

Assignment 03

Programmazione concorrente e distribuita

Madina Kentpayeva:

madina.kentpayeva@studio.unibo.it

Jahrim Gabriele Cesario:

jahrim.cesario2@studio.unibo.it

Il sorgente del sistema è disponibile sul seguente repository:
> `git clone https://github.com/jahrim/pcd-assignment-03.git`

Sommario

1. Ex-01: Actor Framework.....	3
1.1. Analisi del problema.....	3
1.2. Architettura proposta	3
1.2.1. Design concorrente.....	3
1.2.2. Design applicativo	5
1.3. Analisi delle performance	10
1.3.1. Risultati delle analisi.....	10
1.3.2. Osservazioni.....	14
1.4. Istruzioni all'uso	14
2. Ex-02: Distributed Actor Framework.....	15
2.1. Analisi del problema.....	15
2.2. Architettura proposta	17
2.2.1. Design concorrente.....	17
2.2.2. Design applicativo	18
2.3. Istruzioni all'uso	31
2.4. Demo dell'applicazione	32

1. Ex-01: Actor Framework

1.1. Analisi del problema

Si rimanda all'analisi del primo assignment.

1.2. Architettura proposta

1.2.1. Design concorrente

Considerando i macro-task individuati nel primo assignment, si è deciso di adottare la stessa architettura master-worker, riadattata però secondo i principi del paradigma ad attori.

Nell'architettura progettata, sono state modellate due tipologie di attori: un **coordinatore** e diversi **delegati** (rappresentati in *Figura 1*).

Ogni attore sarà descritto in termini della sua **funzione** all'interno del sistema, dello **stato** che deve mantenere, dei **messaggi** che deve poter ricevere e dei **comportamenti** che può adottare durante il suo ciclo di vita.

Il coordinatore è un attore che ha lo scopo di gestire l'esecuzione di una simulazione dall'inizio alla fine. Per farlo si affida a un insieme di delegati, a cui invierà dei task da eseguire aspettandone il risultato.

Durante un'iterazione della simulazione, il coordinatore svolge in ordine i tre macro-task, eseguendo per ognuno la seguente sequenza di azioni:

1. **Partizionamento dei corpi:** partiziona in modo omogeneo l'insieme dei corpi della simulazione, in un numero di partizioni pari al numero dei suoi delegati;
2. **Delegazione dei compiti:** invia un messaggio ad ogni suo delegato, ordinando di svolgere la parte del macro-task relativa ad una specifica partizione;
3. **Ricostruzione della simulazione:** attende di ricevere un messaggio da ogni suo delegato, contenente il risultato del compito a lui assegnato. Alla ricezione di ogni messaggio, aggiorna i corpi della simulazione sostituendoli con quelli nuovi ricevuti;
4. **Ripete il procedimento per il macro-task successivo.**

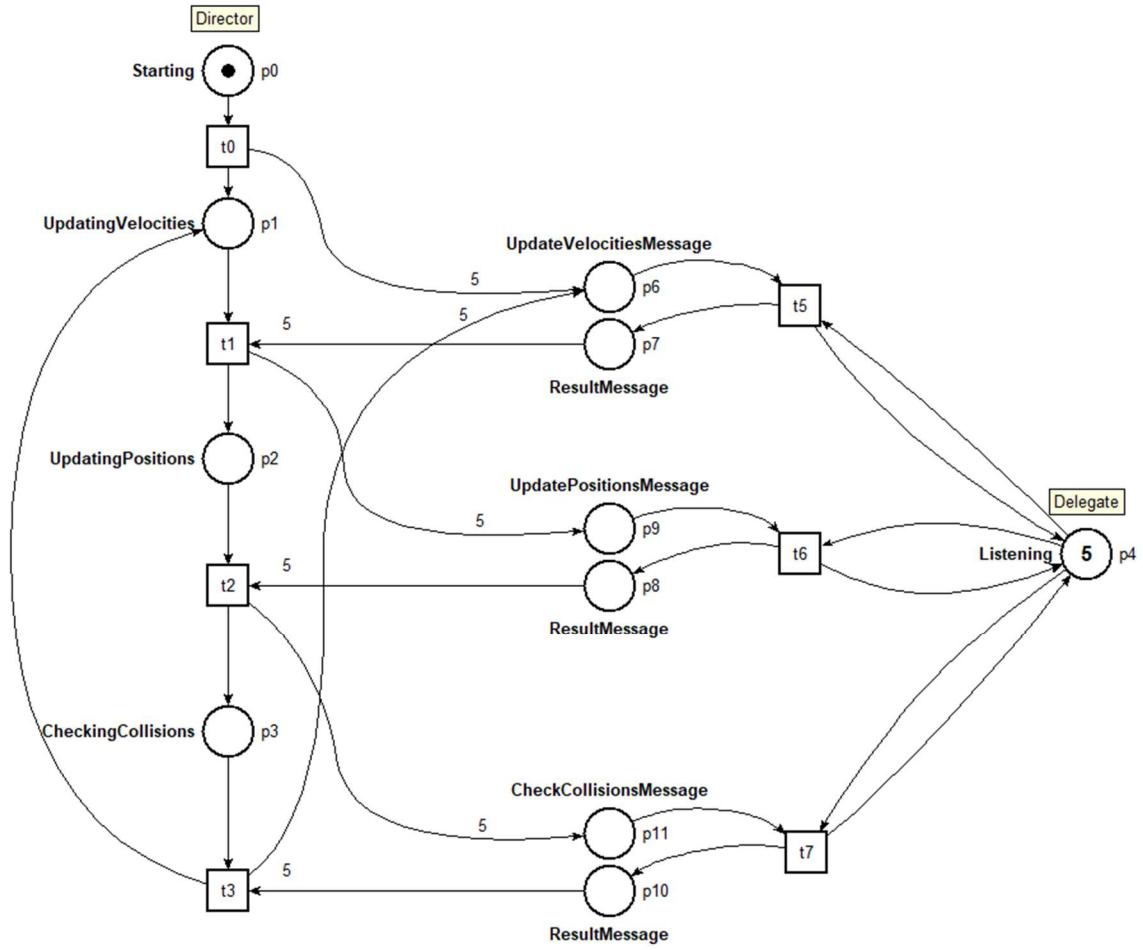


Figura 1. Rete di Petri che descrive i comportamenti del coordinatore e dei delegati, insieme ai messaggi che possono scambiarsi tra loro.

Per ricostruire la simulazione aggiornata durante l'esecuzione di un macro-task, il coordinatore deve predisporre a ricevere almeno un tipo di **messaggio**, che deve contenere i corpi aggiornati da un delegato in seguito all'esecuzione del suo compito. Ogni volta che un macro-task viene completato, il coordinatore cambia il proprio **comportamento**, occupandosi del macro-task successivo. Quindi, il coordinatore possiede almeno tre comportamenti corrispondenti ai tre macro-task della simulazione. In questo modo, si evita che macro-task diversi possano essere eseguiti concorrentemente, rispettando i vincoli relativi al loro ordine d'esecuzione. Per adempiere alla sua funzione, il coordinatore necessita di mantenere nel suo **stato** i *riferimenti ai propri delegati* e lo *stato corrente della simulazione*.

Un delegato, appena viene creato, si mette in attesa di ricevere dall'esterno degli ordini da eseguire. In particolare, deve poter accettare almeno tre tipologie di **messaggi**, corrispondenti ai task che gli saranno assegnati dal

coordinatore durante la simulazione. Ogni messaggio deve indicare al delegato la partizione della simulazione su cui eseguire uno specifico task ed il mittente, a cui sarà inoltrato il risultato della sua richiesta una volta completata. Per adempiere alla sua funzione, un delegato non necessita di mantenere alcuno **stato**, inoltre, è sufficiente l'unico **comportamento** in cui rimane in attesa di ordini dall'esterno.

1.2.2. Design applicativo

Per la realizzazione dell'applicazione si è scelto come design pattern il **Model-View-Controller (MVC)**, come descritto in *Figura 2*. In particolare, sono stati riutilizzati il Model e la View del primo assignment, riadattati a un nuovo Controller, basato sul paradigma ad attori e precisamente sulla libreria **Akka** per Java.

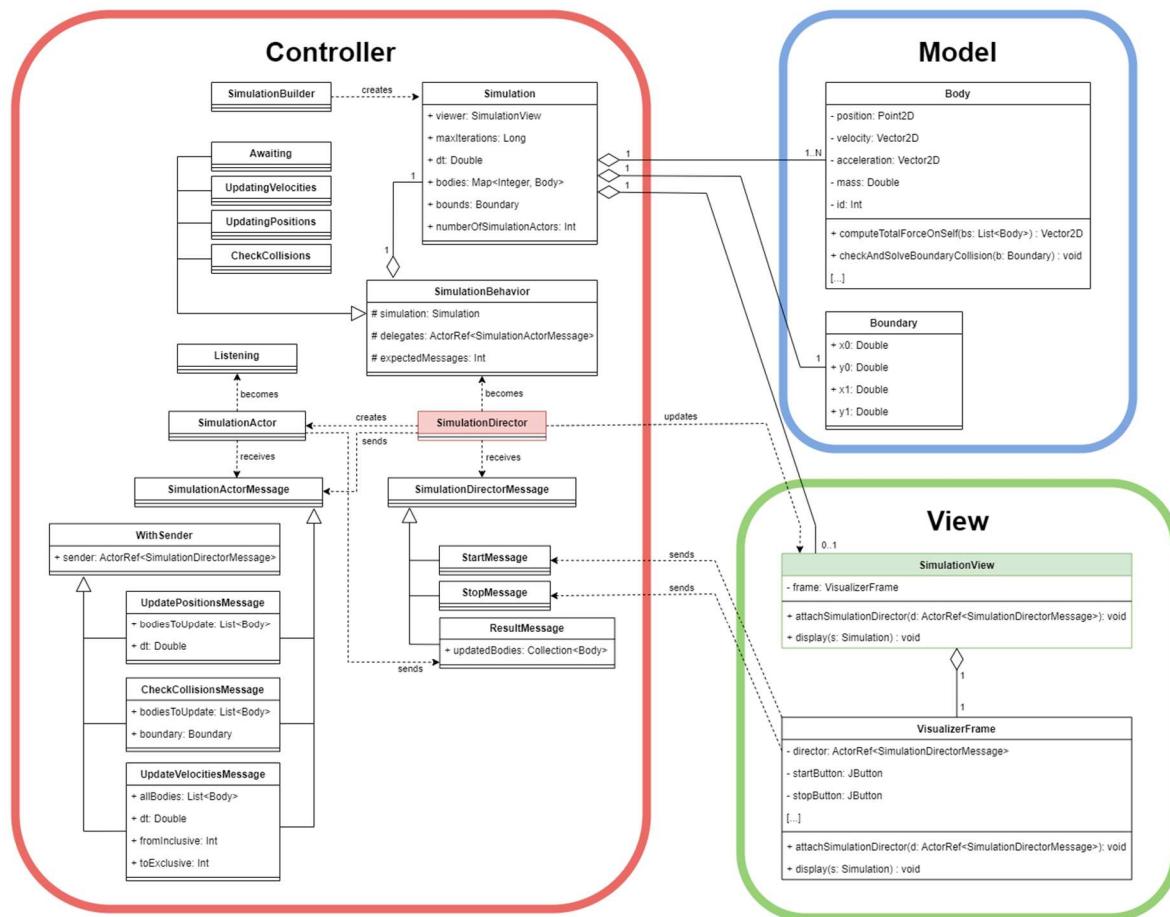


Figura 2. Architettura di massima del sistema.

