Asyncronous Message Passing with Actors

(not distributed)

Autore: Sajmir Buzi

Anno Accademico 2025

Contents

1 Introduzione

Il progetto nella cartella ex1 implementa una simulazione interattiva del comportamento di stormo, ispirata al modello di boids di Craig Reynolds. L'obiettivo principale è riprodurre il movimento collettivo di un gruppo di agenti artificiali (boid) in uno spazio bidimensionale, dove ogni agente segue semplici regole comportamentali. La simulazione fornisce un'interfaccia utente grafica (GUI) che consente agli utenti di configurare i parametri, avviare, mettere in pausa, riprendere e interrompere la simulazione in tempo reale.
Per garantire efficienza e reattività, soprattutto quando si simula un numero elevato di boids, il sistema sfrutta tecniche di programmazione concorrente. Il modello di attore viene adottato per strutturare la logica di simulazione, consentendo un parallelismo sicuro e scalabile. Questo approccio consente alla simulazione di rimanere interattiva e reattiva ai comandi dell'utente, anche sotto un carico computazionale elevato.

2 Analisi del Problema

L'obiettivo del progetto è simulare il comportamento di stormo di boidi (uccelli artificiali) in uno spazio 2D. Ogni boide si muove secondo semplici regole (coesione, separazione, allineamento) e interagisce con gli altri boidi nelle vicinanze. La simulazione deve essere interattiva, consentendo all'utente di avviare, mettere in pausa, riprendere, interrompere e configurare i parametri in fase di esecuzione.

Aspetti di concorrenza:

- La simulazione coinvolge un gran numero di boidi, ognuno dei quali deve aggiornare il proprio stato in base alle posizioni e alle velocità degli altri.
- Per gestire in modo efficiente il calcolo e mantenere la reattività dell'interfaccia utente grafica (GUI), il sistema adotta un modello concorrente in cui ogni boide è rappresentato come un attore indipendente
- Gli attori comunicano in modo asincrono, evitando stati mutevoli condivisi e condizioni di competizione. L'interfaccia utente grafica (GUI) viene eseguita sullo Swing

Event Dispatch Thread, mentre la logica di simulazione e gli aggiornamenti dei boidi sono gestiti dagli attori Akka, garantendo che i vari calcoli a lunga esecuzione non blocchino l'interfaccia utente.

3 Architettura e Design

Il cuore della simulazione è costruito attorno all'architettura ad attori fornita da **Akka Typed**, un modello concorrente che consente la gestione di entità indipendenti e scalabili attraverso la comunicazione asincrona. In questo contesto, ogni *boid* è rappresentato da un attore autonomo, coordinato da un attore centrale responsabile dell'intera simulazione. L'interfaccia grafica è anch'essa integrata nel modello attoriale, garantendo un'interazione fluida e thread-safe con il sistema. Di seguito si descrivono i principali attori coinvolti e le rispettive responsabilità.

3.1 Actor

3.1.1 BoidActor

Ogni `BoidActor` rappresenta un singolo boid della simulazione e viene gestito in modo completamente indipendente. Questo approccio consente un'elevata parallelizzazione, poiché ogni boid può aggiornare il proprio stato in autonomia, sfruttando il modello ad attori per evitare problemi di concorrenza.

Responsabilità principali:

- Mantenere lo stato locale, costituito da posizione (`P2d`), velocità (`V2d`) e parametri specifici di comportamento (`BoidsParams`).
- Ricevere messaggi di aggiornamento (`UpdateRequest`) dal `ManagerActor`, contenenti lo stato globale degli altri boid.
- Calcolare la nuova posizione e velocità applicando le classiche regole del *flocking*:
  - **Allineamento:** orientamento verso la direzione media dei boid vicini.
  - **Coesione:** movimento verso il centro di massa dei vicini.
  - **Separazione:** allontanamento da boid troppo prossimi.

- Applicare limiti alla velocità e alla posizione, garantendo che il boid rimanga all'interno dello spazio simulato (attraverso la funzione `wrap()`).
- Comunicare il nuovo stato al manager con il messaggio `BoidUpdated`.
- Adattarsi dinamicamente a variazioni nei parametri, rispondendo ai messaggi `UpdateParams`.

