Pcd-Assigment03 - part 2 Report

Distributed Agar.io

Bedeschi Federica
0001175655 federica.bedeschi4@studio.unibo.it

Pracucci Filippo
0001183555 filippo.pracucci@studio.unibo.it

Anno accademico 2024-2025

Indice

Analisi

1.1 Descrizione del problema

Agar.io è un popolare gioco multiplayer online in cui i giocatori controllano una cella circolare in un ambiente 2D. Lo scopo principale è di aumentare la propria dimensione consumando due tipi di entità:

- Cibo: piccole entità statiche distribuite casualmente, che permettono ai giocatori di aumentare la propria dimensione quando consumate;
- Giocatori: possono mangiare giocatori di dimensioni inferiori.

La partita termina quando un giocatore raggiunge 1000 unità di massa.

1.2 Visione distribuita

Seguendo un approccio distribuito abbiamo rispettato i seguenti requisiti:

- il gioco deve essere sempre attivo: i giocatori possono unirsi alla partita in ogni momento e cominciare a giocare;
- ogni giocatore deve avere una visione consistente del mondo, incluse le posizioni di tutti i giocatori e del cibo;
- il cibo deve essere generato casualmente e visibile a tutti i giocatori, e quando consumato deve essere rimosso dal sistema per tutti i giocatori;
- la condizione di fine partita deve essere controllata e imposta in maniera distribuita a tutti i giocatori.

Design e architettura

Abbiamo utilizzato il paradigma ad attori sfruttando il framework **Akka** in Scala, in particolare **Akka Typed** e lo stile funzionale coi **Behavior**. Per creare il sistema distribuito abbiamo utilizzato **Akka Cluster**. Per quanto riguarda l'architettura del sistema utilizziamo un cluster composto da un nodo (con ruolo "world") che ospita un **Cluster Singleton** per il manager del mondo e un nodo (con ruolo "player") per ogni giocatore che ospita il suo attore e quello della propria view. Abbiamo creato diversi attori sia per il model (quindi la business logic) che per la view (ovvero la visualizzazione e le interazioni con l'utente):

- model:
  - `WorldManager`: attore che crea e gestisce il mondo e coordina tutti gli altri attori;
  - `EatingManager`: attore che si occupa della logica di consumazione delle entità da parte del giocatore, aggiornando il mondo tramite la comunicazione con il `WorldManager`;
  - `EndGameManager`: attore che si occupa della logica di fine partita, segnalando lo stato al `WorldManager`;
  - `PlayerActor`: attore che rappresenta un giocatore, gestendo le proprie azioni e le comunicazioni con la propria view; è possibile crearlo in modalità AI, in tal caso la differenza è trasparente al resto degli attori;
- view:
  - `GlobalView`: attore responsabile dell'interfaccia grafica della visione globale del mondo;
  - `PlayerView`: attore responsabile dell'interfaccia grafica della visione del singolo giocatore e degli eventi generati dalle interazioni con l'utente per effettuare il movimento.

Abbiamo utilizzato il **Cluster Singleton** in modo da semplificare la gestione dello stato globale, nonostante introduca un *single point of failure* e potrebbe limitare la scalabilità. Tuttavia, abbiamo voluto garantire un certo livello di **fault tolerance** all'interno del Cluster Singleton adottando una strategia di restart in caso di fallimento del `WorldManager`. Inoltre, per non perdere lo stato del cibo al riavvio del `WorldManager` abbiamo utilizzato l'estensione **DistributedData** che ci permette di avere lo stato del mondo, e in particolare del cibo, distribuito fra i nodi. In questo modo la partita può continuare come se non fosse stata interrotta. Analogamente per `PlayerActor` e `PlayerView` abbiamo adottato una strategia di restart in caso di fallimento, in questo modo se il giocatore o la sua interfaccia subiscono un'interruzione durante l'esecuzione, la partita continua dal punto in cui si era interrotta grazie al riutilizzo del dato distribuito che mantiene lo stato del mondo.
Per quanto riguarda i nodi dei giocatori, `PlayerActor` e `PlayerView` comunicano in locale (come da diagramma in Figura 2.1) per effettuare i movimenti del giocatore, così da favorire la fluidità e in caso di errore di comunicazione attraverso la rete si predilige l'*availability* rispetto alla consistenza.

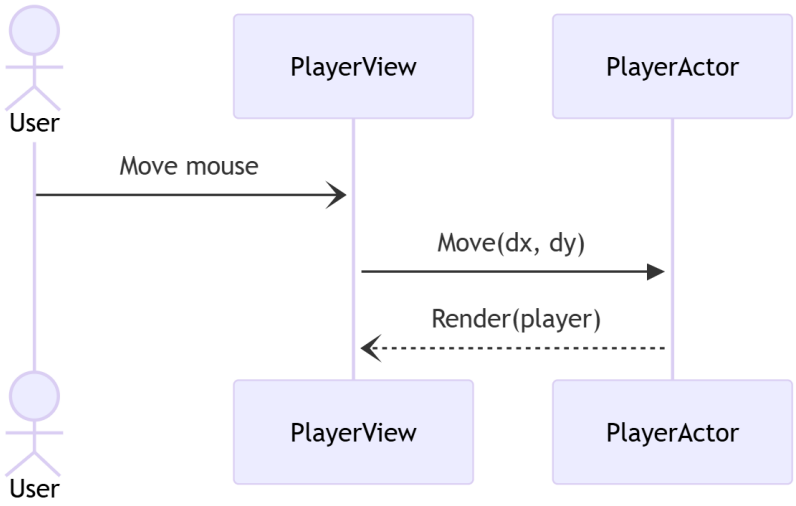

Figura 2.1: Diagramma di sequenza - comunicazione in locale tra PlayerActor e PlayerView

