Relazione PCD-Assignment-3

Studente: Manuel Buizo
Corso: Programmazione Concorrente e Distribuita
Anno Accademico: 2023/2024

Indice

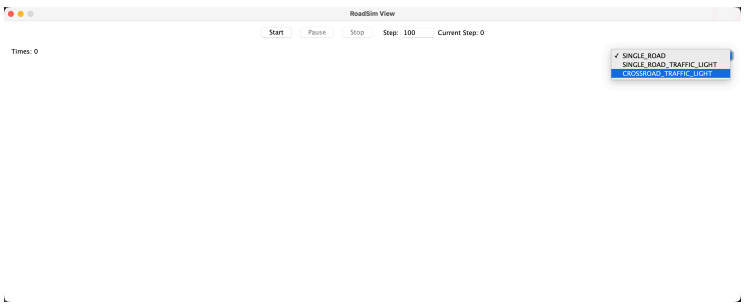
Introduzione

La presente relazione descrive quattro progetti sviluppati nell'ambito dell'esame di **[Programmazione Concorrente e Distribuita]**.
Ciascun progetto è stato sviluppato con l'obiettivo di analizzare un problema specifico e progettare una soluzione software appropriata, sfruttando le tecnologie trattate durante le lezioni.

Breve panoramica dei progetti:

1. **[Cars Simulation]** – Simulazione multi-agente di automobili basata sul paradigma ad attori con Akka.
2. **[Sudoku MOM]** – Implementazione distribuita di Sudoku tramite Message-Oriented Middleware (MOM) e scambio asincrono di messaggi.
3. **[Sudoku Java-RMI]** – Sviluppo di Sudoku con Distributed Object Computing e comunicazione remota tramite Java RMI.
4. **[Guess The Number]** – Gioco Guess the Number basato su scambio sincrono di messaggi, implementato in Go.

Progetto 1 – Cars Simulation



Analisi del problema

Partendo dall'Assignment 1 relativo alla simulazione delle macchine, il progetto estende la gestione del sistema adottando il paradigma ad attori tramite il framework Akka in Scala. In tale modello, ogni automobile è rappresentata da un attore autonomo, responsabile della gestione del proprio stato, del comportamento e delle interazioni con l'ambiente e con gli altri attori. La simulazione complessiva è coordinata attraverso la comunicazione asincrona tra attori, consentendo una gestione concorrente e scalabile del sistema.

Punti critici:

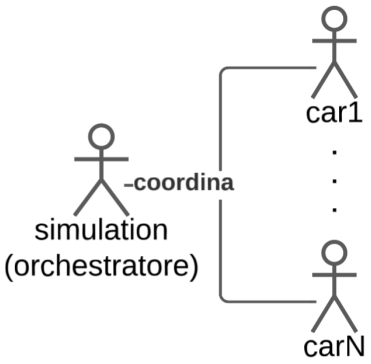
- Gestione del muro per la sincronizzazione e il coordinamento delle automobili durante la simulazione.
- Gestione degli step temporali e dello stato di ciascuna automobile a ogni iterazione, mediante l'utilizzo di attori Akka.

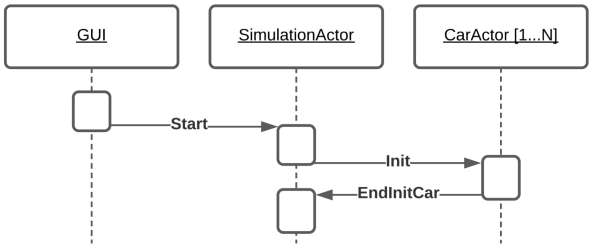
- Gestione della comunicazione tramite messaggi per l'integrazione e l'interazione con l'interfaccia grafica (GUI).

Architettura proposta

La simulazione è implementata utilizzando il framework Akka, che consente di modellare il sistema secondo il paradigma actor-based. Tale approccio facilita la gestione concorrente e distribuita dei processi, permettendo di rappresentare in modo naturale gli elementi della simulazione come attori indipendenti che comunicano tramite scambio di messaggi.