Funzionalità e metodi principali:

- `behavior()`: implementa la logica di reazione ai messaggi.
- `onUpdateRequest()`: aggiorna lo stato del boid in base allo stato globale ricevuto.
- `avgPosition()`, `avgVector()`, `avgSeparation()`: funzioni di supporto per il calcolo delle regole di flocking.
- `wrap()`: funzione di confinamento nello spazio simulato.

Aspetti concorrenti: L'adozione di un attore per ciascun boid elimina il rischio di condizioni di gara, garantendo la sicurezza dell'accesso ai dati. La comunicazione avviene esclusivamente tramite scambio di messaggi, favorendo l'indipendenza e la scalabilità del sistema.

3.1.2 ManagerActor

Il `ManagerActor` è il nodo centrale della simulazione, responsabile della creazione, gestione e coordinamento di tutti gli attori boid, oltre a fungere da ponte tra la simulazione e l'interfaccia grafica.

Responsabilità principali:

- Ricevere i comandi dalla UI attraverso messaggi del protocollo `ManagerProtocol.Command`.
- Inizializzare e gestire gli attori `BoidActor`, mantenendo la lista dei riferimenti e lo stato globale.
- Gestire il ciclo di aggiornamento: inviare periodicamente richieste (`UpdateRequest`) ai boid e raccoglierne le risposte (`BoidUpdated`).
- Controllare il flusso della simulazione: avvio, pausa, ripresa e arresto.

- Inviare aggiornamenti alla UI, tra cui lo stato aggregato della simulazione (`RenderFrame`) e eventuali messaggi di stato (`UpdateStatus`).

Metodi principali:

- `behavior()`: logica di gestione dei messaggi ricevuti.
- `onStartSimulation()`, `onPauseSimulation()`, `onResumeSimulation()`, `onStopSimulation()`: controllo del ciclo di vita della simulazione.
- `onBoidUpdated()`: raccolta e aggregazione dei nuovi stati calcolati dai boid.

Aspetti concorrenti: Il manager agisce come orchestratore, garantendo la sincronizzazione delle informazioni tra gli attori e mantenendo la consistenza dell'intero sistema. Questo ruolo centrale evita conflitti e rende controllabile l'evoluzione dello stato globale.

3.1.3 SimulationUIActor

L'attore `SimulationUIActor` rappresenta il punto di contatto tra l'utente e la simulazione. È incaricato di gestire la GUI e di inoltrare i comandi ricevuti ai componenti interni del sistema.

Responsabilità principali:

- Inizializzare e mantenere l'interfaccia grafica principale (`BoidsGUI`).
- Aggiornare graficamente lo stato della simulazione sulla base dei messaggi ricevuti dal manager.
- Ricevere comandi utente (start, pausa, resume, stop, modifica parametri) e trasmetterli al `ManagerActor`.
- Gestire lo stato dei pulsanti in modo thread-safe tramite `SwingUtilities.invokeLater`.

Metodi principali:

- `createMainGUI()`: costruisce la finestra principale o la rende visibile.
- `onStartSimulation()`, `onPauseResume()`, `onStop()`: callback associati agli eventi dell'interfaccia.

- `updateButtonStates()`: aggiorna dinamicamente i controlli in base allo stato corrente.

Aspetti concorrenti: Tutte le operazioni relative alla GUI avvengono nel thread dedicato a Swing, evitando accessi simultanei. L'interfaccia non interroga mai direttamente il modello, ma si limita a ricevere notifiche, garantendo una separazione chiara e sicura tra logica e presentazione.

3.2 Model

Il componente `model` racchiude tutte le classi responsabili della rappresentazione del dominio della simulazione, ovvero lo stato individuale dei boid, i parametri di configurazione e le metriche statistiche. Queste classi non sono attori, ma vengono utilizzate dagli attori per comunicare e condividere informazioni in modo strutturato e sicuro.

3.2.1 BoidState

La classe `BoidState` rappresenta lo stato istantaneo di un singolo boid all'interno dello spazio simulato. Tale informazione viene frequentemente utilizzata per scambiare dati tra attori, in particolare tra il `ManagerActor` e i singoli `BoidActor`.