Il Controller dell'applicazione (*Figura 3*) è ora modellato come un attore ed è il *coordinatore* del sistema, rappresentato dalla classe **SimulationDirector**.

Il **SimulationDirector** si occupa di gestire l'esecuzione di una certa simulazione. Per essere creato, il **SimulationDirector** richiede che gli sia specificata una simulazione da eseguire, rappresentata dalla classe **Simulation**. Una **Simulation** contiene tutta la configurazione relativa a una simulazione, tra cui il numero di iterazioni da eseguire e il numero di attori da utilizzare come *delegati*. È possibile configurare e costruire facilmente una simulazione attraverso un **SimulationBuilder**.

Alla sua creazione, il **SimulationDirector** genera degli attori figli che fungeranno da *delegati*, rappresentati dalla classe **SimulationActor**, ai quali invierà dei task da eseguire per tutta la durata della simulazione.

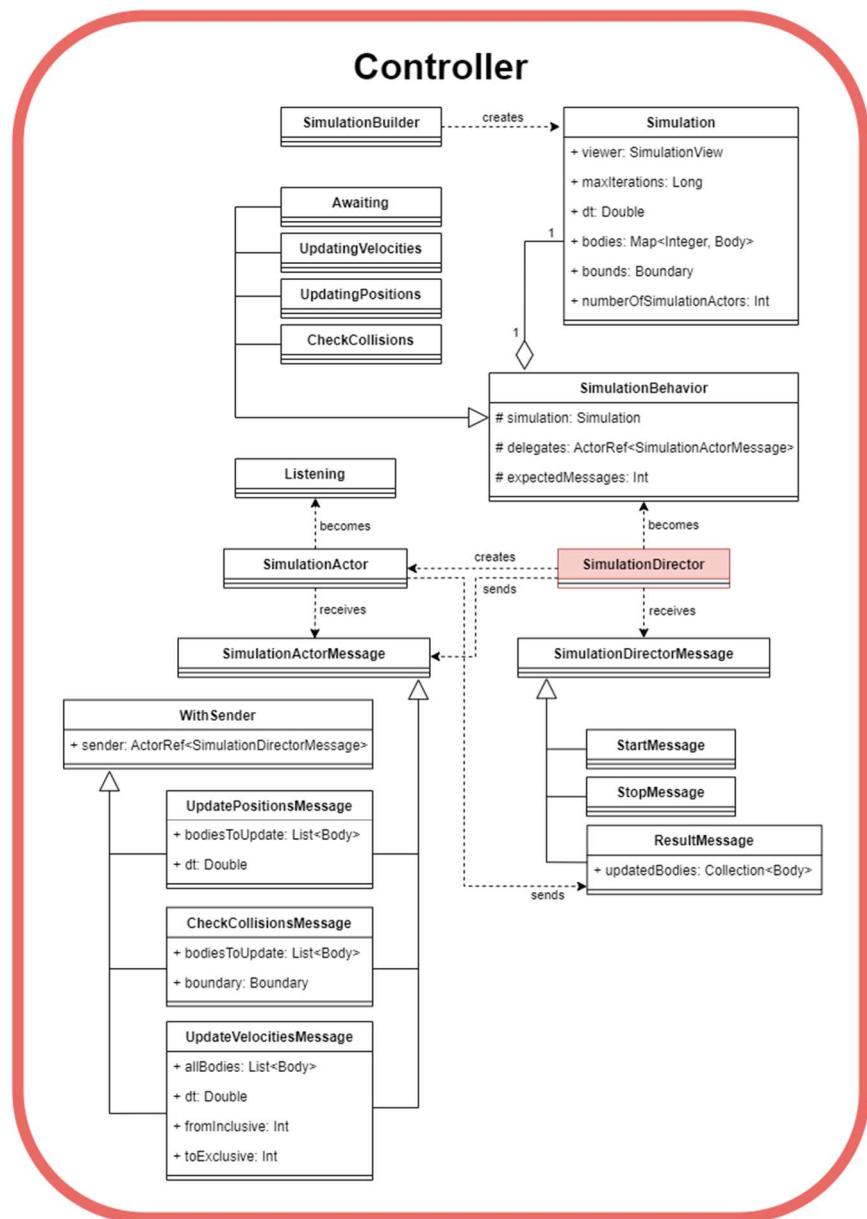


Figura 3. Diagramma delle classi del Controller dell'applicazione.

1.2.2.1. SimulationDirector

Il **SimulationDirector** (Figura 4) modella il Controller e il coordinatore del sistema.

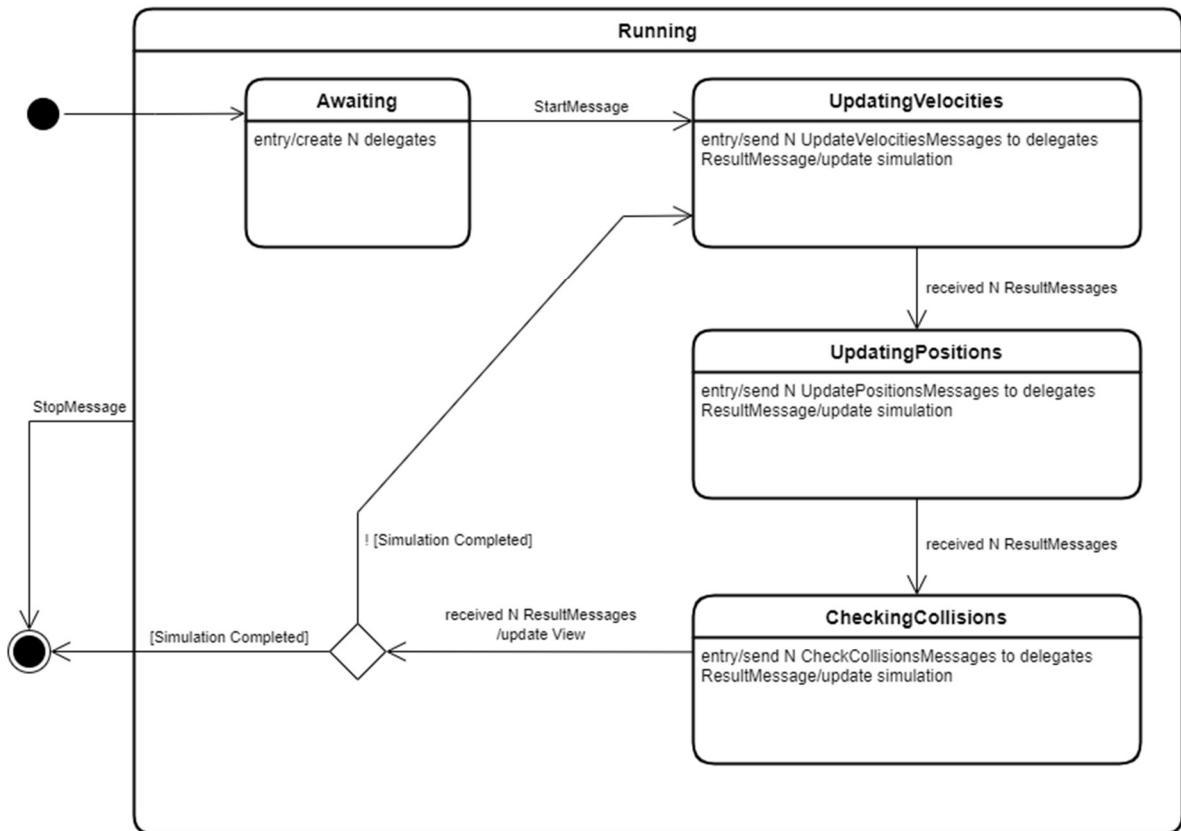


Figura 4. Diagramma a stati dell'attore **SimulationDirector**.

In quanto attore, è caratterizzato da:

- Uno **stato**: durante tutto il suo ciclo di vita, il **SimulationDirector** deve memorizzare lo *stato corrente della simulazione*. Inoltre, deve mantenere anche i *riferimenti ai propri figli*, ovvero i **SimulationActor**, a cui delegherà i task della simulazione. Per sincronizzarsi con i propri figli, il **SimulationDirector** tiene anche traccia del *numero di messaggi che ancora attende di ricevere* da essi.
- Un insieme di **messaggi** accettati:
 - **StartMessage**: indica al **SimulationDirector** di cominciare la simulazione;
 - **StopMessage**: indica al **SimulationDirector** di terminare la simulazione;

- **ResultMessage**: indica al `SimulationDirector` di sostituire alcuni corpi della sua simulazione con i corrispondenti aggiornati contenuti in questo messaggio, riconoscibili attraverso l'identificatore univoco che è associato ad ogni corpo.
- Un insieme di **comportamenti** assumibili:
 - **Awaiting**: il primo comportamento assunto dal `SimulationDirector`. In questo comportamento, il `SimulationDirector` crea i `SimulationActor` a cui saranno delegati i task della simulazione, dopodiché si mette in attesa del messaggio `StartMessage` prima di passare ad `UpdatingVelocities`;
 - **UpdatingVelocities**: in questo comportamento, il `SimulationDirector` partiziona i corpi della simulazione in un numero di partizioni pari al numero dei delegati. Per ogni partizione, invia un messaggio `UpdateVelocitiesMessage`, contenente la partizione, ad uno dei suoi figli. Alla ricezione di un numero di `ResultMessage` pari al numero di partizioni, passa ad `UpdatingPositions`;
 - **UpdatingPositions**: in questo comportamento, il `SimulationDirector` partiziona i corpi della simulazione in un numero di partizioni pari al numero dei delegati. Per ogni partizione, invia un messaggio `UpdatePositionsMessage`, contenente la partizione, ad uno dei suoi figli. Alla ricezione di un numero di `ResultMessage` pari al numero di partizioni, passa a `CheckingCollisions`;
 - **CheckingCollisions**: in questo comportamento, il `SimulationDirector` partiziona i corpi della simulazione in un numero di partizioni pari al numero dei delegati. Per ogni partizione, invia un messaggio `CheckCollisionsMessage`, contenente la partizione, ad uno dei suoi figli. Alla ricezione di un numero di `ResultMessage` pari al numero di partizioni, il `SimulationDirector` aggiorna la `View`, prima di tornare ad `UpdatingVelocities` o di terminare, se la simulazione è stata completata.

In un qualunque comportamento, il `SimulationDirector` può ricevere un messaggio `StopMessage` che termina la sua esecuzione.

1.2.2.1. *SimulationActor*

Il `SimulatorActor` (*Figura 5*) modella un delegato nel contesto di esecuzione di una simulazione gestita da un certo `SimulatorDirector`.

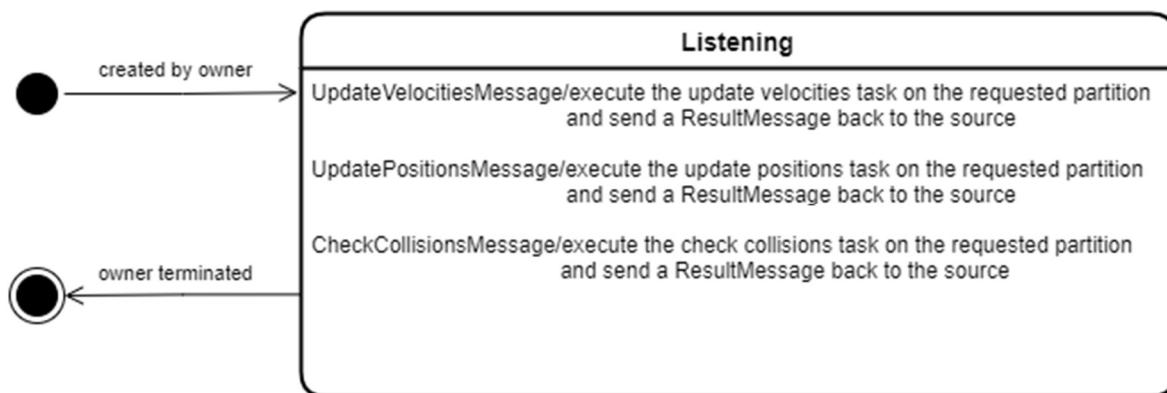


Figura 5. Diagramma a stati dell'attore `SimulationActor`.