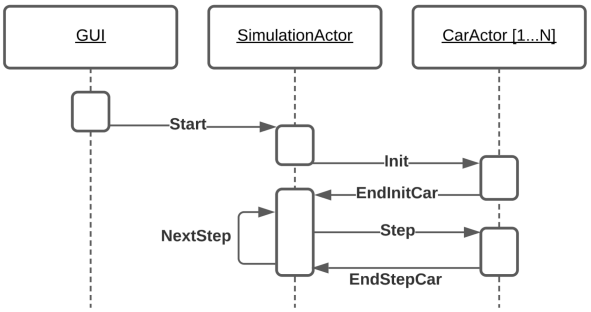
L'architettura si basa su un attore principale che funge da orchestratore e gestisce un insieme di attori, ciascuno dei quali rappresenta un'automobile. L'attore principale è responsabile dell'avanzamento temporale della simulazione, che procede a passi discreti di durata Δt. È inoltre possibile abilitare una temporizzazione basata sul tempo reale (analoga al frame rate nei videogiochi) per mantenere una sincronizzazione costante durante l'esecuzione.

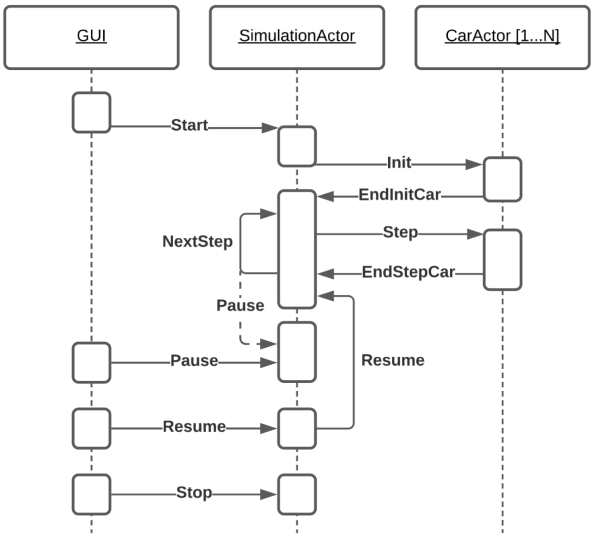




Prima dell'avvio della simulazione, viene eseguita una fase di setup iniziale, durante la quale l'attore principale coordina l'inizializzazione dei dati e delle condizioni di partenza di ciascun attore automobile.

Ogni automobile è modellata come un attore autonomo, incaricato di gestire il proprio stato interno ed eseguire le tre fasi del proprio comportamento: sense, decide e act. Durante ciascun passo di simulazione, le automobili eseguono queste fasi in modo concorrente, e quindi non deterministico rispetto all'ordine con cui gli attori vengono schedulati. Al termine della propria esecuzione, ogni attore automobile invia un messaggio di completamento all'attore principale. In questo modo, per ogni step della simulazione, l'attore principale attende che tutte le automobili abbiano completato la propria azione e aggiornato il rispettivo stato prima di procedere con lo step successivo.





Infine, l'attore della simulazione riceve anche messaggi provenienti dalla GUI, che consentono all'utente di interagire con l'esecuzione. In particolare, la GUI fornisce controlli per gestire gli stati della simulazione — come start, pause, resume e stop — oltre a un campo di testo (text field) che permette di specificare il numero totale di step che la simulazione deve eseguire. Queste interazioni vengono gestite attraverso messaggi inviati all'attore principale, che aggiorna di conseguenza il flusso di esecuzione.

