PCD Assignment 03 - Distributed implementation of Agar.io

A cura di

Alessandra Versari - alessandra.versari2@studio.unibo.it

Lorenzo Rigoni - lorenzo.rigoni2@studio.unibo.it

Riccardo Moretti - riccardo.moretti6@studio.unibo.it

Analisi del problema

L'obiettivo della seconda parte dell'assignment 3 consisteva nel realizzare una versione distribuita del famoso gioco *Agar.io*, partendo dalla versione centralizzata fornita.
Agar.io è un gioco online multigiocatore in cui ciascun player controlla una cellula, con l'obiettivo di cibarsi di cellule più piccole per accrescere la propria massa.
Il gioco termina non appena una delle cellule dei giocatori raggiunge una massa pari a 1000.
All'interno del gioco è possibile distinguere due tipi di entità:

- Cellule cibo: hanno massa fissa, vengono generate, distribuite casualmente e sono statiche.
- Cellule dei giocatori: possono muoversi e mangiare tutte le cellule di massa inferiore alla propria.

Nel passaggio da versione centralizzata a distribuita, gli aspetti principali su cui ci siamo focalizzati sono stati:

- Possibilità di join dinamico: un giocatore deve poter iniziare a giocare in qualsiasi momento.
- Distributed food management: quando un giocatore mangia una cellula cibo, questa deve essere rimossa dalla vista di tutti i giocatori nel sistema.
- Consistent world view: ogni player deve avere una visione coerente del mondo di gioco. Tutti i giocatori devono essere visibili tra loro e le posizioni delle cellule cibo devono essere uguali per tutti.
- Distributed food generation: le cellule cibo vengono generate in modo casuale e le nuove cellule devono essere visibili a tutti i giocatori.
- Distributed game end condition: il gioco deve terminare per tutti non appena un giocatore raggiunge la massa obiettivo. Tutti gli altri devono essere informati della vittoria.

Design, strategia ed architettura

Per lo sviluppo della versione distribuita di Agar.io abbiamo scelto di utilizzare il framework Akka e il linguaggio Scala.
E' stato necessario ripensare il gioco utilizzando la programmazione ad attori, per questo sono stati creati i seguenti:

1) WorldManagerActor
2) PlayerActor
   2.1) PlayerMovementActor
   2.2) CollisionManagerActor
   2.3) ScoreManagerActor
3) ViewActor
4) GlobalViewActor

Segue la descrizione dettagliata di ciascun attore.

WorldManagerActor

Il WorldManagerActor rappresenta il cuore logico del sistema di gioco distribuito di Agar.io, agendo come orchestratore del mondo di gioco. È responsabile della gestione dello stato globale, contenente tutti i giocatori (Player), le entità di cibo (Food), e delle relative interazioni. All'avvio, viene inizializzato con uno stato World vuoto o preconfigurato e istanzia anche una GlobalViewActor, utile per visualizzare lo stato globale del mondo in tempo reale.

Tra le principali responsabilità, il WorldManagerActor:

- Registra i giocatori tramite il messaggio *RegisterPlayer*, creando un nuovo oggetto Player posizionato casualmente nella mappa e aggiornando la lista dei partecipanti. In risposta al messaggio, viene inviato al mittente un messaggio *InitialPlayerInfo* contenente le informazioni iniziali del giocatore (coordinate e massa), che il GameClient utilizza per istanziare un attore PlayerActor coerente con lo stato iniziale del server.
- Registra le view locali e globali tramite *RegisterView*, memorizzando gli ActorRef per inviare aggiornamenti periodici.
- Aggiorna i movimenti dei giocatori tramite *UpdatePlayerMovement*, aggiornando la posizione nel modello World.
- Gestisce la generazione di cibo a intervalli regolari tramite *GenerateFood*, garantendo che il numero totale di entità alimentari resti sotto una soglia predefinita.
- Rimuove entità di cibo tramite *RemoveFood* in seguito a collisioni.
- Gestisce la rimozione dei giocatori (*RemovePlayer*) terminando l'attore associato e rimuovendo il giocatore dal mondo.