In quanto attore, è caratterizzato da:

- Uno **stato**: per adempiere alle sue funzioni, un `SimulationActor` non necessita di mantenere alcuna informazione.
- Un insieme di **messaggi** accettati:
 - **UpdateVelocitiesMessage**: contiene tutti i corpi della simulazione e indica al `SimulationActor` di valutare le nuove velocità dei corpi appartenenti a una partizione della simulazione;
 - **UpdatePositionsMessage**: contiene una partizione dei corpi della simulazione e indica al `SimulationActor` di valutarne le nuove posizioni;
 - **CheckCollisionsMessage**: contiene una partizione dei corpi della simulazione e indica al `SimulationActor` di gestire eventuali collisioni tra quei corpi e il perimetro della simulazione.

Ognuno di questi messaggi contiene anche un riferimento al mittente, che sarà utilizzato per restituire il risultato dei task eseguiti dal `SimulationActor`.

- Un insieme di **comportamenti** assumibili:
 - **Listening**: il primo ed unico comportamento di un `SimulationActor`. In questo comportamento, il

`SimulationActor` attende di ricevere dei task dall'esterno. Alla ricezione di un messaggio inerente a un certo task, esegue il compito richiesto ed invia al mittente un `ResultMessage`, contenente un insieme dei corpi aggiornati.

In quanto vincolato al contesto di un `SimulationDirector`, un `SimulationActor` termina quando il `SimulationDirector` che lo controlla ha terminato la sua esecuzione.

1.3. Analisi delle performance

1.3.1. Risultati delle analisi

Come richiesto dalle specifiche, sono state testate le performance del sistema sotto condizioni variabili, relativamente al numero di corpi, iterazioni e attori utilizzati durante l'esecuzione della simulazione.

I risultati sono mostrati nelle seguenti tabelle, che descrivono i tempi di esecuzione delle varie simulazioni, con a fianco le approssimazioni dei relativi speed up. Al fine di confrontare i risultati delle performance rispetto a quelli del primo e secondo assignment, si riportano anche i risultati ottenuti con il `SinglePoolMaster` e l'`ExecutorMaster`.

Come processore di riferimento, è stato scelto l'Intel i7-8750H (2.2GHz & 6 Core + 6 Core logici), ovvero quello che forniva i risultati più affidabili.

1.3.1.1. Risultati del primo assignment

1 Worker

#Body \ #Iteration	1000	5000	10000
100	0.14s (1)	0.71s (1)	1.41s (1)
1000	7.58s (1)	38.54s (1)	78.14s (1)
5000	200.23s (1)	853.90s (1)	2022.17s (1)

2 Worker

#Body \ #Iteration	1000	5000	10000
100	0.14s (1)	0.65s (1.1)	1.35s (1.0)
1000	4.27s (1.8)	20.99s (1.8)	41.85s (1.9)
5000	106.90s (1.9)	437.71s (1.9)	1208.01s (1.7)

4 Worker

#Body \ #Iteration	1000	5000	10000
100	0.13s (1.1)	0.71s (1)	1.28s (1.1)
1000	2.44s (3.1)	12.39s (3.1)	25.44s (3.5)
5000	64.35s (3.1)	265.76s (3.2)	653.26s (3.1)

8 Worker

#Body \ #Iteration	1000	5000	10000
100	0.14s (1)	0.66s (1.1)	1.36s (1.0)
1000	2.04s (3.7)	9.81s (3.9)	19.67s (3.9)
5000	42.24s (4.7)	253.34s (3.4)	458.60s (4.4)

13 Worker

#Body \ #Iteration	1000	5000	10000
100	0.13s (1.1)	0.66s (1.1)	1.39s (1.0)
1000	1.84s (4.1)	9.16s (4.2)	18.33s (4.3)
5000	42.78s (4.7)	174.82s (4.9)	453.77s (4.5)

1.3.1.2. Risultati del secondo assignment

Fixed Thread Pool Executor (1 Worker)

#Body \ #Iteration	1000	5000	10000
100	0.36s (1)	0.84s (1)	1.64s (1)
1000	7.56s (1)	38.63s (1)	76.26s (1)
5000	179.84s (1)	939.56s (1)	2521.93s (1)

Fixed Thread Pool Executor (4 Worker)

#Body \ #Iteration	1000	5000	10000
100	0.27s (1.3)	0.90s (0.9)	1.70s (0.9)
1000	2.37s (3.2)	12.24s (3.2)	24.61s (3.1)
5000	51.02s (3.5)	266.56s (3.5)	703.94s (3.6)

Fixed Thread Pool Executor (8 Worker)

#Body \ #Iteration	1000	5000	10000
100	0.17s (2.1)	0.82s (1.0)	1.61s (1.0)
1000	1.94s (3.9)	9.66s (4.0)	19.90s (3.8)
5000	38.81s (4.6)	206.91s (4.5)	544.10s (4.6)

Fixed Thread Pool Executor (13 Worker)

#Body \ #Iteration	1000	5000	10000
100	0.18s (2)	0.80s (1.1)	1.61s (1.0)
1000	1.86s (4.1)	8.99s (4.3)	19.11s (4.0)
5000	35.57s (5.1)	239.33s (3.9)	458.11s (5.5)

Cached Thread Pool Executor

#Body \ #Iteration	1000	5000	10000
100	0.69s (0.5)	2.89s (0.3)	6.07s (0.3)
1000	5.20s (1.5)	26.76s (1.4)	55.19s (1.4)
5000	62.49s (2.9)	386.94s (2.4)	729.06s (3.5)

1.3.1.2. Risultati del terzo assignment

Akka framework (1 Actor)

#Body \ #Iteration	1000	5000	10000
100	1.70s (1)	0.97s (1)	1.79s (1)
1000	6.24s (1)	37.60s (1)	74.99s (1)
5000	183.65s (1)	904.73s (1)	2164.06s (1)

Akka framework (2 Actor)

#Body \ #Iteration	1000	5000	10000
100	0.41s (4.1)	0.97s (1.0)	1.75s (1.0)
1000	5.14 s (1.2)	31.40s (1.2)	62.68s (1.2)
5000	140.21s (1.3)	721.89s (1.3)	1579.23s (1.4)

Akka framework (4 Actor)

#Body \ #Iteration	1000	5000	10000
100	0.32s (5.3)	0.94s (1.0)	1.72s (1.0)
1000	4.11s (1.5)	24.57s (1.5)	49.08s (1.5)
5000	103.97s (1.8)	517.22s (1.7)	1194.70s (1.8)

Akka framework (8 Actor)

#Body \ #Iteration	1000	5000	10000
100	0.36s (4.7)	1.04s (0.9)	2.03s (0.9)
1000	3.42s (1.8)	18.65s (2.0)	37.12s (2.0)
5000	71.39s (2.6)	363.73s (2.5)	820.15s (2.6)

Akka framework (13 Actor)

#Body \ #Iteration	1000	5000	10000
100	0.38s (4.5)	1.35s (0.7)	2.62s (0.7)
1000	3.41s (1.8)	16.76s (2.2)	33.40s (2.2)
5000	60.60s (3.0)	308.75s (2.9)	738.20s (2.9)

Akka framework (26 Actor)

#Body \ #Iteration	1000	5000	10000
100	0.50s (3.4)	1.90s (0.5)	3.73s (0.5)
1000	3.14s (2.0)	15.65s (2.4)	30.73s (2.4)
5000	52.60s (3.5)	270.49s (3.3)	636.87s (3.4)

1.3.2. Osservazioni

Dai risultati ottenuti si osserva che:

- **Le performance ottenute mediante gli attori in Akka sono leggermente peggiori rispetto a quelle ottenute nel primo assignment.** Questo probabilmente dipende dal fatto che il framework utilizzato da Akka è più complesso e più ad alto livello rispetto a quello implementato nel primo assignment.
- **Le performance ottenute mediante gli attori in Akka continuano a migliorare anche utilizzando un numero di attori maggiore rispetto al numero di processori disponibili.** Da ciò consegue che aggiungere un attore al sistema non corrisponde ad utilizzare un thread in più per l'esecuzione.
- **Alla prima simulazione del benchmark si ottiene un tempo fuorviante, molto più lento di quanto previsto.** Questo potrebbe essere dovuto a un breve periodo di inizializzazione del framework Akka.
- **Le performance migliori sono state ottenute utilizzando il framework Executor di Java.** Tuttavia, è da sottolineare come la progettazione di sistemi basati su attori risulti più naturale e spontanea. Inoltre, gli attori sono più facilmente portabili in un ambiente distribuito.

1.4. Istruzioni all'uso

All'interno del progetto realizzato è possibile eseguire due applicazioni:

- **<project>/ex-01/src/main/java/App:**
permette l'esecuzione di una specifica simulazione con l'interfaccia grafica.
- **<project>/ex-01/src/main/java/SimulationBenchmark:**
permette l'esecuzione di un insieme di simulazioni senza l'interfaccia grafica, provando tutte le combinazioni possibili dei parametri specificati e monitorandone il tempo di esecuzione.

2. Ex-02: Distributed Actor Framework

2.1. Analisi del problema

Questo progetto ha l'obiettivo di realizzare un sistema di monitoraggio di allagamenti per una città divisa in distretti, ognuno dei quali è gestito da una caserma dei pompieri, che risponde agli allarmi generati da un insieme di pluviometri installati in sede. Il sistema dovrà essere realizzato in un ambiente distribuito, basandosi sul paradigma ad attori.

Durante l'analisi del problema, sono stati individuati i seguenti componenti attivi all'interno del sistema:

- **City:** rappresenta la città monitorata dal sistema (*Figura 6*). È caratterizzata da una posizione virtuale in uno spazio bidimensionale, una larghezza e un'altezza. Inoltre, una città è partizionata in un certo numero di distretti (*Figura 7*);

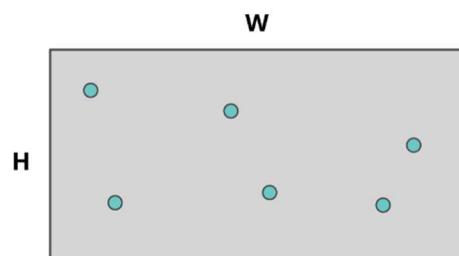


Figura 6. Modello bidimensionale di una città.

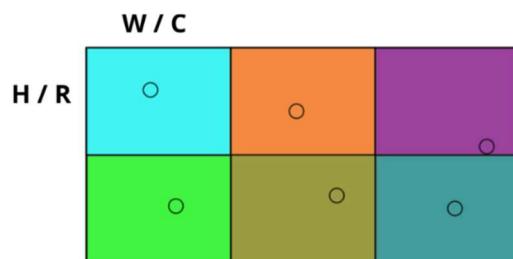


Figura 7. Suddivisione di una città in distretti.

