Assignment03

Giacomo Foschi
Matricola: 0001179137
Email: giacomo.foschi3@studio.unibo.it

Giovanni Pisoni
Matricola: 0001189814
Email: giovanni.pisoni@studio.unibo.it

Giovanni Rinchiuso
Matricola: 0001195145
Email: giovanni.rinchiuso@studio.unibo.it

Gioele Santi
Matricola: 0001189403
Email: gioele.santi2@studio.unibo.it

Capitolo 1 - Analisi del problema

L'assignment si inserisce nel contesto della programmazione concorrente e distribuita, ambito fondamentale per lo sviluppo di sistemi scalabili e reattivi che siano in grado di gestire un elevato numero di entità interagenti in modo efficiente e affidabile. In particolare, il lavoro si concentra sull'uso del **modello ad attori** e di tecnologie di comunicazione basate sul **message passing**, strumenti chiave per realizzare sistemi distribuiti moderni.

Il problema è suddiviso in tre parti principali, ognuna delle quali affronta aspetti specifici di questa disciplina: la simulazione di un gruppo di agenti autonomi che interagiscono in maniera asincrona, la realizzazione di un gioco multiplayer distribuito e, infine, alcune estensioni e tecnologie alternative che permettono di esplorare ulteriori modelli di comunicazione distribuita.

Comunicazione asincrona tramite attori

La simulazione del comportamento collettivo di entità autonome, noti come *boids*, è un esercizio complesso che mette alla prova sia la modellazione computazionale sia la progettazione architetturale di sistemi concorrenti. I boids si muovono in uno spazio bidimensionale seguendo tre semplici regole locali: evitare collisioni con i vicini (separazione), dirigersi nella stessa direzione del gruppo (allineamento) e cercare di rimanere vicino agli altri membri dello stormo (coesione).

Dal punto di vista tecnico, la vera sfida è trovare una soluzione che modelli ogni boid come un'entità indipendente che riesca a interagire con gli altri senza generare conflitti, race condition o deadlocks. I modelli concorrenti tradizionali, basati su thread e meccanismi di sincronizzazione esplicita, tendono a essere rigidi, difficili da scalare e spesso inclini a errori difficili da individuare. In alternativa, il modello ad attori offre una prospettiva più moderna, in cui ogni entità è completamente autonoma e comunica solo tramite messaggi asincroni, evitando quindi la condivisione diretta di memoria, tra cui le critical section, e di conseguenza la necessità di sincronismi di accesso alle risorse.

Questo approccio si adatta particolarmente bene a un contesto come quello della simulazione dei boids. Ogni boid può essere implementato come un attore, responsabile esclusivamente del proprio stato interno, che aggiorna in base ai messaggi ricevuti dagli attori vicini. Tuttavia, passare a un'architettura basata su attori non è privo di difficoltà. Una delle sfide principali riguarda la sincronizzazione degli aggiornamenti. In una simulazione in tempo reale, è essenziale che tutti i boid si aggiornino in modo sufficientemente coordinato, altrimenti si rischiano artefatti visivi o comportamenti incoerenti. Il fatto che la comunicazione tra attori sia asincrona e che non esista uno stato condiviso rende questa coordinazione particolarmente complessa, costringendo a ideare meccanismi alternativi, come attori supervisori o cicli di aggiornamento regolati.

C'è poi un altro aspetto da considerare: come strutturare il sistema di attori in modo efficiente. Non si può semplicemente modellare ogni boid come un attore senza prendere i considerazione il numero totale di entità, il carico di messaggi o le interazioni necessarie. Bisogna evitare che si crei un'eccessiva granularità, che renderebbe il sistema inefficiente, o un'eccessiva centralizzazione, che ne limiterebbe la scalabilità. Ogni attore deve comunicare solo con i boid realmente vicini, limitando l'invio di messaggi e riducendo il traffico complessivo.

Anche l'integrazione con l'interfaccia grafica gioca un ruolo cruciale. La simulazione deve essere visualizzata in modo fluido, con aggiornamenti costanti e coerenti. La GUI non deve interferire con il

calcolo della simulazione, altrimenti ne risentono le prestazioni complessive. Il modello ad attori, in questo caso, permette di isolare la logica grafica in attori dedicati, che ricevono aggiornamenti dal "motore" della simulazione, ma è fondamentale progettare con attenzione la comunicazione tra questi livelli per evitare colli di bottiglia e ritardi percettibili.

