Report Assignment-03
Concurrent and Distributed Programming

Muhamad Huseyn Salman

June 16, 2025

Contents

1 Local Actors

The main challenges involved effectively leveraging actor concurrency and message passing while avoiding shared memory, side effects, and race conditions. Additionally, multiple actors needed to remain responsive to GUI events and requiring synchronization.

1.1 Design

In the current design, each boid is managed by a dedicated actor responsible for updating its state at every time frame. A central Manager actor initiates each update cycle by broadcasting a message to all boid actors, sending the current list of boids. It then collects the updated states from each actor and forwards the new boid list to the view for rendering.

To avoid race conditions and that boids sees the side-effects of other boid actors in the same iteration, the calculation phase and the update phase have been divided. During the calculation phase, each actor accesses the same list, and then calculates the new state of its boid independently, without interference.The coordination between the two phases is guaranteed by the Manager actor, who ensures that all actors complete the calculation before proceeding with the update. Once all the new states have been collected, the manager replaces the previous list with the updated one and notifies its updated references to each actor

1.2 Message Protocol

The boid message protocol comprises a set of messages divided into different categories, based on their functionality.

Simulation Workflow Messages

1. **StartUpdate(List boids)**: Sent to a Boid actor to begin its update cycle. Includes the current list of all boids.
2. **CalculateVelocity(List boids)**: The manager sends the message to all boid actor to start computing the update.
3. **VelocityCalculated()**: Sent by a Boid actor to the manager after calculating the new state.
4. **UpdateBoid()**: The manager sends the message to all boid actors to update the new computed state to each boid.

5. **BoidUpdated()**: Sent by a Boid actor to the manager after completing the update.
6. **ContinueSimulation()**: The manager sends this message to itself to start the next iteration of the simulation loop.
7. **UpdateView(BoidsModel model, int framerate)** Sent to the GUI component to update the visual representation using the current model and framerate.

Simulation Control Messages

1. **BootSimulation(BoidsModel model)** Initializes the simulation, creating all necessary Boid actors.
2. **StartSimulation()** Starts the simulation loop, transitioning the system to an active running state.
3. **StopSimulation()** Stops the simulation, pausing all boid updates.
4. **ResetSimulation(List¡Boid¿ boids)** Resets the simulation state with a new list of boids.

Model Parameter Update Messages

1. **SetSeparationWeight(double weight)** Sets the weight for the separation rule.
2. **SetAlignmentWeight(double weight)** Sets the weight for the alignment rule.
3. **SetCohesionWeight(double weight)** Sets the weight for the cohesion rule.

1.3 Manager Actor Behaviors

To orchestrate the boid actors, run the simulation loop and remain reactive to GUI events, the manager actor switches between several distinct behaviors in order to manage only some message types and stash the others.

1. **Boot**: initial behavior responsible for bootstrapping the simulation creating all the necessary Boid actors.

2. **Update**: the manager either starts a new update cycle (by sending update messages to the boids) or reacts to user events from the GUI. A message is also sent to itself in advance to continue the simulation, but this is handled after completing the current update phase.

3. **Collect**: the manager carries on the two update phases and gathers updated states from all Boid actors and stashes all messages from the GUI (therefore it's not reactive). Once all updates are received, it refreshes the GUI with the new model state.

4. **Stopped Simulation** the manager stays reactive to GUI events and allows resetting the simulation state, but no updates are performed until resumed.

Here is the finite state machine representing how the manager switches between the different behaviors.