- **Zone:** rappresenta un distretto di una città. È caratterizzato da una posizione virtuale in uno spazio bidimensionale, una larghezza, un'altezza e uno stato che ne indica la condizione di allarme. Inoltre, contiene una caserma dei pompieri e un massimo di tre pluviometri;
- **Firestation:** rappresenta una caserma dei pompieri. È caratterizzata da una posizione virtuale in uno spazio bidimensionale e uno stato che ne

indica la disponibilità. Una caserma di pompieri gestisce gli allarmi della città e può operare solo all'interno del distretto che la contiene.

Quando una caserma dei pompieri interviene nel proprio distretto, il distretto entra in uno stato di gestione dell'allarme e la caserma diventa occupata. Una volta risolto l'allarme, il distretto entra in uno stato di quiete e la caserma diventa disponibile;

- **Pluviometer:** rappresenta un pluviometro, ovvero un sensore che misura il livello delle precipitazioni. È caratterizzato da una posizione virtuale in uno spazio bidimensionale. Quando la maggioranza dei pluviometri di un distretto rileva una quantità di precipitazioni superiore ad una certa soglia, il distretto entra in uno stato di allarme, da cui non uscirà fino all'intervento di una caserma dei pompieri.

L'applicazione dovrà permettere all'utente di visualizzare le condizioni in cui si trova una certa città e quindi dovrà permettere di:

- *Visualizzare lo stato di ogni distretto;*
- *Visualizzare il numero di pluviometri in un certo distretto;*
- *Visualizzare lo stato delle caserme dei pompieri.*

Infine, attraverso l'applicazione sarà possibile *disattivare manualmente gli allarmi dei distretti della città.*

2.2. Architettura proposta

2.2.1. Design concorrente

In seguito alle riflessioni eseguite in fase di analisi, sono stati individuati cinque attori principali:

- **CityActor**: rappresenta una città. Gestisce la generazione dei distretti, delle caserme dei pompieri e dei pluviometri rispettando i vincoli evidenziati in fase di analisi. Inoltre, acquisirà periodicamente uno snapshot del sistema notificandone l'utente;
- **ZoneActor**: rappresenta un distretto. Gestisce la rilevazione dei segnali ricevuti dai propri pluviometri, entrando in uno stato di allarme se la maggioranza dei pluviometri ha emesso un segnale. Una volta entrato in allarme, notifica periodicamente la propria caserma dei pompieri fino a quando l'allarme non viene gestito;
- **FirestationActor**: rappresenta una caserma dei pompieri. Risponde agli allarmi del proprio distretto, intervenendo per disattivarli. Ricevuto un allarme, notifica il proprio distretto della presa in gestione di tale allarme. In seguito al proprio intervento, notifica il distretto che l'allarme è stato risolto;
- **PluviometerActor**: rappresenta un pluviometro. Misura periodicamente il livello delle precipitazioni nei dintorni, emettendo un segnale se la misurazione supera una certa soglia;
- **ViewActor**: gestisce l'interfaccia grafica dell'utente. In particolare, permette di connettersi al sistema di una certa città per monitorarne lo stato.

Il sistema distribuito è composto da un unico cluster e si è deciso di mappare ogni attore ad un nodo del cluster. Ad ogni attore, viene associato un identificatore univoco, con il quale sarà possibile accedere al servizio fornito da quell'attore all'interno del cluster. In questo modo, chiunque conosca l'identificatore di un certo attore potrà interagirvi.

Dopo l'inizializzazione del cluster, è possibile creare un CityActor rendendolo un membro del cluster. Alla creazione del CityActor, tutti componenti della città vengono generati e registrati sul cluster. In particolare, una città viene divisa in un certo numero di distretti, dopodiché per ogni distretto si genera

una caserma dei pompieri e un numero variabile di pluviometri, compreso tra un minimo di un pluviometro e un massimo di tre.

A questo punto, è possibile creare un ViewActor rendendolo un membro del cluster. Conoscendo l'identificatore di una certa città, è possibile connettere il ViewActor al CityActor di quella città, quindi visualizzandone lo stato in evoluzione.

2.2.2. Design applicativo

Per la realizzazione dell'applicazione, si è scelto di implementare due moduli principali, come descritto in *Figura 8*:

- **Actor Module:** realizzato attraverso la libreria **Akka** per Scala; gestisce il cluster degli attori dell'applicazione;
- **View:** realizzato mediante **ScalaFX**; gestisce l'interfaccia grafica dell'applicazione.

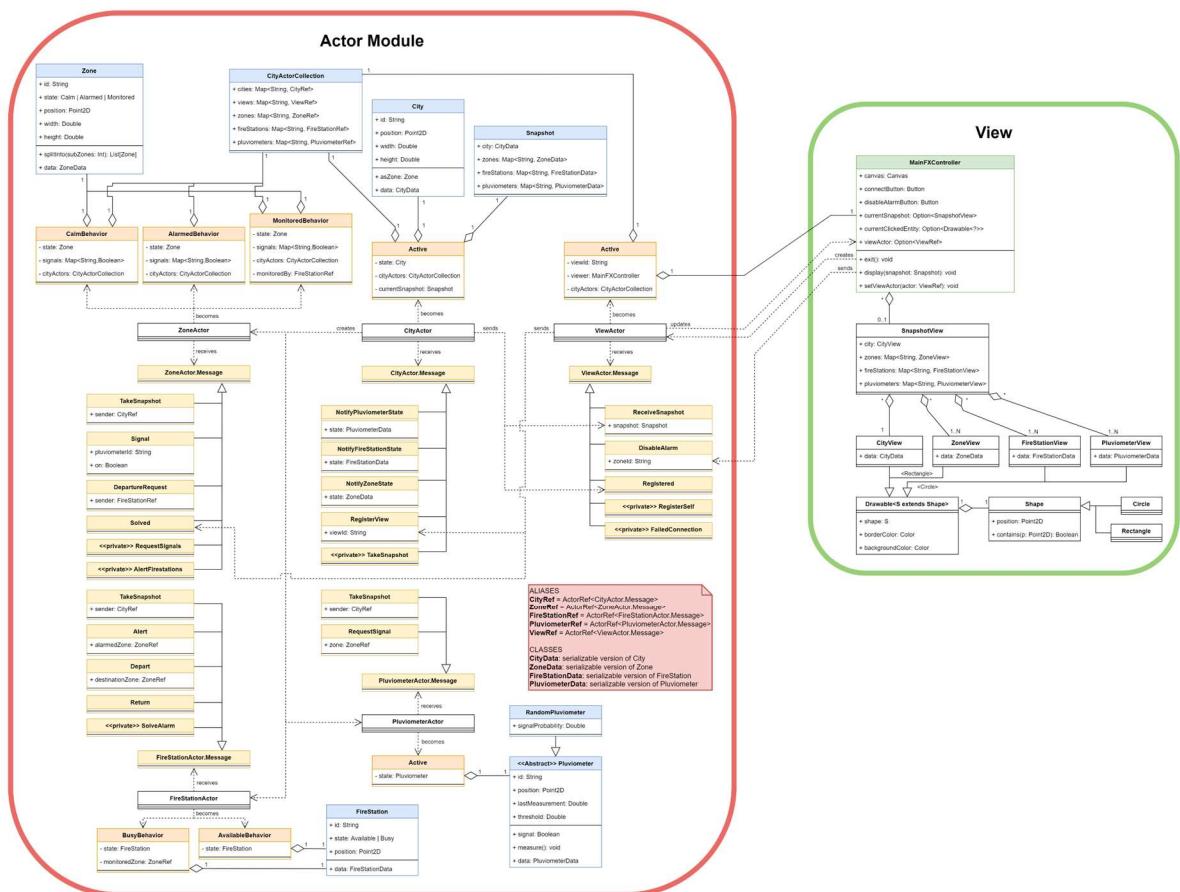


Figura 8. Diagramma delle classi del sistema.

2.2.2.1. Actor Module

L'Actor Module (Figura 9) contiene la definizione di tutti gli attori utilizzati all'interno del sistema. In particolare, definisce tutti gli attori individuati nella fase di design concorrente.

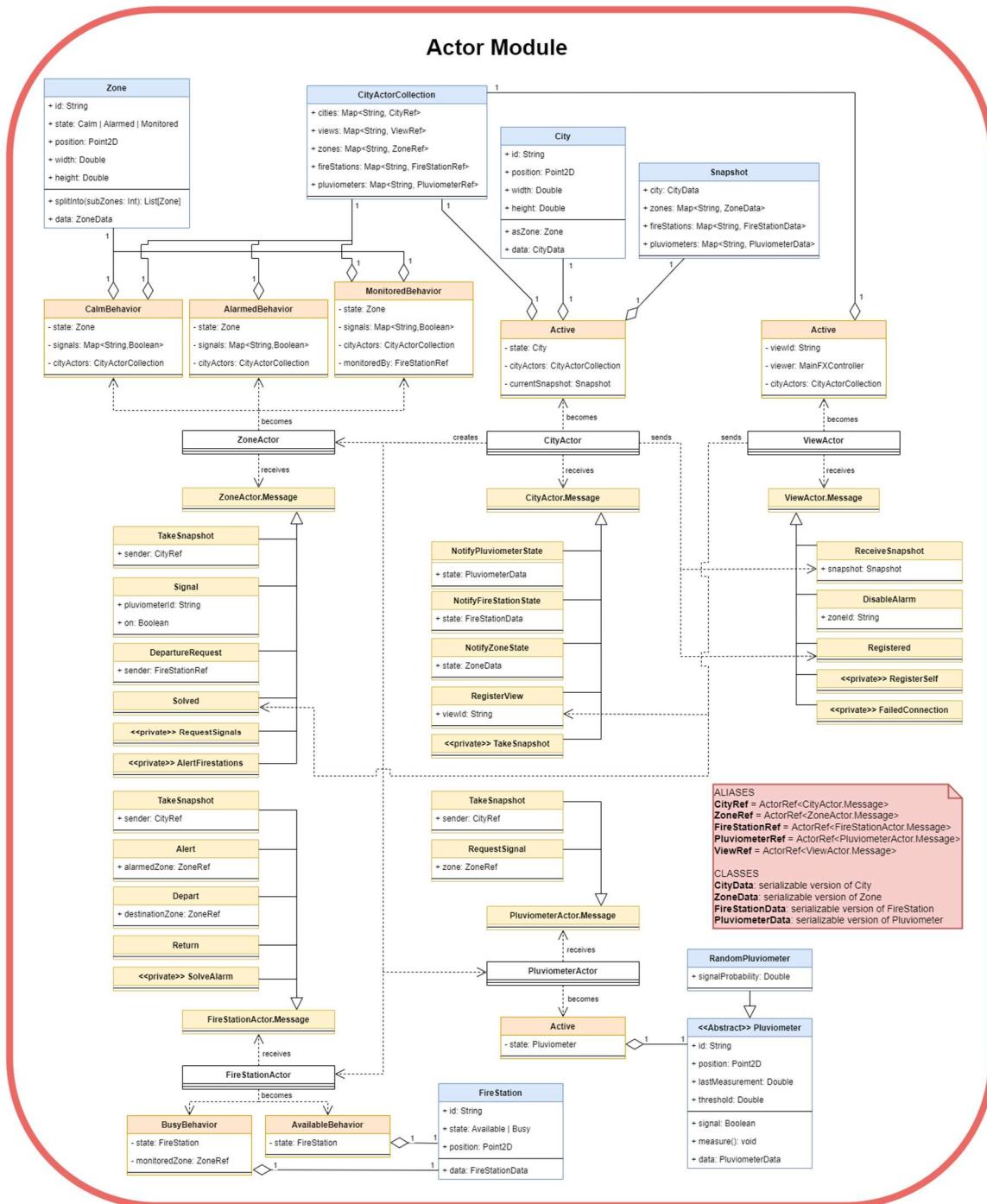


Figura 9. Diagramma delle classi dell'ActorModule. In blu, le classi che rappresentano lo stato degli attori. In arancione, le classi che rappresentano i comportamenti degli attori. In giallo, le classi che rappresentano i messaggi accettati dagli attori.