Per gestire tutti gli aspetti sopra citati, l'uso di un framework come Akka si rivela particolarmente utile. Akka implementa il modello ad attori in modo robusto e offre strumenti avanzati per la gestione della concorrenza, la supervisione degli attori e la resilienza del sistema. Grazie a questa infrastruttura, è possibile costruire una simulazione reattiva, modulare e sufficientemente scalabile, capace di supportare un numero elevato di boid senza compromettere l'affidabilità o la qualità visiva dell'esperienza.

Distributed Computing

Il secondo punto dell'assignment consiste nell'estensione di Agar.io, un gioco multiplayer in cui ogni giocatore controlla una cellula che si muove in un ambiente condiviso, passando a un'architettura distribuita locale.

Uno degli aspetti principali riguarda la sincronizzazione tra i nodi. Tutti i partecipanti devono avere una visione coerente del mondo di gioco: quando un giocatore raccoglie un pezzo di cibo o interagisce con un altro giocatore, l'evento deve essere gestito da un singolo nodo e comunicato tempestivamente agli altri. Per evitare comportamenti incoerenti, è importante che la gestione degli eventi sia centralizzata e correttamente coordinata.

Anche la generazione e la rimozione del cibo richiedono coordinamento, è infatti necessario garantire che due nodi non generino lo stesso elemento o che un cibo già consumato non rimanga visibile per altri giocatori. Lo stesso vale per la gestione dello stato dei giocatori: se una cellula viene assorbita, tutti i nodi devono riflettere il cambiamento in modo sincrono.

Dal punto di vista tecnico, per implementare questa architettura si è scelto di utilizzare Akka Cluster**,** un'estensione del framework Akka che consente di distribuire attori tra più nodi mantenendo il paradigma del message-passing. Questa scelta consente di modellare in modo chiaro le entità di gioco (come giocatori, cibo o gestori di mappa) come attori indipendenti, delegando al framework la gestione della comunicazione tra nodi, del bilanciamento del carico e della rilevazione della disponibilità dei peer.

Rock-Paper-Scissors con message parsing sincrono in Go

L'implementazione del gioco carta-forbice-sasso utilizzando il modello di concorrenza di Go rappresenta un interessante caso di studio sul message passing sincrono. In questo contesto, arbitro e giocatori agiscono come entità concorrenti indipendenti, ciascuna incarnata da una *goroutine*, e collaborano tra loro scambiandosi messaggi attraverso canali dedicati. Questo approccio consente di simulare efficacemente un ambiente concorrente in cui le entità cooperano attraverso lo scambio

esplicito di messaggi, seguendo il paradigma CSP (Communicating Sequential Processes), su cui si basa il modello di concorrenza di Go.

La sfida principale risiede nella definizione e nel rispetto di un protocollo di comunicazione ben strutturato. L'arbitro funge da coordinatore centrale del gioco e deve gestire un ciclo preciso di operazioni: avviare una nuova partita, richiedere le mosse a ciascun giocatore, attendere entrambe le risposte, determinare il risultato dello scontro (vittoria, pareggio o sconfitta) e infine comunicare l'esito ai giocatori. Tutto questo deve avvenire in modo ordinato e senza che le goroutine restino bloccate in attesa di messaggi che non arriveranno mai.

Un punto critico dell'implementazione riguarda la sincronizzazione delle mosse: l'arbitro non può determinare l'esito finché non ha ricevuto entrambe le mosse, e i giocatori devono attendere che venga richiesto il loro input prima di inviarlo. Questo richiede che i canali siano utilizzati con attenzione, evitando race condition, deadlock o perdita di messaggi. Ad esempio, una possibile soluzione prevede che l'arbitro invii un segnale di "inizio turno" a ciascun giocatore, i quali rispondono con la propria scelta tramite canali dedicati. Solo dopo aver ricevuto entrambe le mosse, l'arbitro può processare il risultato e inviarlo ai rispettivi giocatori.