Campi principali:

- `P2d position`: la posizione corrente del boid nel piano bidimensionale.
- `V2d velocity`: il vettore velocità attuale.
- `int id`: identificatore univoco del boid.

Ruolo: Funge da contenitore per lo scambio di stato tra attori. Ogni boid aggiorna il proprio stato locale a partire da una lista di istanze `BoidState` ricevute dal manager. Ciò consente una comunicazione immutabile e sicura nel contesto concorrente.

Metodi: Fornisce metodi di accesso (*getter*) per ciascun campo: posizione, velocità e identificativo.

3.2.2 BoidsParams

La classe `BoidsParams` incapsula tutti i parametri che controllano il comportamento della simulazione. Tali parametri possono essere configurati dinamicamente per osservare l'effetto di diverse impostazioni sulle dinamiche del sistema.

Campi principali:

- Numero di boid presenti nella simulazione.
- Dimensioni dello spazio simulato (larghezza e altezza).
- Pesi per le tre regole fondamentali: allineamento, coesione, separazione.
- Velocità massima consentita per ciascun boid.
- Raggio di percezione: distanza entro cui un boid considera gli altri vicini.
- Raggio di evitamento: distanza minima da mantenere per evitare sovrapposizioni.
- Limiti spaziali entro cui confinare il movimento dei boid.

Ruolo: Permette di controllare e regolare in tempo reale il comportamento emergente del sistema, adattando i pesi delle regole o i vincoli spaziali. È una componente fondamentale per esperimenti parametrici o simulazioni interattive.

Metodi: Include metodi *getter* e *setter* per ogni parametro, consentendo una gestione flessibile da parte della UI e del manager.

3.3 BoidsMetrics

La classe `BoidsMetrics` è dedicata alla raccolta di dati statistici sulla simulazione in corso. I dati raccolti vengono aggiornati periodicamente e possono essere visualizzati in tempo reale dall'interfaccia utente.

Campi principali:

- Numero di **tick** (unità temporali simulate) trascorsi.
- Tempo effettivo di simulazione (in millisecondi o secondi).
- Valori medi calcolati dinamicamente, come la velocità media dei boid.

- Eventuali altri indicatori prestazionali o comportamentali.

Ruolo: Questa classe è utile per monitorare le performance della simulazione, valutare l'efficacia dei parametri impostati e fornire feedback all'utente tramite grafici o etichette all'interno della GUI.

Metodi: Fornisce metodi di accesso e aggiornamento per ogni statistica, consentendo la raccolta incrementale dei dati nel tempo.

3.4 Protocols

Nel modello ad attori adottato, la comunicazione tra componenti avviene esclusivamente attraverso lo scambio di messaggi tipizzati. Per garantire coerenza, chiarezza e sicurezza nella comunicazione, sono stati definiti protocolli specifici sotto forma di interfacce o sealed trait, ciascuno dei quali raccoglie i messaggi validi tra coppie di attori. In questa sezione vengono descritti i tre protocolli principali che regolano la comunicazione tra UI, Manager e Boid.

3.4.1 ManagerProtocol

Il file `ManagerProtocol.java` definisce il set di messaggi che possono essere scambiati tra l'interfaccia utente (`SimulationUIActor`) e l'attore centrale della simulazione (`ManagerActor`). È il punto di ingresso principale per il controllo del ciclo di vita della simulazione.

Messaggi di comando (dalla UI al Manager):

- `StartSimulation`: avvia una nuova simulazione.
- `PauseSimulation`: sospende temporaneamente l'evoluzione dei boid.
- `ResumeSimulation`: riavvia una simulazione in pausa.
- `StopSimulation`: termina e smantella la simulazione.
- `UpdateParams`: invia nuovi parametri di configurazione (es. pesi delle regole).

Messaggi di risposta (dal Manager alla UI):

- `BoidUpdated`: notifica lo stato aggiornato di un boid (usato internamente).

- `SimulationStatus`: comunica lo stato attuale della simulazione (es. attiva, in pausa, terminata).

Ruolo del protocollo: Questo protocollo definisce in maniera chiara e tipizzata l'interfaccia tra l'utente e il motore della simulazione. Garantisce che ogni comando venga elaborato in modo sicuro, evitando ambiguità o errori di tipo nei messaggi.