Ogni attore è caratterizzato da:

- Uno **stato**: contiene le informazioni conosciute dall'attore in un dato istante;
- Un insieme di **messaggi** accettati: descrive in quali modi è possibile interagire con l'attore;
- Un insieme di **comportamenti** assumibili: descrive in quali modi l'attore può reagire ai messaggi ricevuti.

Il modulo fa affidamento a un insieme di utility per la gestione di un cluster. In particolare, utilizza la classe **AkkaCluster**, che permette di creare un cluster data la sua configurazione. Conoscendo tale configurazione, è poi possibile registrare nuovi nodi nel cluster.

2.2.2.2.1 *CityActor*

Il **CityActor** (Figura 10) modella una città all'interno del sistema e gestisce l'acquisizione degli snapshot della città. Un CityActor accetta la registrazione da parte di molteplici ViewActor, notificandoli dei cambiamenti avvenuti nella propria città.

Alla creazione, un CityActor genera in modo casuale tutti i componenti della città che gestisce, registrandoli sul cluster.

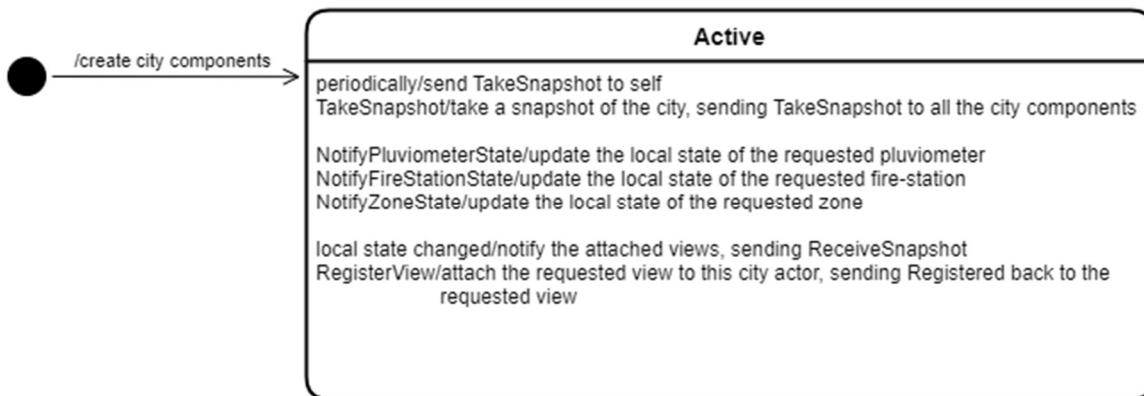


Figura 10. Diagramma a stati dell'attore CityActor.

In quanto attore, è caratterizzato da:

- Uno **stato**: per adempiere alle sue funzioni, un CityActor deve mantenere lo *stato attuale della città*, rappresentato dalla classe **City**,

l'ultimo snapshot eseguito, contenente gli stati di tutti i componenti della città, rappresentato dalla classe **Snapshot**, e i *riferimenti agli attori che gestiscono i componenti della città*, contenuti nella classe **CityActorCollection**.

- Un insieme di **messaggi** accettati:
 - **NotifyPluviometerState**: contiene il nuovo stato di un pluviometro e indica al CityActor di aggiornare lo snapshot corrente;
 - **NotifyFireStationState**: contiene il nuovo stato di una caserma dei pompieri e indica al CityActor di aggiornare lo snapshot corrente;
 - **NotifyZoneState**: contiene il nuovo stato di un distretto e indica al CityActor di aggiornare lo snapshot corrente;
 - **RegisterView**: contiene l'identificatore di un ViewActor e indica al CityActor di registrare tale ViewActor tra quelli conosciuti, così da notificarlo dei cambiamenti della città;
 - **TakeSnapshot**: indica al CityActor di richiedere lo stato ai componenti della città.
- Un insieme di **comportamenti** assumibili:
 - **Active**: il primo ed unico comportamento di un CityActor. In questo comportamento, il CityActor acquisisce periodicamente uno snapshot dei componenti della città, richiedendo il loro stato. Inoltre, accetta le richieste di registrazione da parte dei ViewActor. Ogni volta che riceve il nuovo stato da uno dei componenti della città, il CityActor aggiorna lo snapshot corrente, notificandone i ViewActor registrati.

Da notare come lo snapshot acquisito dalla città corrisponde al punto di vista della città sui suoi componenti. Quindi, potrebbero esserci delle discrepanze temporanee, ad esempio, tra gli stati dei pluviometri conosciuti dalla città e quelli conosciuti dal distretto a cui appartengono. Questo perché ogni attore aggiorna il proprio punto di vista sul sistema con tempistiche diverse, pertanto la città potrebbe aver ricevuto dei dati più aggiornati sui pluviometri, che il distretto a cui appartengono non ha ancora ricevuto.

2.2.2.2.2 ZoneActor

Lo **ZoneActor** (Figura 11) modella un distretto di una città e gestisce l'acquisizione dei segnali dei pluviometri del distretto.

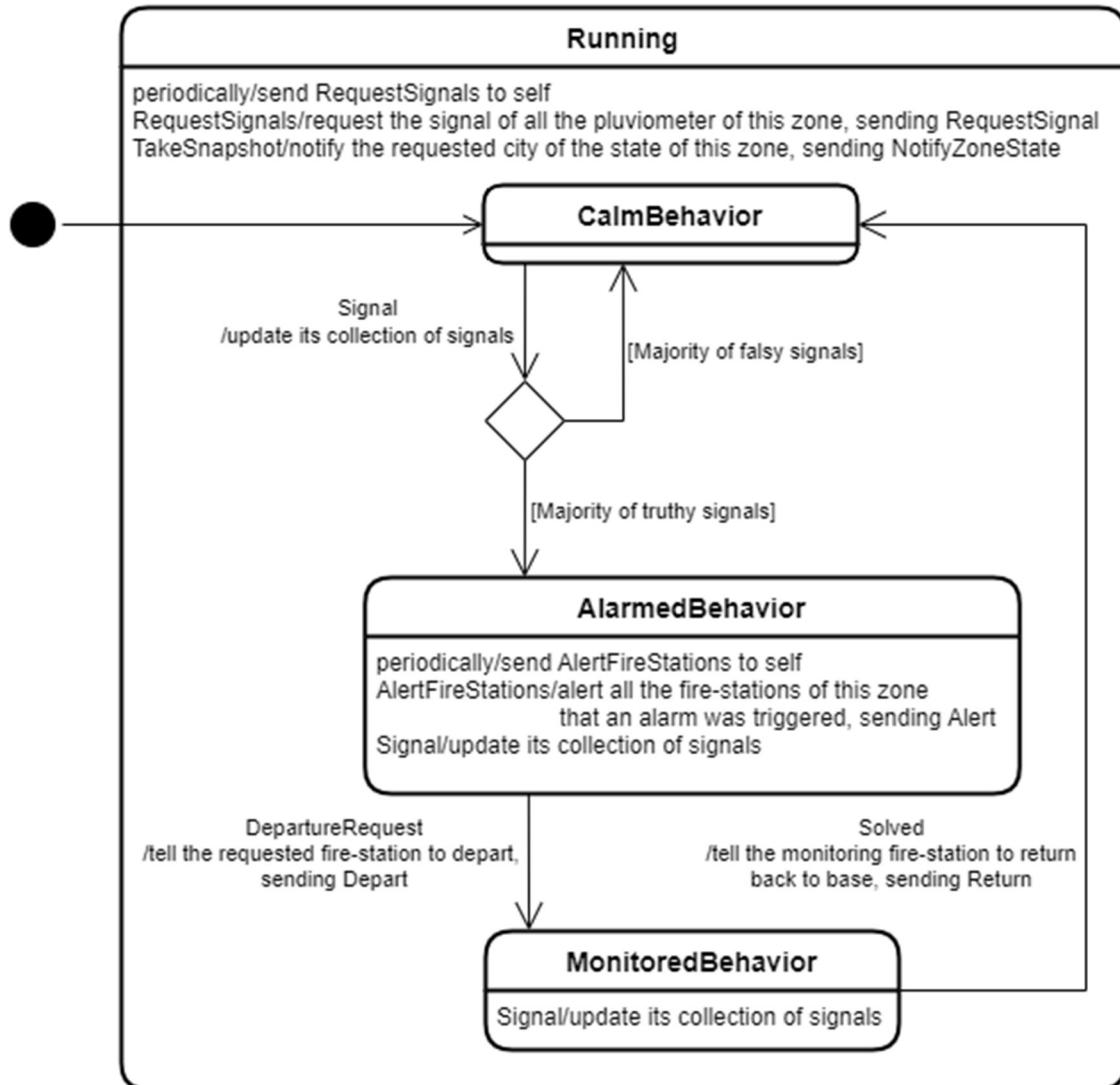


Figura 11. Diagramma a stati dell'attore ZoneActor.

In quanto attore, è caratterizzato da:

- Uno **stato**: per adempiere alle sue funzioni, uno ZoneActor deve mantenere lo *stato attuale del distretto*, rappresentato dalla classe **Zone**, i *riferimenti ai pluviometri del distretto*, contenuti nella classe **CityActorCollection**, una *mappa dei segnali ricevuti dai pluviometri del distretto* ed eventualmente un *riferimento alla caserma dei pompieri che la sta monitorando*.
- Un insieme di **messaggi** accettati:

- **TakeSnapshot**: indica allo ZoneActor di rispondere al mittente con il proprio stato;
- **Signal**: contiene l'identificatore di un pluviometro ed il suo segnale. Indica allo ZoneActor di aggiornare la mappa dei segnali;
- **DepartureRequest**: contiene il riferimento a una caserma dei pompieri e indica allo ZoneActor che tale caserma è disponibile a monitorare l'allarme del distretto;
- **Solved**: indica allo ZoneActor che la caserma dei pompieri che sta monitorando il distretto ha risolto l'allarme;
- **RequestSignals**: indica allo ZoneActor di richiedere i segnali ai pluviometri del distretto;
- **AlertFireStations**: indica allo ZoneActor di allertare le caserme dei pompieri del distretto.
- Un insieme di **comportamenti** assumibili:
 - **CalmBehavior**: il primo comportamento di uno ZoneActor. In questo comportamento, alla ricezione del segnale di un pluviometro, se la maggioranza o la metà dei pluviometri sta emettendo un segnale, passa ad **AlarmedBehavior**;
 - **AlarmedBehavior**: in questo comportamento, lo ZoneActor segnala periodicamente a tutte le caserme dei pompieri del distretto che è stato innescato un allarme. Alla prima caserma che notifica il distretto della propria disponibilità, è richiesto di intervenire per risolvere l'allarme, dopodiché lo ZoneActor passa a **MonitoredBehavior**;
 - **MonitoredBehavior**: in questo comportamento, lo ZoneActor aspetta che la caserma dei pompieri che sta monitorando il distretto risvolvi l'allarme. Alla ricezione di una conferma da parte della caserma o dell'utente che l'allarme è stato risolto, lo ZoneActor ordina ai pompieri di tornare alla base e ritorna in **CalmBehavior**.