Un ulteriore aspetto interessante riguarda la gestione dell'isolamento e della sequenzialità: ogni goroutine è indipendente, ma partecipa a un protocollo che impone vincoli temporali e di dipendenza tra le operazioni. Il modello CSP incoraggia una progettazione in cui le entità mantengono stati locali e comunicano solo tramite messaggi, evitando la condivisione diretta della memoria. Questo approccio, oltre a ridurre la complessità di sincronizzazione esplicita, promuove un'architettura modulare e scalabile.

Capitolo 2 - Strategia risolutiva e architettura

Comunicazione asincrona tramite attori

La soluzione adottata per l'implementazione dei Concurrent Boids con Akka si basa su una decomposizione del sistema in tre tipologie principali di attori, ciascuno con responsabilità ben definite e isolate. Questa architettura riflette il principio fondamentale del modello ad attori: entità autonome che mantengono il proprio stato privato e comunicano esclusivamente attraverso messaggi asincroni.

Il punto di ingresso del sistema è la classe BoidsSimulation, che contiene il metodo main responsabile dell'avvio dell'intero sistema di attori. Questa classe crea l'ActorSystem di Akka con il ManagerActor come attore radice, denominandolo "boids-system". La semplicità di questa classe nasconde la complessità del sistema sottostante, dimostrando come Akka permetta di astrarre i dettagli dell'infrastruttura di concorrenza. Il main si blocca poi sulla terminazione del sistema, garantendo che l'applicazione rimanga attiva finché la simulazione è in esecuzione.

Il cuore del sistema è rappresentato dal ManagerActor, che funge da orchestratore centrale della simulazione. Questo attore ha la responsabilità di creare e gestire l'intero ciclo di vita dei BoidActor, coordinare la sincronizzazione degli aggiornamenti, e interfacciarsi con la GUI per la visualizzazione. La decisione di centralizzare queste responsabilità in un singolo attore è stata guidata dalla necessità di mantenere una visione coerente dello stato globale della simulazione e di coordinare efficacemente gli aggiornamenti sincronizzati di tutti i boids.

L'implementazione del ManagerActor utilizza diversi stati comportamentali attraverso il pattern di Akka Typed Behaviors. Lo stato iniziale idle attende il comando StartSimulation dal GUIActor. Una volta ricevuto, il Manager transita nello stato running dove crea tutti i BoidActor necessari, generando per ciascuno posizioni e velocità iniziali casuali. Un aspetto interessante dell'implementazione è l'uso di TimerScheduler di Akka per generare eventi Tick periodici ogni 40 millisecondi, che guidano l'intera simulazione. Il Manager mantiene due strutture dati fondamentali: una mappa dei BoidActor attivi e una mappa degli stati correnti di tutti i boids, garantendo così una visione consistente del mondo.

La gestione dei parametri della simulazione è incapsulata nella classe BoidsParams, che mantiene le costanti fisiche del sistema come i raggi di percezione e evitamento, la velocità massima, e i pesi per i tre comportamenti fondamentali. Questa classe fornisce anche metodi di utilità per calcolare i confini del mondo e gestire il wrap-around dell'ambiente. La separazione di questi parametri in una classe dedicata permette una facile configurazione e modifica del comportamento della simulazione senza dover modificare la logica degli attori.

Ogni boid nella simulazione è rappresentato da un BoidActor indipendente. Questa scelta progettuale sfrutta appieno il modello ad attori, permettendo a ciascun boid di mantenere il proprio stato interno (posizione e velocità) e di calcolare autonomamente le forze comportamentali basandosi sulle informazioni ricevute dal Manager. L'implementazione del calcolo delle forze segue fedelmente l'algoritmo classico dei boids, con ogni attore che calcola separazione, allineamento e coesione basandosi sulla lista dei vicini ricevuta.

La classe BoidActor implementa un sofisticato sistema di stati comportamentali. Quando riceve un UpdateRequest, calcola prima i vicini entro il raggio di percezione utilizzando il metodo findNearby, che filtra efficientemente solo i boids rilevanti. I tre comportamenti fondamentali sono implementati in metodi separati: calculateSeparation genera una forza repulsiva dai vicini troppo vicini, calculateAlignment allinea la velocità con la media del gruppo, e calculateCohesion attrae verso il centro di massa dei vicini. Questi vettori vengono poi combinati utilizzando i pesi configurabili per produrre la velocità finale, che viene limitata alla velocità massima per mantenere il realismo della simulazione.