2.1 Interazioni

Per gestire le interazioni abbiamo utilizzato il pattern **Request-Response**, ad eccezione dei casi dove era necessario inviare messaggi a sè stessi per reagire ad un evento, in cui abbiamo sfruttato le **Future** e il `pipeToSelf`. Per far scoprire gli attori tra loro abbiamo utilizzato il **Receptionist**, che permette a un attore di registrarsi specificando il proprio *Service*, identificato con una *ServiceKey*, al quale gli attori interessati si possono sottoscrivere.

2.1.1 Interazioni del WorldManager

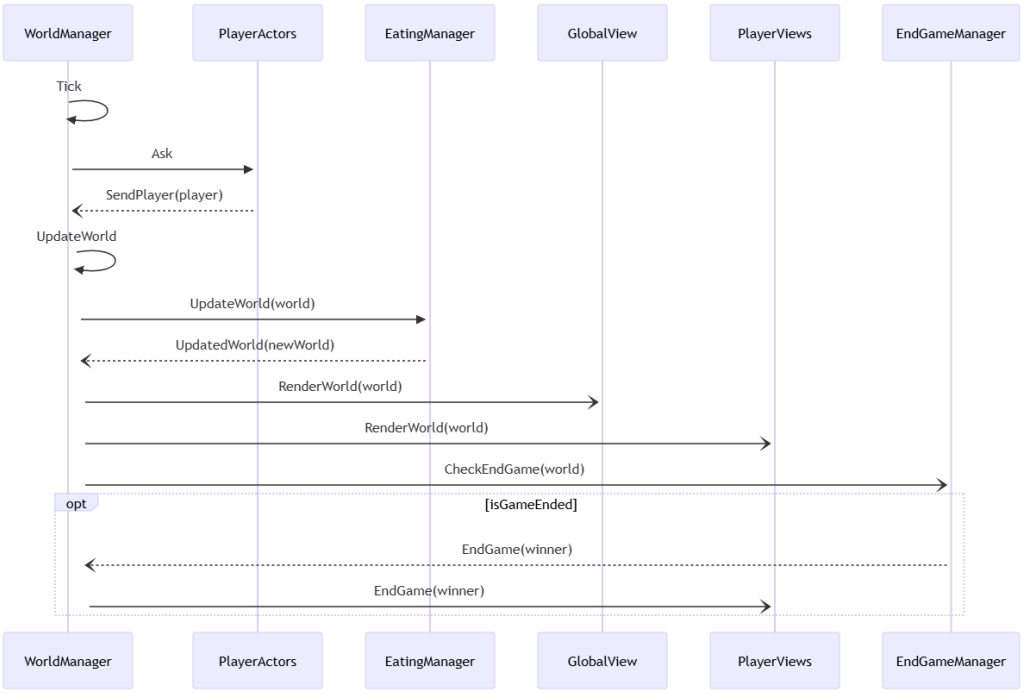

Figura 2.2: Diagramma di sequenza - interazioni del WorldManager

Il `WorldManager` per gestire il gioco simula un *gameloop* tramite l'invio di un messaggio a sè stesso (`Tick`) utilizzando un timer. Alla ricezione di ogni messaggio `Tick` si occupa di coordinare gli altri attori come mostrato nel diagramma in Figura 2.2. In particolare, per controllare la fine della partita si avvale dell'`EndGameManager` che gli invia il messaggio `EndGame(winner)` solo in caso ci sia un vincitore. In questo caso il `WorldManager` invia il messaggio `EndGame(winner)` alle `PlayerView` che si occupano di chiudere sè stesse e

terminare il proprio nodo, prima di terminare il proprio, che causa anche la chiusura della `GlobalView`.
Inoltre, in seguito alla ricezione del messaggio `UpdatedWorld(newWorld)` il `WorldManager`, oltre a mandare `RenderWorld(world)` alle view, comunica ai `PlayerActor` di aggiornarsi tramite messaggi `Grow` (se ha mangiato un'entità) o `Stop` (se è stato mangiato da un altro giocatore).

Conclusioni

Uno degli obiettivi principali era di mantenere lo stato del mondo distribuito e per farlo abbiamo sfruttato i **DistributedData**, i quali ci hanno anche permesso di garantire un riavvio degli attori ripartendo con lo stato del mondo in cui si erano interrotti, in questo modo l'utente non percepisce gli eventuali errori durante l'esecuzione. L'aver realizzato un sistema distribuito consente di allocare ogni giocatore su un nodo diverso, permettendo di lasciare o unirsi alla partita dinamicamente, in maniera trasparente rispetto alla posizione geografica. Un punto critico era quello di avere consistenza sulla visione del mondo da parte di tutti i giocatori; siamo riusciti ad ottenerla in tutti i casi ad eccezione di quelli in cui si verificano problemi di rete, nei quali la visione locale del giocatore pone maggiore priorità all'*availability*, grazie alla comunicazione locale tra `PlayerActor` e `PlayerView`.