema del consenso. Nello specifico quando:

- un pacman mangia un *powerdot*;
- due personaggi di tipo diverso si scontrano.

Nel primo caso avviene un cambiamento di stato nel sistema, per cui durante un breve tempo è consentito ai pacman di mangiare fantasmini. L'inizio e la fine di questo stato, chiamato *reverse*, dev'essere sancito dal leader, per evitare incoerenze e sovrapposizioni nel tempo di più periodi di *reverse* avviati da giocatori differenti. Anche il secondo caso è arbitrato dal leader, infatti può accadere che personaggi opposti abbiamo un reverse locale differente al momento dello scontro; evento che produrrebbe risultati incoerenti se gestito localmente dai due nodi. La figura seguente rappresenta lo schema di arbitraggio del leader in questo caso specifico.
I nodi comunicano la loro azione al leader, il quale decreta un esito univoco e lo comunica in broadcast a tutti i nodi. Lo schema d'arbitraggio è uguale in entrambi i casi, l'unica differenza è che nel primo caso il leader comunica in broadcast sia l'inizio che la fine del *reverse*, l'ultima allo scadere di un timer impostato al momento del primo broadcast.

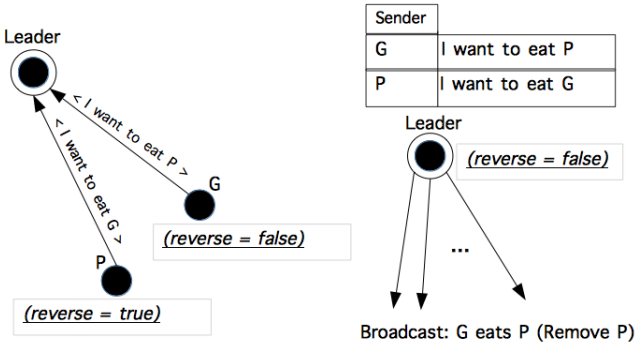


Figura 5: `Esempio di arbitraggio del leader`

In realtà questo schema descrive il comportamento del leader in maniera semplificata, poiché questo non si ritrova mai ad arbitrare azione contemporanee. Il In realtà questo schema descrive il comportamento del leader in maniera semplificata, poiché questo non si ritrova mai ad arbitrare azione contemporanee. Il leader si comporta invece in modo asincrono rispetto alle azioni: così come ogni altro nodo, mantiene uno stato locale che descrive il proprio mondo; quando riceve un azione verifica se questa è possibile o meno nel suo mondo e comunica il risultato in broadcast a tutti i nodi attivi. Riprendendo l'esempio in figura, se il suo stato *reverse* è falso, quando riceve l'azione di un fantasmino G che si appresta a mangiare un pacman P, il leader non trova incoerenze nel suo mondo e comunica il cambio di stato a tutti i nodi, ovvero che P è stato mangiato da G; quando riceverà l'azione di P che si appresta a mangiare G, il leader troverà più di un'incoerenza, sia per il valore del *reverse* che la rende impossibile, ma anche perché ormai P non esiste più nel suo mondo; decretata l'impossibilità dell'azione si limiterà a non comunicare alcun cambio di stato. Dato che ogni cambiamento di stato è comunicato solo dal leader, la coerenza tra i mondi dei nodi di gioco è quindi garantita.

Fine della partita La vittoria di una delle due squadre decreta la fine della partita: i pacman vincono quando riescono a mangiare tutti i pallini nel labirinto, mentre i fantasmini vincono quando riescono a mangiare tutti i pacman. Se un personaggio viene mangiato, il corrispondente giocatore è eliminato dal gioco, ma il nodo è mantenuto ancora vivo. Ciò significa che il nodo non trasmette più messaggi inerenti al suo personaggio, ma continua a riceverli dagli altri nodi; il giocatore diventa quindi un semplice spettatore. Nel caso in cui il nodo in questione sia leader, la leadership e le conseguenti funzioni sono mantenute.

3.2 Architettura del nodo

L'architettura di ogni singolo nodo del sistema può essere suddivisa in due componenti fondamentali: una racchiude le classi designate alla comunicazione con gli altri nodi durante il gioco e con il server in fase di registrazione; l'altra comprende le classi utilizzate per la gestione locale, tra cui l'aggiornamento della visuale di gioco e l'interazione con l'utente. Queste due componenti sono mostrate nei rispettivi diagrammi delle classi.

Gestione della comunicazione Nella componente di comunicazione si possono distinguere tre interfacce di comunicazione, ovvero due interfacce per l'interazione tra client e server in fase di registrazione, rispettivamente AuthServerInterface e AuthClientInterface, e un'interfaccia NodeInterface per l'interazione tra i nodi. Nella stesura del codice si è evidenziato concettualmente, raggruppando in classi distinte, il cambio architetturale da client-server a node-node. L'interfaccia NodeInterface contiene i metodi invocabili tramite RMI dagli altri nodi, questi metodi sono implementati nella classe NodeImpl ed utilizzati dalla classe principale

Node. Tramite questa componente i nodi si scambiano i dati relativi ai personaggi e si coordinano durante le varie fasi del gioco. La classe Node interagisce con entrambe le componenti e compare conseguentemente in entrambi i diagrammi delle classi.