In ogni comportamento, lo ZoneActor richiede periodicamente i segnali ai pluviometri del distretto, aggiornando la mappa dei segnali alla loro ricezione. Inoltre, quando richiesto, invia il proprio stato alla città a cui appartiene.

2.2.2.2.3 FireStationActor

Il **FireStationActor** (Figura 12) modella una caserma dei pompieri di una città e gestisce automaticamente gli allarmi del distretto a cui appartiene.

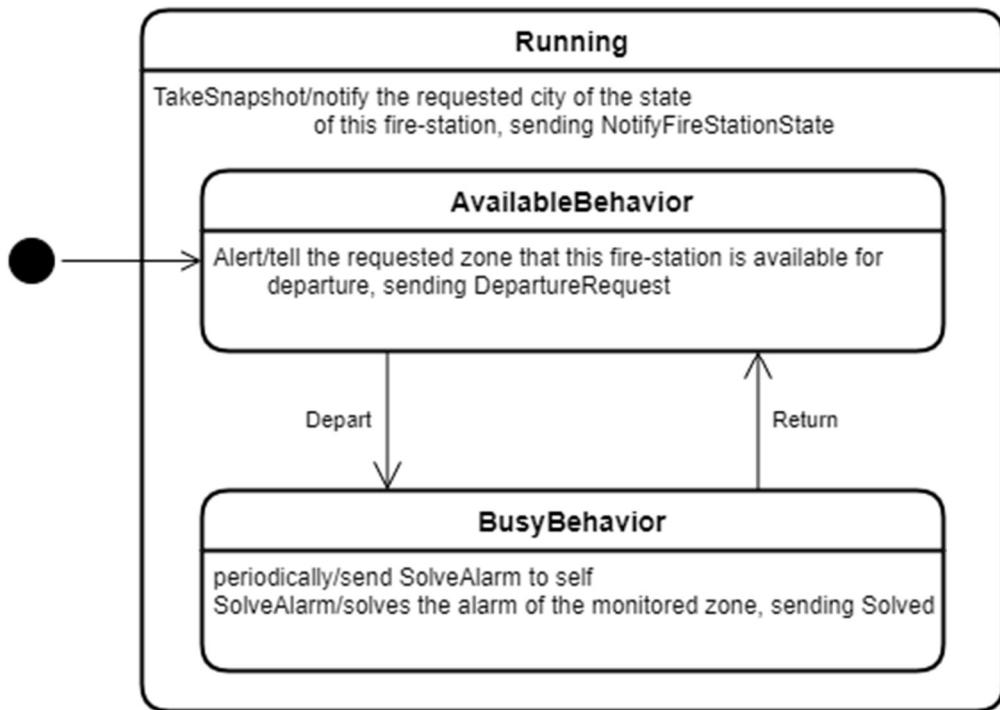


Figura 12. Diagramma a stati dell'attore **FireStationActor**.

In quanto attore, è caratterizzato da:

- Uno **stato**: per adempiere alle sue funzioni, un **FireStationActor** deve mantenere lo *stato attuale della caserma dei pompieri*, rappresentato dalla classe **FireStation**.
- Un insieme di **messaggi** accettati:
 - **TakeSnapshot**: indica al **FireStationActor** di rispondere al mittente con il proprio stato;
 - **Alert**: contiene il riferimento a un distretto e indica al **FireStationActor** che tale distretto è in allarme;
 - **Depart**: contiene il riferimento a un distretto e indica al **FireStationActor** che deve intervenire su tale distretto;
 - **Return**: indica al **FireStationActor** che può smettere di monitorare il distretto su cui è intervenuto;
 - **SolveAlarm**: indica al **FireStationActor** che il suo intervento sul distretto è terminato.
- Un insieme di **comportamenti** assumibili:

- **AvailableBehavior**: il primo comportamento di un FireStationActor. In questo comportamento, alla ricezione di un segnale d'allarme da parte di un distretto, il FireStationActor notifica il distretto della propria disponibilità. Alla ricezione di una richiesta di intervento da parte del distretto, il FireStationActor passa a **BusyBehavior**;
- **BusyBehavior**: in questo comportamento, il FireStationActor risolve periodicamente l'allarme del distretto che sta monitorando, notificando il distretto che l'intervento è terminato. Alla ricezione di una richiesta di rientro dei pompieri da parte del distretto, il FireStationActor torna in AvailableBehavior.

In ogni comportamento, il FireStationActor, quando richiesto, invia il proprio stato alla città a cui appartiene.

2.2.2.2.4 PluviometerActor

Il **PluviometerActor** (Figura 13) modella un pluviometro di una città e gestisce la rilevazione del livello delle precipitazioni nei suoi dintorni.

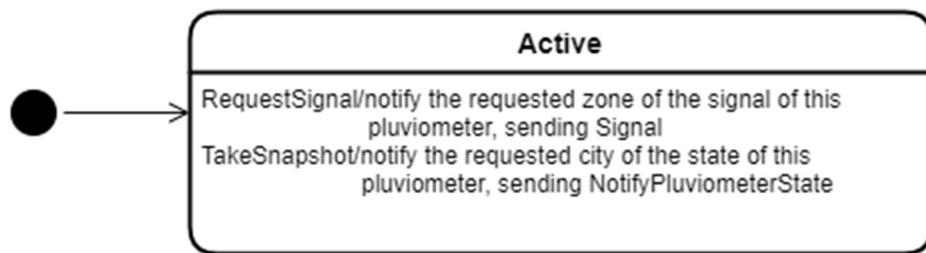


Figura 13. Diagramma a stati dell'attore PluviometerActor.

In quanto attore, è caratterizzato da:

- Uno **stato**: per adempiere alle sue funzioni, un PluviometerActor deve mantenere lo *stato attuale del pluviometro*, rappresentato dalla classe **Pluviometer**.
- Un insieme di **messaggi** accettati:
 - **TakeSnapshot**: indica al PluviometerActor di rispondere al mittente con il proprio stato;
 - **RequestSignal**: indica al PluviometerActor di rispondere al mittente con il proprio segnale.

- Un insieme di **comportamenti** assumibili:
 - **Active**: il primo ed unico comportamento di un PluviometerActor. In questo comportamento, quando richiesto, invia il proprio segnale al distretto a cui appartiene. Inoltre, sempre su richiesta, invia il proprio stato alla città a cui appartiene.

2.2.2.2.5 ViewActor

Il **ViewActor** (Figura 14) permette di connettersi a un CityActor per osservare i cambiamenti di una specifica città.

Per essere creato, un ViewActor necessita dell'identificatore della città che deve osservare e dell'interfaccia grafica a cui deve notificare i cambiamenti della città.

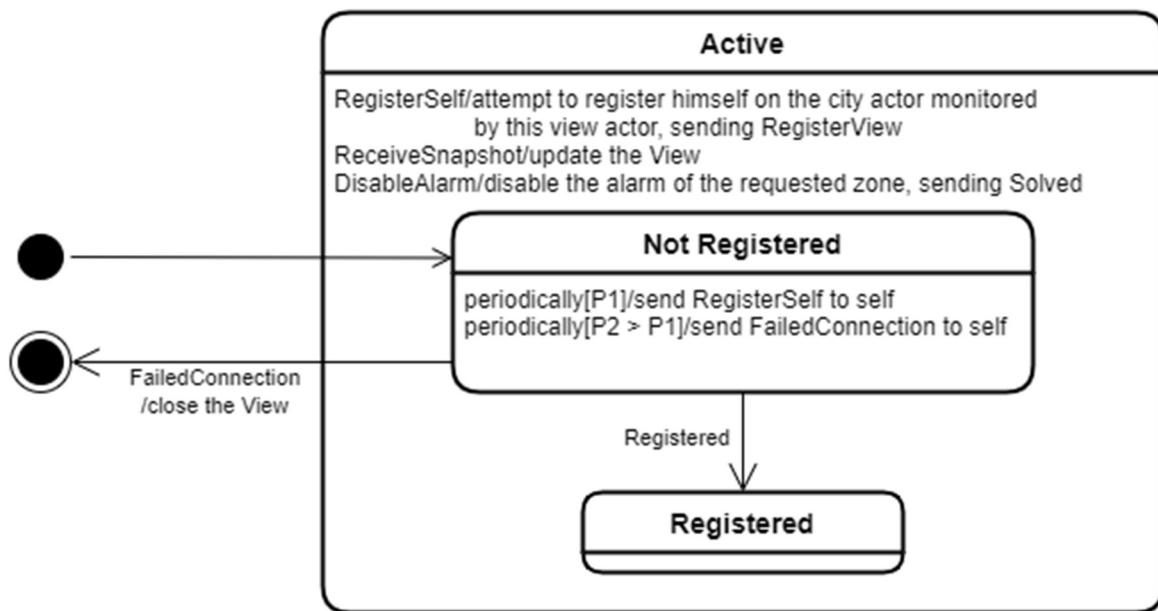


Figura 14. Diagramma a stati dell'attore ViewActor.

In quanto attore, è caratterizzato da:

- Uno **stato**: per adempiere alle sue funzioni, un ViewActor deve conoscere il *proprio identificatore*, l'*interfaccia grafica che controlla* ed i *riferimenti alla città a cui deve connettersi e ai distretti di cui l'utente può disabilitare l'allarme*, contenuti nella classe **CityActorCollection**.
- Un insieme di **messaggi** accettati:

- **ReceiveSnapshot**: contiene lo snapshot di una città e indica al ViewActor di aggiornare l’interfaccia grafica dell’applicazione;
 - **DisableAlarm**: contiene l’identificatore di un distretto e indica al ViewActor di disabilitare l’allarme di tale distretto;
 - **Registered**: indica al ViewActor che è riuscito a connettersi con successo alla città che deve osservare;
 - **RegisterSelf**: indica al ViewActor di riprovare a connettersi alla città che deve osservare;
 - **FailedConnection**: indica al ViewActor che ogni tentativo di connessione alla città che deve osservare è fallito.
- Un insieme di **comportamenti** assumibili:
 - **Active**: il primo ed unico comportamento di un ViewActor. In questo comportamento, il ViewActor tenta periodicamente di connettersi alla città che deve osservare. Alla ricezione di una conferma di avvenuta registrazione da parte della città, il ViewActor si considera effettivamente registrato. Se ciò non accade entro un certo timeout, il ViewActor termina e chiude l’interfaccia grafica dell’applicazione.
Una volta registrato, alla ricezione di un nuovo snapshot dalla città, aggiorna l’interfaccia grafica dell’applicazione. Alla richiesta dell’utente di disabilitare l’allarme di un distretto, il ViewActor notifica tale distretto che il suo allarme è stato risolto.

2.2.2.2. View

La **View** (Figura 15) contiene la definizione di tutti i componenti grafici necessari alla visualizzazione di una città.