Tecnologie utilizzate

Componente	Tecnologia	Ruolo
Frontend	Java.Swing	Interfaccia utente
Backend	Java + Scala.Akka	Logica applicativa

Sviluppo

È stata sviluppata una classe `Engine` dedicata alla gestione del tempo e degli step della simulazione. Essa consente inoltre di calcolare diverse statistiche temporali, come il tempo trascorso, la durata complessiva della simulazione, e include funzionalità per la gestione della pausa e la sincronizzazione temporale tramite il calcolo del delay. La struttura dell'Engine è stata realizzata interamente secondo il paradigma funzionale.

```
trait Engine extends Scheduler, Stepper:
  val startTime: Long
  val endTime: Long
  val allTimeSpent: Long
  val isInPause: Boolean

  override def start(): Engine
  def pause(): Engine
  def resume(): Engine
  override def stop(): Engine
  override def nextStep(): Engine
  override def setTotalSteps(value: Int): Engine
  def averageTimeForStep(): Double
  def timeElapsedSinceStart(): Long
```

`SimulationActor` è l'attore della simulazione. Per la sua implementazione è stato utilizzato il framework Akka insieme al linguaggio Scala. Sono stati definiti i messaggi attraverso cui è possibile interagire con la simulazione, inclusi i messaggi di controllo — start, pause, resume e stop. Inoltre, sono stati definiti i messaggi necessari alla fase di setup, utili a inizializzare le automobili e a verificare che tutte abbiano completato la propria inizializzazione. È stato infine definito il messaggio relativo agli step della simulazione, che permette di attivare le varie automobili; tale messaggio viene inviato nuovamente una volta che tutte le automobili hanno terminato lo step corrente e la simulazione si interrompe al raggiungimento del numero di step desiderato.

```
object SimulationActor:

  sealed trait Command

  case class Start(totalStep: Int) extends Command
  object Stop extends Command
  object Pause extends Command
  object Resume extends Command
  private object NextStep extends Command
  object EndInitCar extends Command
  case class EndStepCar(carAgent: CarAgent) extends Command
```

Come per l'attore della simulazione, anche per l'attore dell'automobile è stato utilizzato il framework Akka. Dentro la classe `CarActor` stati definiti i possibili messaggi che ogni automobile può ricevere, tra cui il messaggio di inizializzazione e quello per l'esecuzione degli step (sense, decide e act), al termine dei quali l'automobile notifica alla simulazione la conclusione dello step.

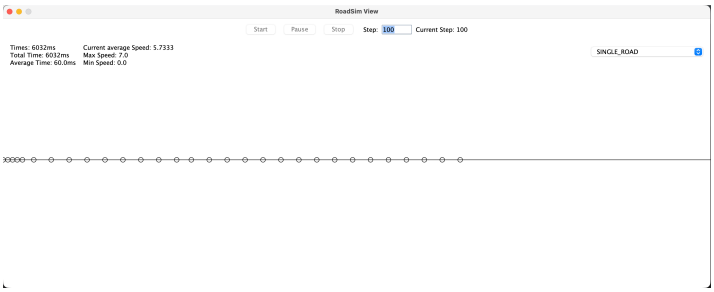
```
object CarActor:

  sealed trait Command

  case class Init(actor: ActorRef[SimulationActor.Command], simulation: AbstractSimulation) extends Command
  case class Step(actor: ActorRef[SimulationActor.Command], simulation: AbstractSimulation) extends Command
```

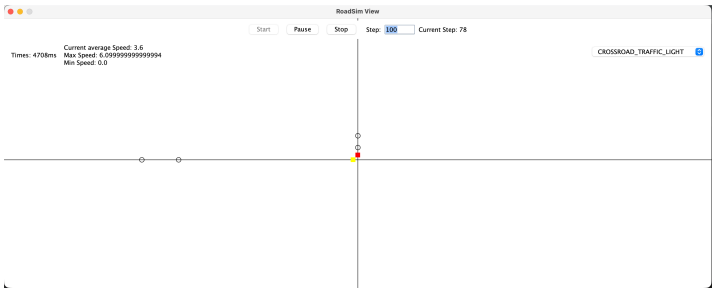
Risultati e considerazioni

In sintensi si è esplorato il framework akka, lavorando con lo scambio di messaggi asincrono in modo da sincronizzare le varie automobili tra di loro e con la simulazione. Per riutilizzare interamente il progetto per precedente assignment son state classi in scala che definiscono in modo separato i comportamenti dei vari attori, incapsulando interamente la logica della simulazione e delle automobili. Sono state testate i vari environment sia una semplice strada dritta fino ad arrivare agli incroci con i semafori.