- Aggiorna il punteggio/massa dei giocatori in seguito a eventi come collisioni (*UpdatePlayerScore*).
- Notifica la vittoria di un giocatore tramite *NotifyVictory*, loggando l'evento e comunicandolo a tutte le view registrate con il messaggio *DisplayVictory*.
- Termina l'esecuzione tramite il messaggio *Stop*.

Il comportamento è supportato da due scheduler periodici:

1) Uno che, ogni 50 ms, controlla collisioni tra giocatori e cibo, o tra giocatori stessi e aggiorna tutte le view con lo stato del mondo.
2) Uno che, ogni secondo, tenta di generare nuovo cibo se necessario, ossia se non è stato raggiunto il limite massimo di cibo esistente.

Questo attore è un punto centrale della distribuzione: esso riceve e coordina le azioni dagli attori locali dei giocatori, aggiorna lo stato condiviso e propaga le modifiche alle varie view (sia locali sia globali), assicurando coerenza e sincronizzazione tra tutti i nodi del sistema.

Nella nostra versione distribuita di Agar.io, abbiamo scelto di implementare il WorldManagerActor come Cluster Singleton, utilizzando le funzionalità offerte da Akka Cluster. Questo significa che, indipendentemente dal numero di nodi attivi nel cluster (client o server), esisterà sempre una sola istanza attiva del *WorldManagerActor* in tutto il sistema.

Abbiamo scelto di utilizzare Akka Cluster Singleton per il *WorldManagerActor* perché il progetto prevede un unico mondo di gioco condiviso da tutti i player, che deve essere gestito in modo coerente e centralizzato, ma al contempo resiliente e scalabile in ambiente distribuito. Abbiamo scelto di scartare altri approcci perché non li abbiamo ritenuti adatti al nostro contesto, avrebbero garantito maggiore scalabilità ma anche più complessità e in questo caso la priorità era mantenere i dati coerenti.

Nella nostra applicazione sono presenti due oggetti, necessari per l'avvio del gioco:

- Il server (*GameServer*) crea inizialmente il mondo e lancia il singleton WorldManagerActor come punto centrale del sistema.
- I client (*GameClient*) si connettono al cluster e utilizzano ClusterSingleton(system).init(...) per ottenere un riferimento a quell'unica istanza, anche se localmente non viene creata. Da quel momento possono inviare messaggi come *RegisterPlayer*, *UpdatePlayerMovement*, *RegisterView*, etc.

PlayerActor

Il PlayerActor rappresenta il controller principale di un singolo giocatore all'interno del mondo di gioco distribuito. Ha il compito di ricevere i comandi di movimento, reagire agli eventi di gioco (come collisioni con cibo o altri giocatori) e coordinare la gestione dello stato del giocatore stesso. Per garantire una separazione delle responsabilità e una maggiore modularità, il PlayerActor è strutturato attorno a tre attori secondari, ognuno con una responsabilità distinta:

1) *PlayerMovementActor*: gestisce la logica di movimento del giocatore in base agli input ricevuti e aggiorna la posizione nel mondo.
2) *CollisionManagerActor*: riceve notifiche di collisione dal WorldManagerActor (FoodCollision, PlayerCollision) e si occupa di elaborarle, decidendo se un giocatore deve mangiare o essere mangiato.
3) *ScoreManagerActor*: gestisce il punteggio/massa del giocatore e l'eventuale vittoria.

Il PlayerActor funge quindi da punto di smistamento dei messaggi tra questi sottocomponenti e il resto del sistema. Non gestisce direttamente logiche complesse, ma le delega ai suoi sottocomponenti.

Il PlayerActor è in grado di gestire i seguenti messaggi:

- *Move(x, y)*: viene inoltrato al PlayerMovementActor per aggiornare la direzione desiderata.
- *Tick*: messaggio periodico (ogni 50 ms) per far avanzare il movimento in modo regolare.
- *FoodCollision(food, x, y)* e *PlayerCollision(player, x, y)*: inoltrati al CollisionManagerActor per gestire la collisione.
- *CurrentScore(score)* : utilizzato per aggiornare il punteggio del giocatore nel ScoreManagerActor.
- StopPlayer: utilizzato per fermare il PlayerActor. Per esempio viene utilizzato quando il player viene mangiato.