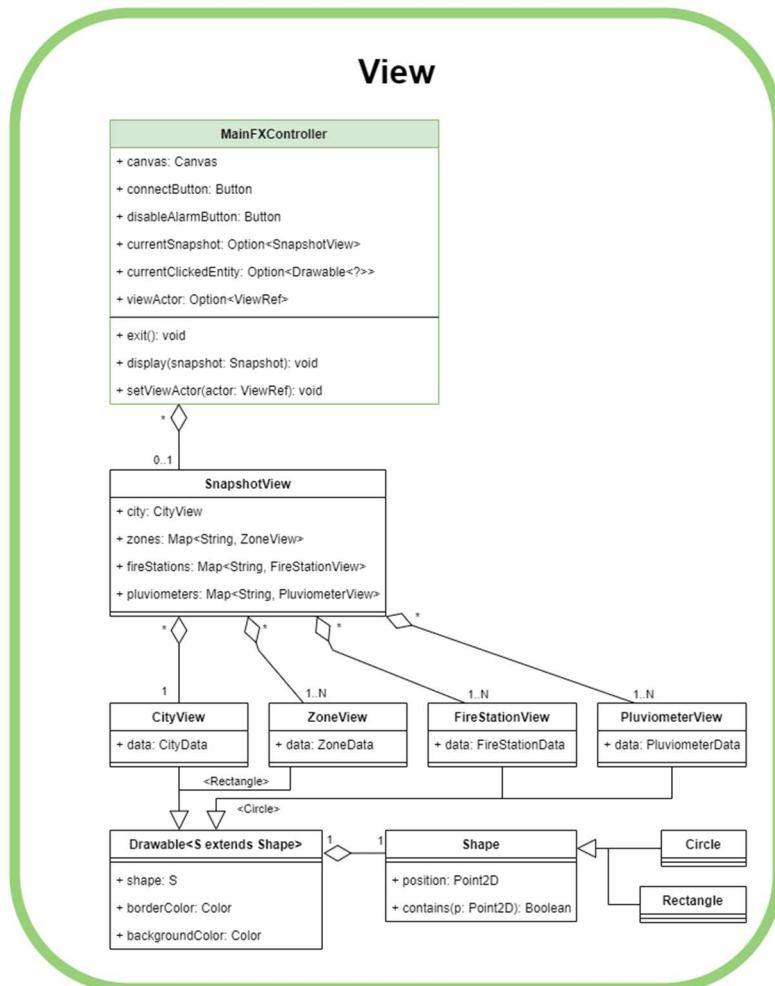


Figura 15. Diagramma delle classi della View.

La **View** si concretizza nella classe **MainFXController**, che gestisce l'intera interfaccia grafica dell'applicazione.

Il **MainFXController** permette di visualizzare lo snapshot di una città su un canvas, mappando ogni componente logico della città nel corrispondente componente grafico. In particolare, lo Snapshot della città viene mappato in uno **SnapshotView**.

Lo **SnapshotView** contiene degli insiemi di entità rappresentabili graficamente, modellate dalla classe **Drawable**. Ogni **Drawable** è caratterizzato da una certa forma, modellata dalla classe **Shape**.

All'interno dello SnapshotView, la città ed i distretti, modellati rispettivamente dalle classi **CityView** e **ZoneView**, assumono una forma rettangolare, mentre le caserme dei pompieri ed i pluviometri, modellati rispettivamente dalle classi **FireStationView** e **PluviometerView**, assumono una forma circolare.

L'interfaccia grafica dell'applicazione (*Figura 16*) permette all'utente di connettersi a una città di cui conosce l'identificatore.

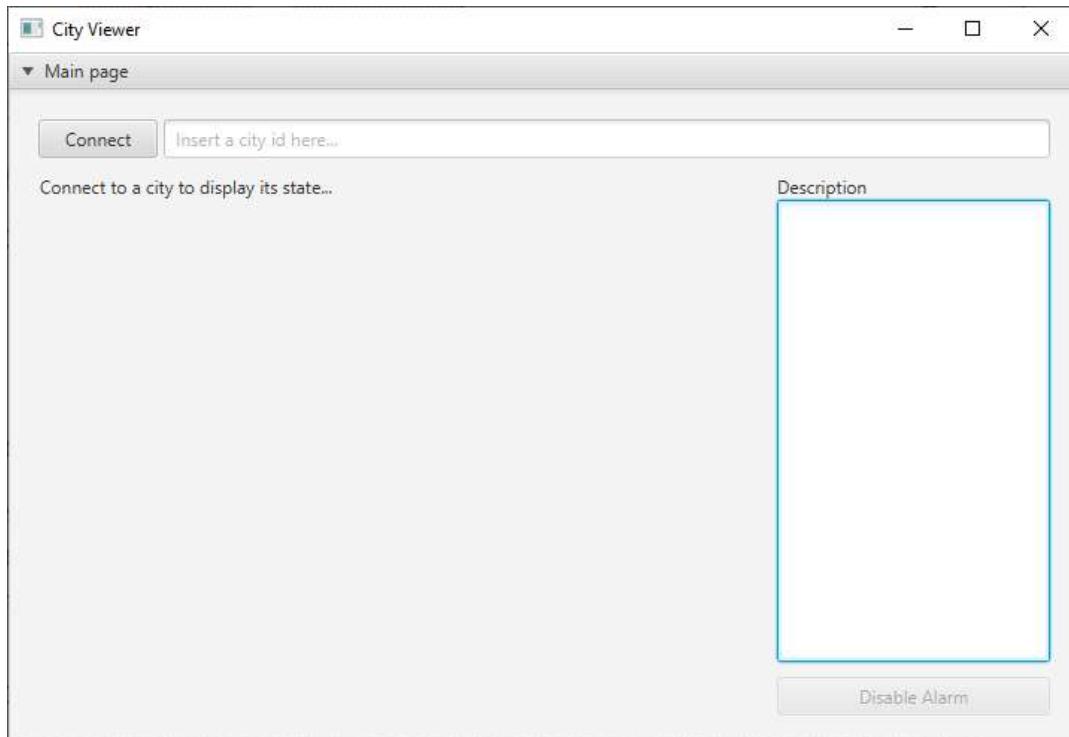


Figura 16. Schermata iniziale dell'interfaccia grafica dell'applicazione.

Alla pressione del pulsante Connect, il MainFXController crea un ViewActor che cerca di connettersi alla città con l'identificatore richiesto dall'utente. Se il tentativo di connessione dovesse fallire troppe volte, l'interfaccia grafica viene chiusa dal ViewActor.

Una volta connesso alla città, il ViewActor mantiene aggiornata l'interfaccia utente, mostrando a mano a mano gli snapshot della città (*Figura 17*).

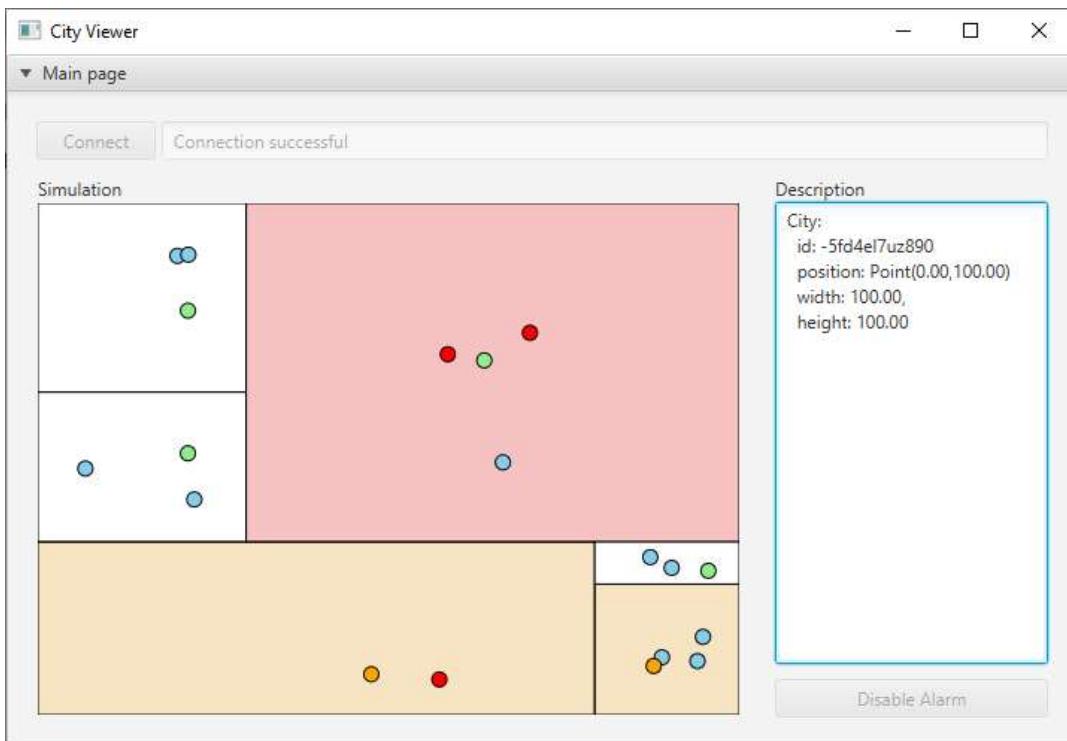


Figura 17. Una volta connessi a una città, l'interfaccia grafica ne mostrerà lo stato in evoluzione.

Cliccando su un componente della città, è possibile visualizzarne lo stato nell'area testuale sulla destra. In particolare, cliccando su un distretto monitorato da una caserma dei pompieri, sarà possibile disattivarne l'allarme manualmente (Figura 18), prima che l'intervento della caserma sia completato.



Figura 18. Cliccando su un distretto monitorato, è possibile disattivarne l'allarme manualmente.

2.3. Istruzioni all'uso

Per utilizzare l'applicazione, per prima cosa è necessario avviare il cluster. Per farlo, è possibile eseguire:

```
>: <project>/ex-02/src/main/scala/StartCluster
```

Una volta avviato il cluster, è possibile creare delle nuove città e registrarle al cluster. Per farlo, è possibile eseguire più volte:

```
>: <project>/ex-02/src/main/scala/CreateCity
```

Una volta creata, ogni città stampa a console il suo snapshot iniziale, contenente il suo identificatore insieme agli identificatori di tutti i suoi componenti.

Per visualizzare lo stato di una città in evoluzione, si può avviare l'interfaccia grafica dell'applicazione. Per farlo, è possibile eseguire:

```
>: <project>/ex-02/src/main/scala/Gui
```

Una volta avviata l'interfaccia grafica dell'applicazione, è necessario connetterla alla città che si vuole monitorare. Per farlo, è necessario conoscere l'identificatore della città, quindi è sufficiente inserirlo dove richiesto e cliccare il pulsante Connect dell'interfaccia grafica.

Da notare come sia possibile creare più città e più interfacce grafiche, connettendo diverse interfacce grafiche alla stessa o a diverse città.