I protocolli di comunicazione tra gli attori sono definiti attraverso interfacce che garantiscono type-safety e chiarezza semantica. Il ManagerProtocol definisce tutti i comandi che il Manager può ricevere, includendo StartSimulation per l'inizializzazione, UpdateParams per la modifica dinamica dei parametri, e i comandi di controllo come PauseSimulation e StopSimulation. Particolarmente importante è il messaggio BoidUpdated che ogni BoidActor invia dopo aver completato il suo aggiornamento, contenente la nuova posizione e velocità.

Il BoidProtocol definisce i messaggi specifici per i BoidActor. L'UpdateRequest è il messaggio principale che contiene il tick corrente e la lista completa degli stati di tutti i boids, permettendo a ciascun boid di calcolare le sue interazioni. Il WaitUpdateRequest è un meccanismo per gestire la

pausa della simulazione: quando ricevuto, il boid entra in uno stato di attesa dove ignora ulteriori WaitUpdateRequest ma rimane pronto a riprendere quando riceve un nuovo UpdateRequest.

Il GUIProtocol gestisce la comunicazione con l'interfaccia grafica. RenderFrame trasporta i dati necessari per visualizzare un frame completo, includendo le posizioni di tutti i boids e le metriche di performance come il framerate. UpdateWeights permette la modifica interattiva dei parametri comportamentali, mentre i messaggi di conferma come ConfirmPause e ConfirmResume implementano un handshake che garantisce che la GUI rifletta sempre lo stato corrente della simulazione.

La sincronizzazione degli aggiornamenti è stata risolta attraverso un meccanismo basato su tick. Il ManagerActor utilizza un timer interno che genera eventi Tick ogni 40 millisecondi. Ad ogni tick, il Manager invia a tutti i BoidActor un messaggio UpdateRequest contenente lo stato corrente di tutti i boids nel sistema. Questo approccio garantisce che tutti i boids basino i loro calcoli sullo stesso snapshot dello stato globale, evitando inconsistenze dovute a race condition.

Un altro aspetto dell'implementazione riguarda la gestione del completamento degli aggiornamenti. Ogni BoidActor, dopo aver calcolato la sua nuova posizione, invia un messaggio BoidUpdated al Manager. Il Manager tiene traccia del numero di risposte ricevute attraverso una variabile atomica completedBoids e solo quando tutti i boids hanno completato il loro aggiornamento procede a inviare il frame aggiornato alla GUI. Questo meccanismo implementa una barriera di sincronizzazione senza utilizzare primitive di sincronizzazione esplicite, sfruttando invece il modello di messaggistica degli attori.

L'integrazione con la GUI è gestita attraverso il GUIActor, che riceve frame completi dal Manager e aggiorna la visualizzazione. L'attore mantiene riferimenti ai componenti Swing come slider e pulsanti, ma tutte le modifiche alla GUI avvengono attraverso SwingUtilities.invokeLater per garantire thread-safety. Un ulteriore aspetto dell'implementazione è la gestione reattiva dei controlli dell'interfaccia. Quando l'utente modifica i parametri della simulazione attraverso gli slider, il GUIActor disabilita temporaneamente i controlli e invia un messaggio UpdateParams al Manager. Il Manager propaga questi nuovi parametri a tutti i BoidActor e, solo dopo aver ricevuto conferma dell'aggiornamento, riabilita i controlli nella GUI. Questo pattern previene race condition e garantisce che tutti i boids utilizzino parametri consistenti.

La visualizzazione effettiva è implementata nella classe BoidsPanel, che estende JPanel e si occupa del rendering dei boids. Questa classe mantiene una copia locale dello stato corrente e implementa il metodo paintComponent per disegnare ogni boid come un piccolo cerchio blu. La classe gestisce anche la trasformazione delle coordinate dal sistema di riferimento della simulazione (centrato nell'origine) al sistema di riferimento dello schermo. Il panel mostra inoltre informazioni di debug come il numero di boids e il framerate corrente.

L'InitialDialog fornisce un'interfaccia user-friendly per configurare la simulazione prima dell'avvio. Questa classe dimostra come integrare dialog modali Swing con il sistema di attori, bloccando l'inizializzazione fino a quando l'utente non ha fornito i parametri necessari. Il dialog valida l'input dell'utente e gestisce gracefully casi di input non valido.