Abbiamo scelto di suddividere le responsabilità tra i vari sotto attori per ottenere una gestione più modulare, cercando di mantenere ciascun attore il più semplice possibile, isolando le logiche.

PlayerMovementActor

Il PlayerMovementActor è un sotto-attore specializzato del PlayerActor responsabile esclusivamente della gestione del movimento di un giocatore all'interno del mondo di gioco. Si occupa di mantenere lo stato locale della posizione del player (x, y) e della direzione corrente di movimento, elaborando gli input direzionali ricevuti in tempo reale. Questo attore è progettato per operare in modo autonomo, rispondendo ciclicamente ai tick di aggiornamento e comunicando ogni cambiamento di posizione al WorldManagerActor.

Funzionalità principali:

1) Gestione della direzione: quando riceve un messaggio *Move(dx, dy)*, aggiorna la direzione desiderata del giocatore senza modificare immediatamente la posizione.
2) Aggiornamento della posizione: a ogni messaggio *Tick* (inviato periodicamente dal PlayerActor ogni 50ms), il PlayerMovementActor calcola la nuova posizione del giocatore in base alla direzione corrente e alla velocità definita (speed).
3) Comunicazione col mondo di gioco: dopo ogni aggiornamento, l'attore invia un messaggio *UpdatePlayerMovement* al WorldManagerActor, che aggiorna la rappresentazione globale dello stato del gioco con la nuova posizione del giocatore.

CollisionManagerActor

Il CollisionManagerActor è un componente specializzato del PlayerActor responsabile della gestione delle collisioni tra il giocatore associato e gli altri oggetti del mondo di gioco, come cibo e altri giocatori. È incaricato di determinare se una collisione può effettivamente comportare un evento di “consumo” e di informare gli attori responsabili (es. ScoreManagerActor e WorldManagerActor) per aggiornare correttamente lo stato del sistema.

Funzionalità principali:

1) Gestione collisioni con il cibo: quando riceve un messaggio *FoodCollision*, salva temporaneamente il cibo in collisione (*pendingFood*) e la posizione del giocatore. Poi richiede allo ScoreManagerActor la massa attuale del player, necessaria per determinare se può mangiare quel cibo.

2) Gestione collisioni con altri giocatori: analogamente al caso del cibo, quando riceve un messaggio *PlayerCollision*, salva il giocatore avversario in collisione (*pendingPlayer*) e chiede la massa attuale del player per determinare se può mangiarlo.

3) Determinazione effettiva del consumo: dopo aver ricevuto la massa corrente tramite *CurrentMass*, l’attore usa l’EatingManager per verificare:
    - Se il player può mangiare il cibo (*canEatFood*), in tal caso:
        - Notifica al ScoreManagerActor l’incremento di punteggio.
        - Richiede al WorldManagerActor di rimuovere il cibo dal mondo.
    - Se può mangiare un altro giocatore (*canEatPlayer*), in tal caso:
        - Aumenta il punteggio proporzionalmente alla massa dell’avversario.
        - Richiede la rimozione del player sconfitto dal mondo.

ScoreManagerActor

Lo ScoreManagerActor è il componente responsabile della gestione e del tracciamento del punteggio (massa) del giocatore nel sistema di gioco distribuito. È uno dei tre sotto-attori creati dal PlayerActor per delegare compiti specifici, in questo caso la logica legata all'evoluzione del punteggio, alla verifica della vittoria e alla comunicazione con il WorldManagerActor.

Funzionalità principali:

1) Aggiornamento del punteggio: quando riceve un messaggio *CurrentScore*, incrementa il punteggio interno (score) con la massa ricevuta. Dopo l'aggiornamento:
    - Invia un messaggio *UpdatePlayerScore* al WorldManagerActor, che aggiorna lo stato globale del mondo.
    - Se il punteggio raggiunge o supera la soglia di vittoria (winScore) e il giocatore non ha ancora vinto, invia *NotifyVictory* al WorldManagerActor per notificare la fine della partita.