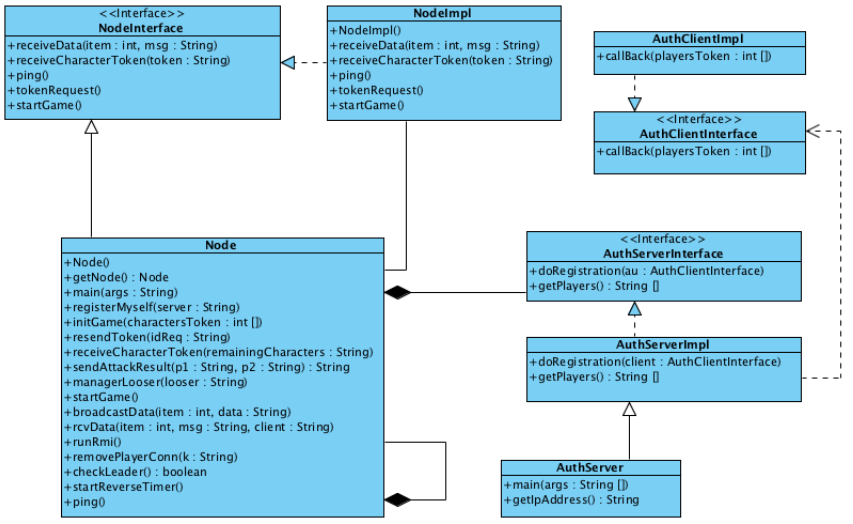


Figura 6: `Diagramma delle classi: comunicazione`

Gestione locale All'interno della componente di gestione locale le classi principali sono CharacterController e World: la prima offre all'utente il controllo del personaggio intercettando gli input da tastiera e imprimendo la giusta direzione al personaggio; la seconda invece aggiorna costantemente la visuale grafica che rappresenta il mondo, spostando tutti i personaggi lungo la relativa direzione all'interno del labirinto; quest'ultimo riflette il modello contenuto nella classe Maze, che comprende gli elementi propri del labirinto tra cui i dot, i powerdot e i muri.

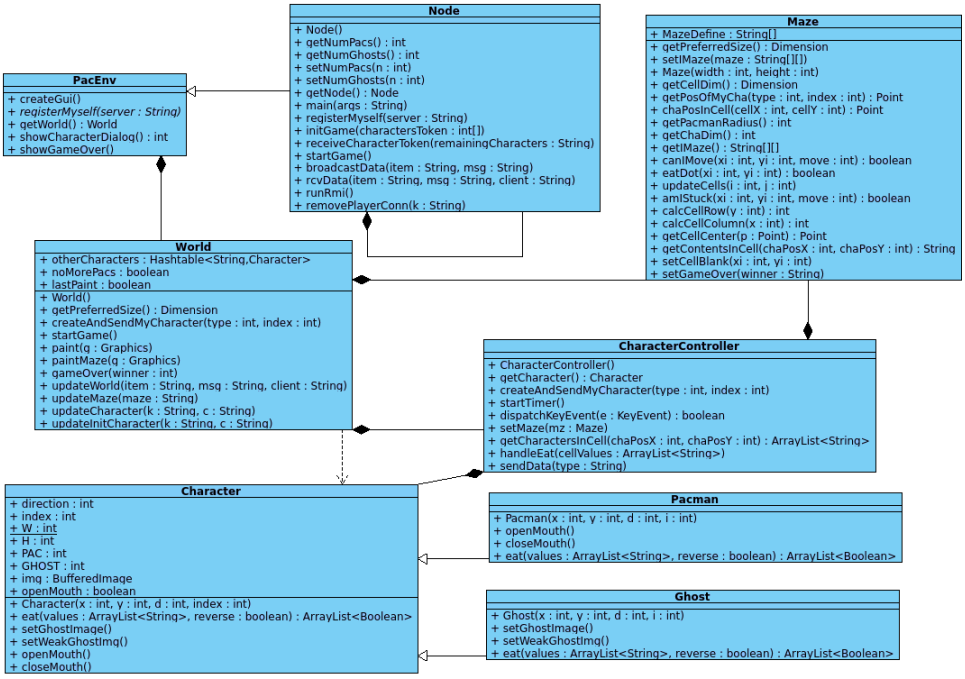


Figura 7: `Diagramma delle classi: gestione locale`

4 Analisi e valutazioni finali

Attraverso l'analisi e l'implementazione di un gioco multiplayer su sistema distribuito, si sono affrontate alcune delle problematiche più comuni relative a questa tipologia di sistema. I criteri di progettazione si sono incentrati sin dall'inizio sulla coerenza tra gli stati locali dei nodi del sistema e sulle performance di gioco. Particolare attenzione è stata spesa inoltre sull'analisi e l 'implementazione di algoritmi capaci di configurare autonomamente la rete del sistema in presenza di crash dei nodi nelle varie fasi di gioco.
Tenendo conto di questi criteri, dai diversi test effettuati si può concludere che il sistema risponde in maniera corretta e consente un'ottima giocabilità.
Nella valutazione del sistema si è trascurato, come da specifica, la presenza di guasti totalmente arbitrari, i cosiddetti guasti di tipo Bizantino; inoltre si è assunto che i canali di comunicazione fossero affidabili.
Un ulteriore aspetto cruciale, che condiziona la giocabilità su questo tipo di sistemi, è la presenza di eventuali latenze dovute a canali di comunicazione lenti, che possono dapprima generare i cosiddetti lag fino a compromettere totalmente la giocabilità. Anche da questo punto divista e in presenza di ritardi accettabili, il sistema realizzato continua a esibire una giocabilità soddisfaciente. Questo livello di performance è stato raggiunto principalmente grazie agli algoritmi e agli schemi di comunicazione utilizzati, progettati con lo scopo di minimizzare il traffico di comunicazione tra i nodi e descritti in dettaglio nei paragrafi precedenti.