Il modello dei dati è completato dalla classe BoidState, un record immutabile che rappresenta lo stato completo di un boid in un dato momento, e SimulationMetrics, che incapsula le metriche di performance della simulazione.

La gestione degli errori e la fault tolerance sono implementate attraverso supervisor strategies di Akka. Ogni attore è configurato con una strategia di restart che garantisce la continuità della simulazione anche in presenza di errori temporanei. Il sistema utilizza anche il pattern di stashing per gestire messaggi che arrivano durante fasi di inizializzazione o transizione di stato, garantendo che nessun messaggio venga perso.

Distributed Computing

L'implementazione distribuita di Agar.io richiede la gestione di stato condiviso attraverso multiple JVM potenzialmente distribuite su nodi fisici diversi. La soluzione adottata sfrutta le capacità di Akka Cluster per creare un sistema resiliente e scalabile.

Il punto di ingresso del sistema distribuito è gestito attraverso diverse classi Main specializzate. La classe principale Main nel package controller funge da dispatcher, permettendo di avviare il sistema in diverse modalità: server, client, AI, o una configurazione completamente personalizzata. Questa flessibilità facilita sia lo sviluppo che il deployment, permettendo di testare diverse configurazioni con facilità. Il Main implementa anche un'interfaccia grafica iniziale per la selezione del nome del giocatore, dimostrando come integrare Swing con un sistema distribuito Akka.

Il GameServer rappresenta il nodo server del cluster. Questa classe configura e avvia il sistema con il ruolo "server", inizializza i Cluster Singleton actors (WorldManager e FoodManager), e crea il GlobalViewActor per la visualizzazione amministrativa del gioco. Un aspetto importante dell'implementazione è l'uso di un shutdown hook per garantire la terminazione graceful del sistema, fondamentale in un ambiente distribuito per evitare stati inconsistenti.

Il GameClient gestisce i nodi client per giocatori umani. Ogni client si connette al cluster con il ruolo "client", ottiene un proxy al WorldManager singleton, e crea un PlayerActor locale per rappresentare il giocatore. L'implementazione gestisce la disconnessione, con timeout configurabili e gestione delle eccezioni durante la terminazione per garantire che il giocatore venga correttamente deregistrato dal gioco.

L'AIClient è una variante specializzata che può creare multipli giocatori AI su un singolo nodo. Questa classe dimostra la scalabilità del sistema, permettendo di popolare rapidamente il gioco con bot per testing o per migliorare l'esperienza quando ci sono pochi giocatori umani. Ogni AI è un PlayerActor completo con logica di movimento autonoma.

L'architettura si basa su una chiara separazione tra nodi server e nodi client. I nodi server ospitano i Cluster Singleton actors che gestiscono lo stato globale del gioco. Il GameClusterSupervisorActor è l'attore radice che supervisiona l'intero cluster. Questo attore si iscrive agli eventi del cluster per monitorare join, leave, e unreachability di nodi, loggando questi eventi per debugging e potenzialmente triggerando azioni di recovery. Il supervisore inizializza anche i singleton actors sui nodi con ruolo "server", garantendo che esistano solo sui nodi appropriati.

Il WorldManagerActor rappresenta il cuore del sistema distribuito. Implementato come Cluster Singleton, garantisce che esista una sola istanza attiva nel cluster in ogni momento, anche in presenza di fallimenti del nodo che lo ospita. Questo attore mantiene lo stato completo del mondo di gioco, includendo le posizioni di tutti i giocatori e del cibo. La sua implementazione gestisce la registrazione dinamica dei giocatori, l'aggiornamento delle posizioni, il rilevamento delle collisioni, e il broadcasting periodico dello stato a tutti i PlayerActor nel cluster.

L'implementazione del WorldManagerActor utilizza un pattern di messaggi interni per gestire operazioni asincrone complesse. Quando riceve una richiesta GetAllFood dal FoodManager, utilizza l'ask pattern per interrogare il FoodManagerActor e aggiorna lo stato interno quando riceve la risposta. Similarmente, utilizza il Receptionist per scoprire dinamicamente tutti i PlayerActor attivi e gestire la loro comunicazione. Un aspetto critico è la gestione del ciclo di vita: il WorldManager si

integra con CoordinatedShutdown di Akka per garantire che tutti i giocatori vengano notificati prima della terminazione del server.