2) Risposta a richieste di massa: quando riceve un messaggio *RequestCurrentMass*, restituisce il punteggio attuale sotto forma di *CurrentMass*, utile ad altri attori (come il CollisionManagerActor) per valutare se il giocatore può effettuare un'azione di "consumo".

3) Controllo stato di vittoria: una variabile booleana interna garantisce che la vittoria venga segnalata una sola volta, evitando comportamenti duplicati in caso di ulteriori incrementi di massa.

ViewActor e GlobalViewActor

Gli attori ViewActor e GlobalViewActor si occupano della gestione dell'interfaccia grafica del gioco, permettendo di visualizzare in tempo reale l'evoluzione del mondo di gioco distribuito in stile Agar.io. Entrambi integrano delle viste Swing, ma con finalità distinte.

- *ViewActor* rappresenta la vista locale di un singolo giocatore: è associato a un PlayerActor specifico e si occupa sia di visualizzare il mondo filtrato per quel giocatore, sia di gestire l'input dell'utente. Riceve i comandi e li inoltra al PlayerActor, mentre aggiorna la vista grafica tramite *UpdateView*.
- *GlobalViewActor*, invece, gestisce una vista globale centralizzata del mondo di gioco, utile ad esempio per debugging o monitoraggio da parte del server. Non riceve input interattivi, ma si limita ad aggiornare lo stato visualizzato ogni volta che riceve un messaggio UpdateView contenente il World completo.