Per configurare il sistema, è possibile modificare le impostazioni contenute nel seguente file:

```
<project>/ex-02/src/main/scala/configuration/C.scala
```

Inoltre, è possibile consultare i log del cluster al seguente percorso:

```
<project>/target/cluster-logging/cluster.log
```

2.4. Demo dell'applicazione

```
City created!
### City Actor Initialized ###

Snapshot:
CityData(id:-1uvw2igfcgdgb, position:Point(0.00,100.00), width:100.0, height:100.0)
ZoneData(id:14kq5spkeggr8, position:Point(71.99,36.45), width:28.01, height:14.16, state:Calm)
ZoneData(id:1a3psezu9k204, position:Point(25.50,100.00), width:74.50, height:29.76, state:Calm)
ZoneData(id:-130bnb551q7po, position:Point(71.99,22.29), width:28.01, height:22.29, state:Calm)
ZoneData(id:1qaymem8qlgfz, position:Point(25.50,70.24), width:74.50, height:33.79, state:Calm)
ZoneData(id:15ttlbpopbcl8, position:Point(0.00,100.00), width:25.50, height:63.55, state:Calm)
ZoneData(id:-z0iojfm1p2js, position:Point(0.00,36.45), width:71.99, height:36.45, state:Calm)
PluviometerData(id:-1oze0zqnaim2o, position:Point(70.16,79.90), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:-qsIuc7abtb88, position:Point(51.23,83.90), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:-1p0qca0maaabr, position:Point(16.91,56.33), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:-llhcxh9zaogb, position:Point(79.97,84.40), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:-1at3wuiodn7z, position:Point(85.07,12.61), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:1irg7c0kl1l7u, position:Point(47.83,17.38), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:n5k7o95n60c7, position:Point(80.91,5.83), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:brc0t19ihye, position:Point(40.45,61.34), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:o08kyrpcnsr0, position:Point(11.67,66.81), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:1mv5i4nk3i63y, position:Point(48.33,56.91), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:-nonvkrdj9t0l, position:Point(41.44,17.46), signal:false, measurement:0.00, threshold:0.90)
PluviometerData(id:lw2ogtb5mu4m, position:Point(86.72,28.02), signal:false, measurement:0.00, threshold:0.90)
FireStationData(id:-b9gkeqgr1th4, position:Point(78.54,77.75), state:Available)
FireStationData(id:1220zbk28ef4f, position:Point(9.52,49.89), state:Available)
FireStationData(id:-1bqzs84qjul8n, position:Point(42.58,16.66), state:Available)
FireStationData(id:19q2yrzyetx7w, position:Point(84.60,31.21), state:Available)
FireStationData(id:fkxihp3lw856, position:Point(83.58,51.11), state:Available)
FireStationData(id:rjni8bjt7vw, position:Point(84.89,8.46), state:Available)
```

Figura 19. Un esempio dello snapshot stampato da una città quando viene creata. L'identificatore della città è evidenziato in azzurro.

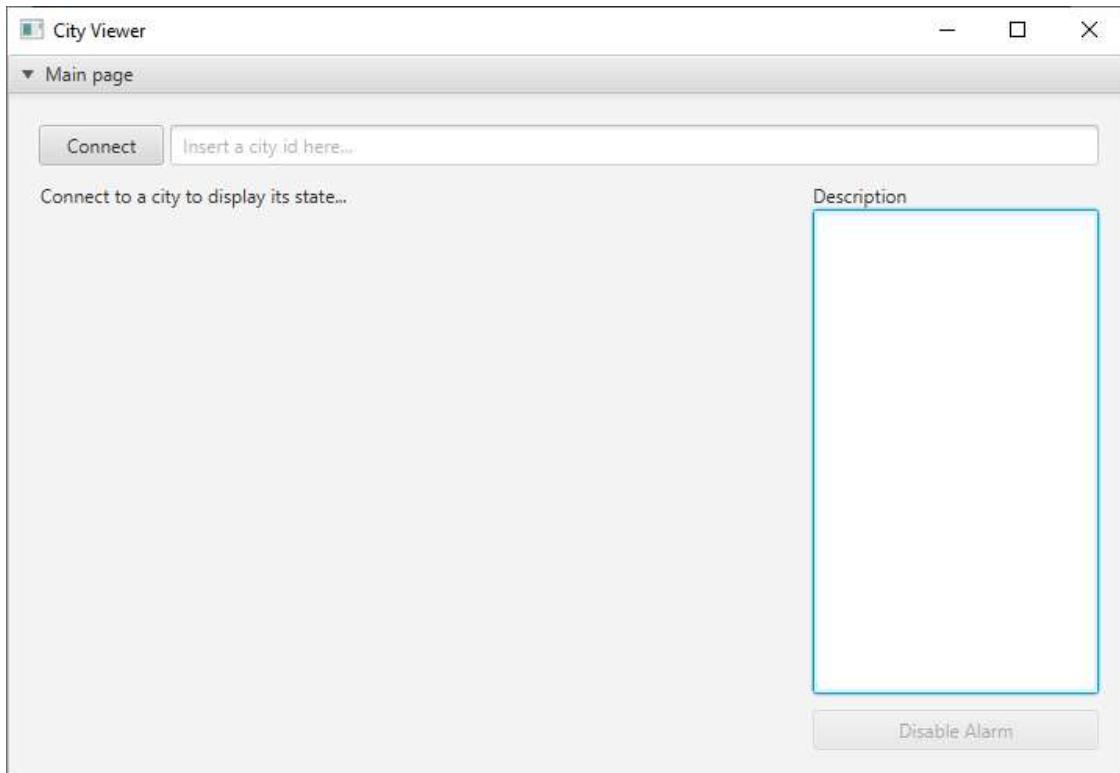


Figura 20. Schermata iniziale dell'interfaccia grafica dell'utente. In alto, il pulsante Connect e il riquadro dove è richiesto di inserire l'identificatore della città a cui ci si vuole connettere. A destra, il riquadro dove saranno mostrate le informazioni relative ai componenti della città quando cliccati. Subito sotto, il pulsante Disable Alarm, che permette di disabilitare l'allarme di un distretto monitorato da una caserma dei pompieri.

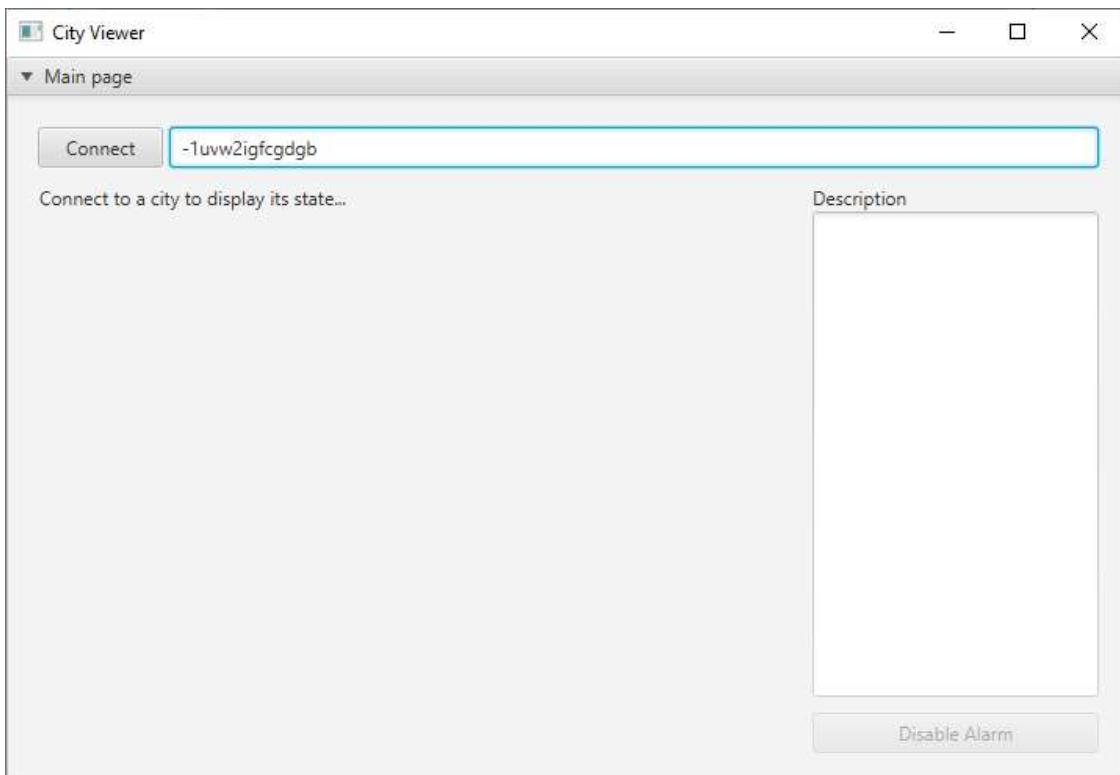


Figura 21. Come identificatore della città, deve essere inserito quello stampato a console dalla città quando viene creata.

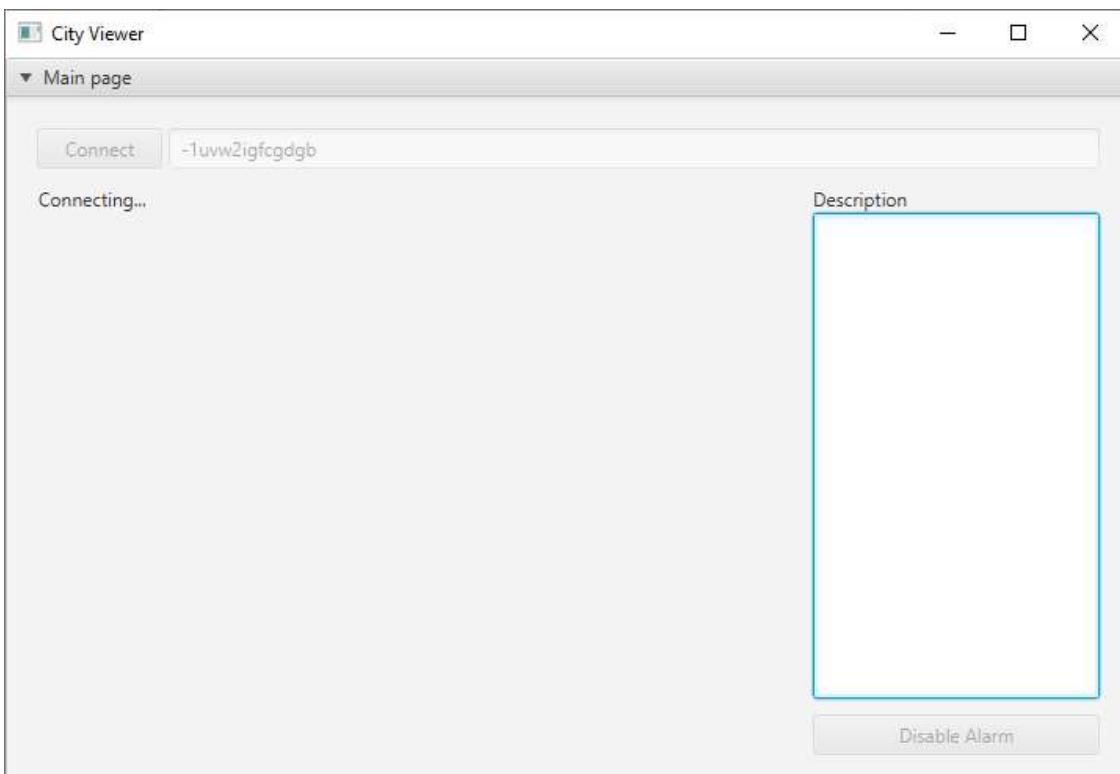


Figura 22. Una volta inserito l'identificatore della città, è possibile premere il pulsante Connect per connettersi a tale città e quindi cominciare a monitorarla.



Figura 23. Una volta connessi alla città, saranno visualizzati tutti i suoi componenti a schermo. I distretti hanno una forma rettangolare ed assumono un colore bianco, se in stato di quiete, un colore rosso, se in stato di allarme e un colore arancione, se monitorate da una caserma dei pompieri. Le caserme dei pompieri hanno una forma circolare ed assumono un colore verde, se disponibili, ed un colore arancione, se occupate a risolvere l'allarme di un distretto. I pluviometri hanno una forma circolare ed assumono un colore blu, se la loro misura è sotto la soglia consentita, ed un colore rosso, se la loro misura è sopra la soglia e quindi se emettono un segnale.



Figura 24. Cliccando su un componente della città è possibile osservarne lo stato in evoluzione.



Figura 25. Cliccando su un distretto monitorato da una caserma dei pompieri, è possibile disattivarne l'allarme manualmente, premendo il pulsante Disable Alarm.



Figura 26. Premendo il pulsante Disable Alarm, il distretto torna in stato di quiete prematuramente e la caserma dei pompieri che se ne stava occupando ritorna libera.