La gestione delle collisioni nel WorldManagerActor è implementata attraverso il metodo checkCollisions, chiamato dopo ogni aggiornamento di posizione. Questo metodo verifica efficientemente sia le collisioni con il cibo che con altri giocatori utilizzando gli helper methods definiti in EatingManager. Quando viene rilevata una collisione, il Manager aggiorna immediatamente lo stato locale e genera gli eventi appropriati. Per le collisioni tra giocatori, il sistema utilizza il Receptionist per trovare l'ActorRef del giocatore "mangiato" e inviargli il messaggio PlayerDied, garantendo che il giocatore venga notificato anche se si trova su un nodo remoto.

Il FoodManagerActor, anch'esso implementato come Cluster Singleton, gestisce autonomamente la generazione e rimozione del cibo. L'implementazione utilizza un timer Akka per generare periodicamente nuovo cibo ogni due secondi, mantenendo un contatore globale per assegnare ID univoci. La classe mantiene una lista immutabile di tutto il cibo presente nel mondo e la aggiorna atomicamente in risposta ai messaggi RemoveFood dal WorldManager.

Un altro aspetto dell'implementazione è il meccanismo di broadcasting dello stato. Ogni 50 millisecondi, il WorldManager utilizza il Receptionist di Akka per scoprire dinamicamente tutti i PlayerActor attivi nel cluster e invia loro lo stato aggiornato del mondo. Questo approccio push-based garantisce che tutti i giocatori ricevano aggiornamenti consistenti e tempestivi. Il broadcasting è ottimizzato per inviare solo ai giocatori il cui PlayerActor corrisponde effettivamente a un giocatore ancora vivo nel mondo, evitando sprechi di banda.

I PlayerActor rappresentano i singoli giocatori, sia umani che AI. L'implementazione di PlayerActor è particolarmente sofisticata, gestendo multiple fasi del ciclo di vita attraverso diversi behavior states. Lo stato initializing gestisce la registrazione iniziale con retry automatici per i giocatori AI, fondamentale per la robustezza in un ambiente distribuito dove il WorldManager potrebbe non essere immediatamente disponibile. Una volta registrato, l'attore transita allo stato active dove gestisce movimento, aggiornamenti della vista, e potenziale morte del giocatore.

Per i giocatori AI, il PlayerActor implementa una logica di movimento autonoma attraverso un timer che triggera il comportamento StartAI ogni 100 millisecondi. L'AI utilizza la classe helper AIMovement per calcolare il cibo più vicino e muoversi verso di esso.

La gestione della morte e respawn è implementata attraverso lo stato waitingForRespawn. Quando un giocatore muore, il PlayerActor mostra un dialog di conferma per i giocatori umani o attende automaticamente 3 secondi per gli AI prima di tentare la re-registrazione. Questo design permette ai giocatori di rimanere nel gioco anche dopo la morte.

Il GlobalViewActor fornisce una vista amministrativa dell'intero mondo di gioco. Questo attore, che gira solo sui nodi server, interroga periodicamente il WorldManager per ottenere lo stato completo e lo visualizza attraverso GlobalViewFrame. L'implementazione utilizza un timer per richiedere aggiornamenti ogni 30 millisecondi e gestisce gracefully i timeout delle richieste, mostrando un mondo vuoto invece di crashare.

La visualizzazione del gioco è gestita attraverso due componenti principali. La DistributedLocalView fornisce la vista del giocatore singolo, mostrando solo la porzione di mondo visibile centrata sulla posizione del giocatore. Questa classe integra elegantemente Swing con il sistema di attori, ricevendo

aggiornamenti asincroni dal PlayerActor e traducendo i movimenti del mouse in comandi MoveDirection. Quando il giocatore muore, la vista si oscura e mostra un messaggio di attesa, mantenendo però la connessione con l'attore per gestire il respawn.

La GlobalViewFrame fornisce invece una vista completa del mondo per scopi amministrativi o di spettatore. Utilizza la stessa logica di rendering di DistributedLocalView ma senza offset della camera, mostrando l'intero mondo di gioco. Entrambe le view delegano il rendering effettivo ad AgarViewUtils, una classe di utilità che incapsula la logica di disegno comune.