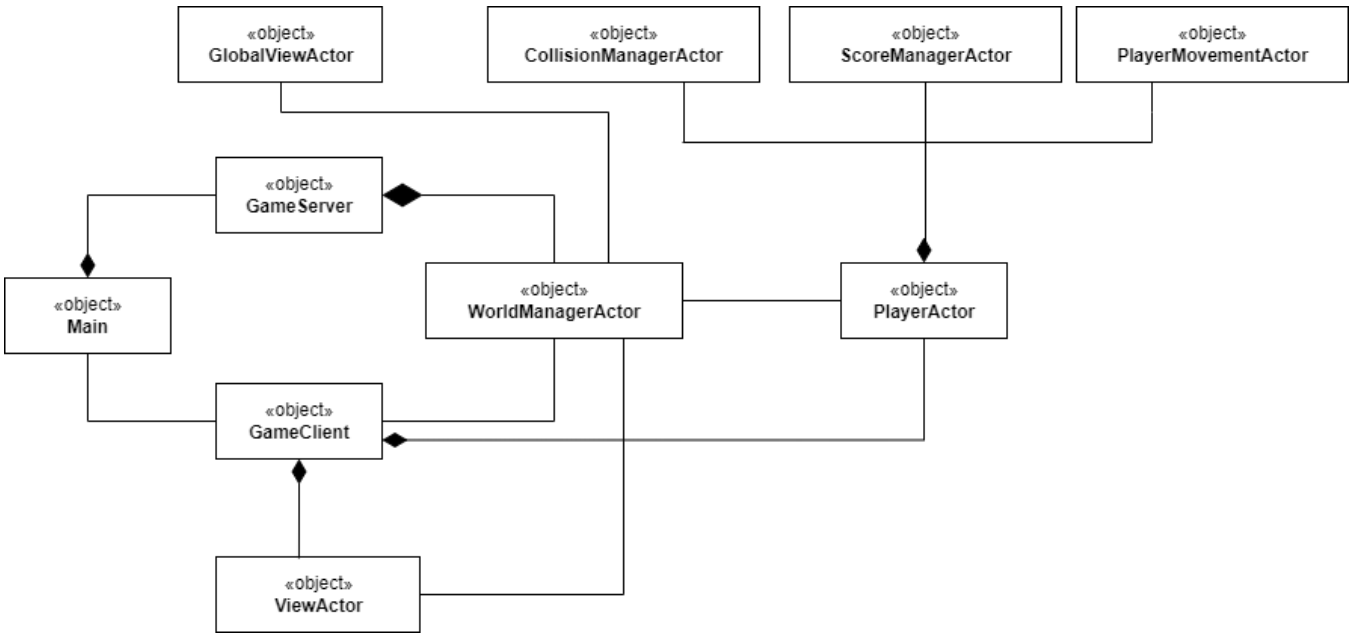

Figura 1: Diagramma UML dell'architettura del sistema

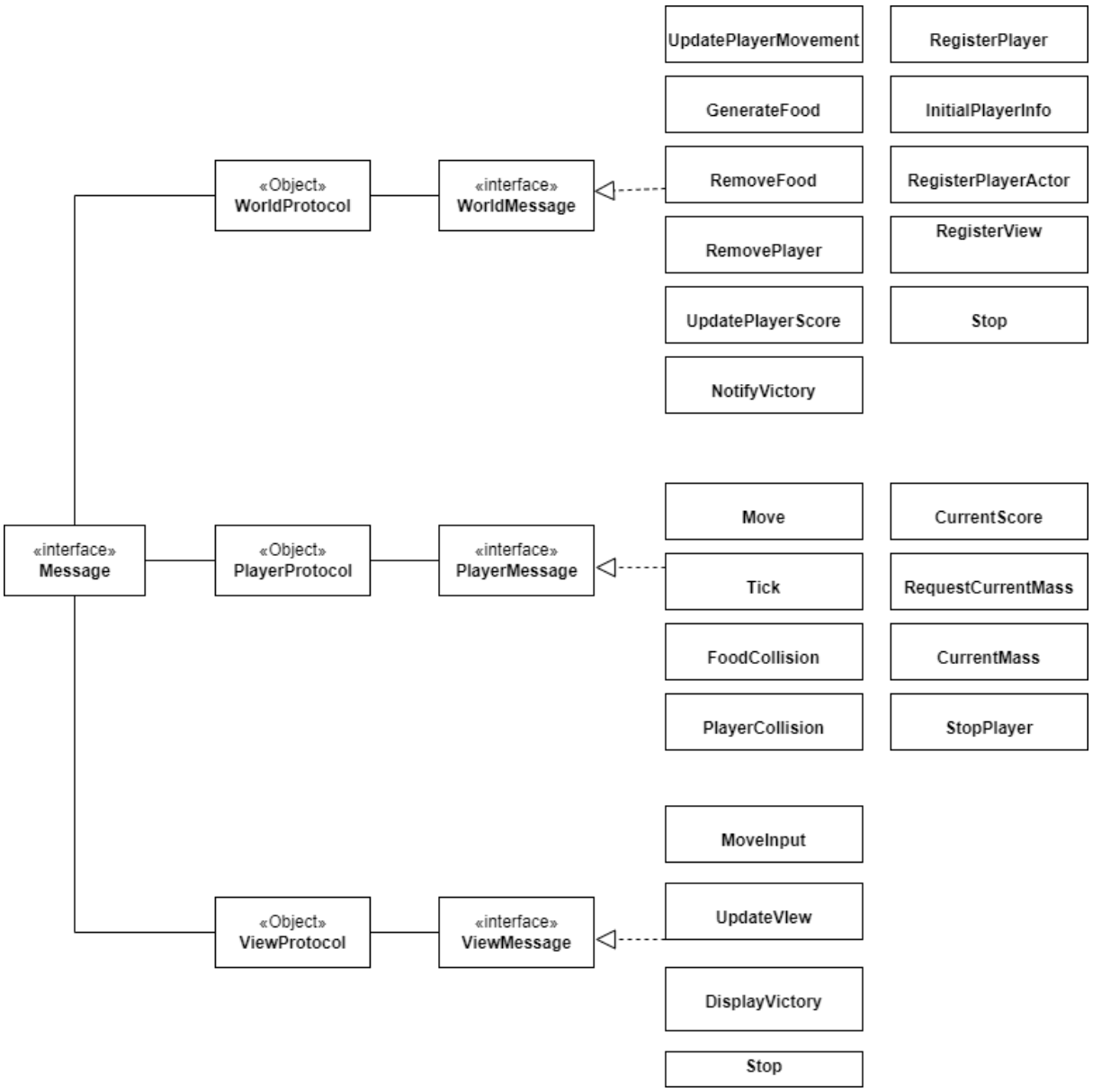

Figura 2: Messaggi scambiati tra attori