3.4.2 BoidProtocol

Il file `BoidProtocol.java` contiene i messaggi utilizzati nella comunicazione tra il `ManagerActor` e i singoli `BoidActor`. Si tratta del protocollo fondamentale che consente la propagazione dello stato globale e la sincronizzazione tra attori.

Messaggi principali:

- `UpdateRequest`: contiene lo stato globale della simulazione e viene inviato dal manager a ogni boid per calcolare il nuovo stato locale.
- `UpdateParams`: invia a un boid l'aggiornamento dei parametri di simulazione (in genere condivisi tra tutti).
- `WaitUpdateRequest`: usato in scenari dove è necessario sincronizzare il ciclo di aggiornamento, forzando un'attesa prima del calcolo.

Ruolo del protocollo: Fornisce il canale formale per implementare un comportamento distribuito ma coerente tra i boid. Consente l'evoluzione parallela e indipendente di ciascun attore boid, mantenendo l'integrità del sistema grazie alla sincronizzazione gestita centralmente.

3.4.3 GUIProtocol

Il file `GUIProtocol.java` definisce i messaggi che regolano la comunicazione tra il `ManagerActor` e l'interfaccia grafica (`SimulationUIActor`). Tali messaggi consentono alla GUI di aggiornare dinamicamente la visualizzazione e lo stato dei controlli.

Messaggi principali:

- `RenderFrame`: contiene l'elenco aggiornato degli stati dei boid da visualizzare nella GUI.

- `UpdateWeights`: comunica alla UI i valori aggiornati dei pesi delle regole, ad esempio in seguito a modifiche.
- `UpdateStatus`: aggiorna la GUI con lo stato attuale della simulazione (es. running, paused, stopped).

Attraverso l'uso dei protocolli si garantisce che l'interfaccia grafica possa riflettere in modo corretto e reattivo lo stato della simulazione, senza accedere direttamente ai dati interni. Tutti i messaggi ricevuti sono gestiti nel thread Swing attraverso chiamate asincrone sicure, prevenendo problemi di concorrenza nella GUI.

4 Conclusioni

Per concludere il discorso il design di questo progetto nasce dall'idea di rendere la simulazione dei boid non solo efficiente e scalabile, ma anche semplice da capire e da estendere. Si è scelto di suddividere il codice in diverse cartelle, ognuna con una responsabilità ben precisa, proprio per facilitare la manutenzione.

La parte centrale della simulazione è gestita dagli attori: ogni boid è un piccolo "robot" indipendente che pensa e si muove per conto suo, mentre un manager tiene le fila della situazione, coordina gli aggiornamenti e si assicura che tutto funzioni come previsto. Questo approccio, basato sul modello Actor, ci permette di sfruttare la concorrenza in modo naturale: ogni boid può calcolare la propria posizione in parallelo agli altri, senza rischiare di "pestarsi i piedi".

I modelli raccolgono tutte le informazioni e i parametri necessari per la simulazione. Qui troviamo la rappresentazione dello stato di ogni boid, i parametri che regolano il loro comportamento e le metriche che ci aiutano a capire come sta andando la simulazione.

Il protocollo di comunicazione è come il linguaggio che gli attori usano per parlarsi: ogni messaggio è ben definito e tipizzato, così non ci sono fraintendimenti. Questo rende la comunicazione tra le varie parti del sistema sicura e chiara.

La view si occupa di tutto ciò che riguarda l'interfaccia grafica. Qui l'utente può vedere la simulazione in tempo reale, interagire con i controlli, cambiare i parametri e osservare come i boid si comportano. L'aggiornamento della grafica avviene sempre in modo reattivo e fluido, grazie all'integrazione con il thread Swing.

Infine, la cartella utils raccoglie tutte le funzioni di supporto e le classi matematiche che servono a semplificare i calcoli e a mantenere il codice pulito.

In sintesi abbiamo visto questo esercizio nelle vaire fasi sia con i thread che con i messaggi, questo design punta a essere robusto, modulare e facilmente estendibile. Ogni parte del sistema sa esattamente cosa deve fare e comunica con le altre solo quando serve. Il risultato è una simulazione che funziona bene anche con molti boid, che può essere facilmente modificata o ampliata, e che offre all'utente un'esperienza fluida e piacevole.