In sintesi, è stato esplorato il framework Akka, lavorando con lo scambio di messaggi asincroni per sincronizzare le varie automobili tra loro e con la simulazione. Per riutilizzare interamente il progetto del precedente assignment sono state create classi in Scala che definiscono separatamente i

comportamenti dei diversi attori, incapsulando la logica della simulazione e delle automobili. Questo incapsulamento permette di scalare il comportamento degli attori utilizzando anche un linguaggio diverso, senza perdere efficienza e mantenendo una chiara separazione dei concetti. Infine sono stati testati diversi environment, da una semplice strada rettilinea fino ad arrivare a scenari più complessi con incroci regolati da semafori.





Progetto 2 – Sudoku MOM

Analisi del problema

Punti critici:

Architettura proposta

Esempio di punti chiave:

- Architettura a 3 livelli (frontend, backend, database)
- API REST per la comunicazione
- Gestione autenticazione via JWT

Tecnologie utilizzate

Componente	Tecnologia	Ruolo
Frontend	fyne.io	Interfaccia utente
Backend	GO	Logica applicativa

Sviluppo

Risultati e considerazioni

Progetto 3 – Java-RMI

Analisi del problema

Punti critici:

Architettura proposta

Tecnologie utilizzate

Componente	Tecnologia	Ruolo
Frontend	fyne.io	Interfaccia utente
Backend	GO	Logica applicativa

Sviluppo

Risultati e considerazioni

Progetto 4 – Guess The Number

Analisi del problema

Si intende progettare un sistema in cui l'**Oracolo** genera un numero pseudo–casuale compreso nell'intervallo 0, **MAX**. A ogni turno, ciascun giocatore deve inviare una proposta di valore nel tentativo di indovinare il numero estratto. Se la proposta coincide con il valore generato, l'Oracolo inoltra un messaggio di *vittoria* al giocatore corretto e un messaggio di *sconfitta* a tutti gli altri partecipanti.
In caso di tentativo errato, l'Oracolo risponde con un messaggio di *hint*, specificando se il valore proposto è maggiore o minore rispetto al numero da indovinare.
Ogni giocatore dispone di un singolo tentativo per turno; una volta ricevute tutte le proposte, l'Oracolo avvia un nuovo ciclo di turno. L'ordine di invio dei tentativi non è deterministico.
A inizio turno l'Oracolo notifica a tutti i giocatori la possibilità di inviare il proprio tentativo e rimane in attesa dei rispettivi messaggi. I giocatori sono implementati come attori autonomi (bot), ciascuno dei quali produce il proprio tentativo dopo un ritardo temporale randomico. È inoltre prevista la possibilità di integrare un giocatore umano, che interagisce con il sistema attraverso un'interfaccia dedicata.

Punti critici:

- Creare la struttura col paradigma del linguaggio GO
- create un sistema per coordinare i vari giocatori (bot)
- Realizzare la ciclicità dei turni per tentare di indovinare il numero

Architettura proposta

coordinazione tra oracolo e giocatori

Tecnologie utilizzate

Componente	Tecnologia	Ruolo
Frontend	fyne.io	Interfaccia utente
Backend	GO	Logica applicativa

Sviluppo

Oracolo

Giocatori

Risultati e considerazioni

Conclusioni

Riflessione generale sull'intero percorso:

- Competenze tecniche acquisite
- Tecnologie più efficaci o interessanti
- Difficoltà comuni e soluzioni riutilizzabili
- Possibili sviluppi futuri dei progetti

Riferimenti

- [1] Documentazione ufficiale delle tecnologie utilizzate

- [2] Tutorial o articoli di riferimento
- [3] Altri materiali consultati