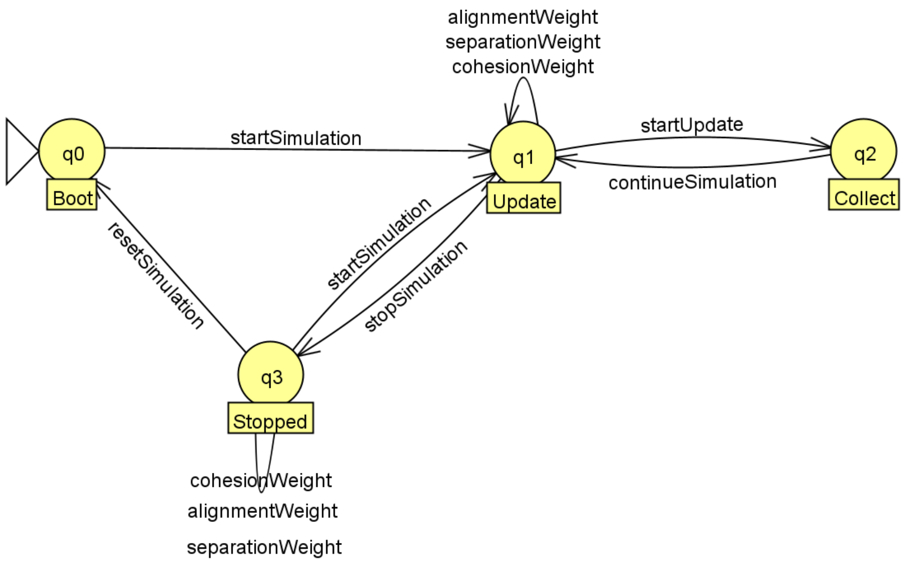

Figure 1: Manager Finite State Machine

2 Distributed Actors

2.1 Design

The system has divided the functionalities and the responsibilities between different actors:

1. **WorldActor**: maintains the World object, receives movements from players, and broadcasts the updated version of the world. It also receives new food from the FoodGeneratorActor and handles players joining and leaving the game.

2. **PlayerActor** It registrates to the game, forwards user inputs to the WorldActor, and receives the updated version of the world, which is then rendered.

3. **AIplayerActor** Represents an AI player, which joins the game through and sends its movements periodically.

4. **FoodGeneratorActor** It spawns new food periodically and sends it to the WorldActor which adds it to the world.

5. **GlobalViewActor** It renders the full world for a global view, and periodically polls the WorldActor to retrieve the updated world.

6. **GameOverActor** It polls periodically polls the world to checking the game over condition on player mass, and communicating the winner when there is one.

Actors discover the `WorldActor` dynamically through Akka `Receptionist`. All world state updates and player movements flow through the WorldActor, which ensures consistency. Both the AIplayers and the FoodGenerator use internal timers to perform periodic actions independently. The `WorldActor` doesn't use a timer, it just updates the world each time it receives a message.
Once the WorldActor is created, the game is automatically started. Once one of the players exceeds a certain mass threshold, the game is stopped and the winner is communicated to all players, which show a prompt message.
The GlobalViewActor and the GameOverActor use an internal timer to poll periodically the World to operate on an updated copy.

The `WorldActor` works as a synchronization point, although many responsibilities are delegated to other actors, like the `FoodGeneratorActor, AIplayerActor and GlobalViewActor`. This synchronization point is necessary to ensure strong consistencies between participants as required. Implementing a fully decentralized solution in which each player has its own copy of the world and sends its moves to all the other players could have caused discrepancies and inconsistencies.

2.2 Message Protocol

World Messages

- `PlayerMove(id: String, dx: Double, dy: Double)`
  Sent by a player (human or AI) to move in a specified direction.
- `FoodGenerated(food: Food)`
  Sent by the `FoodGeneratorActor` to notify the WorldActor of a newly spawned food item.
- `JoinPlayer(id: String, replyTo: ActorRef[PlayerCommand])`
  Used by a player to join the game. Includes a reference to send updates back to the player.
- `LeavePlayer(id: String)`
  Informs the WorldActor that a player has left the game, which then removes it.
- `TriggerGameOver(winner: String)`
  Triggers the game over, sent by the GameOverActor to the world, used to stop the game and propagate the winner ID.
- `RequestWorld(replyTo: ActorRef[WorldResponse])`
  Requests the current state of the world. Typically used by the global view.
- `WorldResponse(world: World)`
  The reply to a `RequestWorld`, containing the current game state.

FoodGenerator Messages

- `GenerateFood`
  Internal timer-based message that periodically triggers food generation.
- `WorldActorListing(listings: Set[ActorRef[WorldActorCommand]])`
  Used to discover the WorldActor through the Akka Receptionist.

Player Messages

- `WorldUpdate(world: World)`
  Sent by the WorldActor to update the player with the current world state.

- `StartGame`
  Optional message to indicate that the game has started.
- `GameOver(winner: String)`
  Sent to notify players that the game has ended and who the winner is.
- `WorldActorListing(listings: Set[ActorRef[WorldActorCommand]])`
  Used for service discovery, inherited from the `ReceptionistListingMessage`.

AIPlayer Messages

- `Tick`
  A periodic message used to trigger AI behavior updates (e.g., pathfinding, decisions).
- `WorldActorListing(listings: Set[ActorRef[WorldActorCommand]])`
  Inherited message used to discover the WorldActor dynamically.

Receptionist Listing Messages

- `WorldActorListing(listings: Set[ActorRef[WorldActorCommand]])`
  Common listing message used by all components to subscribe and discover the WorldActor.