AgarViewUtils implementa il rendering visuale del mondo di gioco. Questa classe gestisce la trasformazione delle coordinate dal sistema di riferimento del gioco a quello dello schermo, disegna il cibo come cerchi verdi e i giocatori con colori distintivi basati sul loro ID.

Il modello dei dati del gioco è strutturato attraverso una gerarchia di trait e case class Scala. Il trait Entity definisce l'interfaccia comune per tutti gli oggetti del gioco, fornendo proprietà base come posizione, massa, e raggio calcolato dinamicamente dalla massa. Le case class Player e Food estendono Entity aggiungendo comportamenti specifici. Player include il metodo grow per aumentare la massa quando consuma altre entità, mentre Food è essenzialmente un Entity con massa fissa.

La classe World rappresenta lo stato completo del mondo di gioco in un dato momento. Implementata come case class immutabile, fornisce metodi funzionali per aggiornare lo stato senza mutazione. I metodi updatePlayer e removePlayers creano nuove istanze del mondo con le modifiche appropriate, seguendo i principi della programmazione funzionale. Il metodo playerById fornisce accesso efficiente a specifici giocatori, mentre playersExcludingSelf è un'utility comune per calcoli che escludono il giocatore corrente.

Il sistema di messaggi è definito attraverso una gerarchia di trait che garantiscono type-safety nella comunicazione tra attori. Il trait Messages funge da marker interface per la serializzazione, mentre GameMessage, WorldManagerMessage, FoodManagerMessage, e PlayerActorMessage definiscono categorie specifiche di messaggi. Ogni messaggio è implementato come case class o case object, fornendo automaticamente equality, pattern matching, e serializzazione.

La configurazione del sistema è centralizzata nel file utils.scala e nell'oggetto GameConfig, che definisce tutte le costanti temporali (durate dei timer) e spaziali (dimensioni del mondo, velocità, masse) del gioco. Questa centralizzazione facilita il tuning del gameplay e garantisce consistenza attraverso tutti i componenti. La funzione startupWithRole incapsula la complessa configurazione di Akka Cluster, permettendo di creare facilmente nodi con ruoli e porte specifiche.

Rock-Paper-Scissors in Go - Message Passing Sincrono

La soluzione sfrutta le goroutine e i canali di Go per creare un sistema di comunicazione completamente sincrono senza necessità di lock o altre primitive di sincronizzazione esplicite.

Il design del sistema si basa su tre componenti principali che comunicano attraverso un protocollo ben definito. L'arbitro (referee) coordina il gioco attraverso un loop infinito dove, ad ogni iterazione, richiede le mosse ai due giocatori, le confronta, determina il vincitore, e comunica i risultati. Questa

implementazione utilizza un pattern di comunicazione: invece di utilizzare canali separati per richieste e risposte, ogni richiesta di mossa include un canale su cui il giocatore invierà la sua risposta.

I giocatori sono implementati come goroutine che attendono continuamente richieste dall'arbitro. Quando ricevono una richiesta (sotto forma di un canale su cui inviare la loro mossa), generano una mossa casuale, la impacchettano in una struttura Move che include anche un canale per ricevere il risultato, e la inviano all'arbitro. Questo design elimina completamente la necessità di stato condiviso: ogni giocatore mantiene localmente il proprio punteggio.

La struttura Move utilizzata per la comunicazione è particolarmente interessante perché incapsula non solo la mossa del giocatore, ma anche il canale su cui il giocatore si aspetta di ricevere il risultato. Questo pattern di "risposta con canale di callback" è un idioma comune in Go che permette comunicazione bidirezionale strutturata senza complessità aggiuntiva.

La sincronizzazione perfetta del gioco è garantita dalla natura bloccante delle operazioni sui canali. L'arbitro non procede finché non ha ricevuto le mosse da entrambi i giocatori, e i giocatori non possono procedere finché non hanno ricevuto il risultato dal'arbitro. Questo elimina completamente la possibilità di race condition o stati inconsistenti.

L'implementazione dimostra anche come il modello CSP di Go permetta di esprimere naturalmente protocolli di comunicazione complessi. Il fatto che i canali siano valori di prima classe nel linguaggio permette di passarli come parametri, includerli in strutture dati, e creare pattern di comunicazione dinamici e flessibili.