Assignment 3.1: sistema centralizzato ad attori

Sintesi del problema

Si voleva realizzare un sistema centralizzato, ad attori, che gestisse delle particelle in movimento su un piano. Il movimento di ogni particella è soggetto all'influenza delle forze delle altre. Si volevano gestire l'avvio, la pausa, il resume e lo stop della simulazione, nonché la possibilità di passare ad una modalità "step by step". Durante l'esecuzione doveva essere possibile aggiungere o rimuovere particelle dinamicamente.

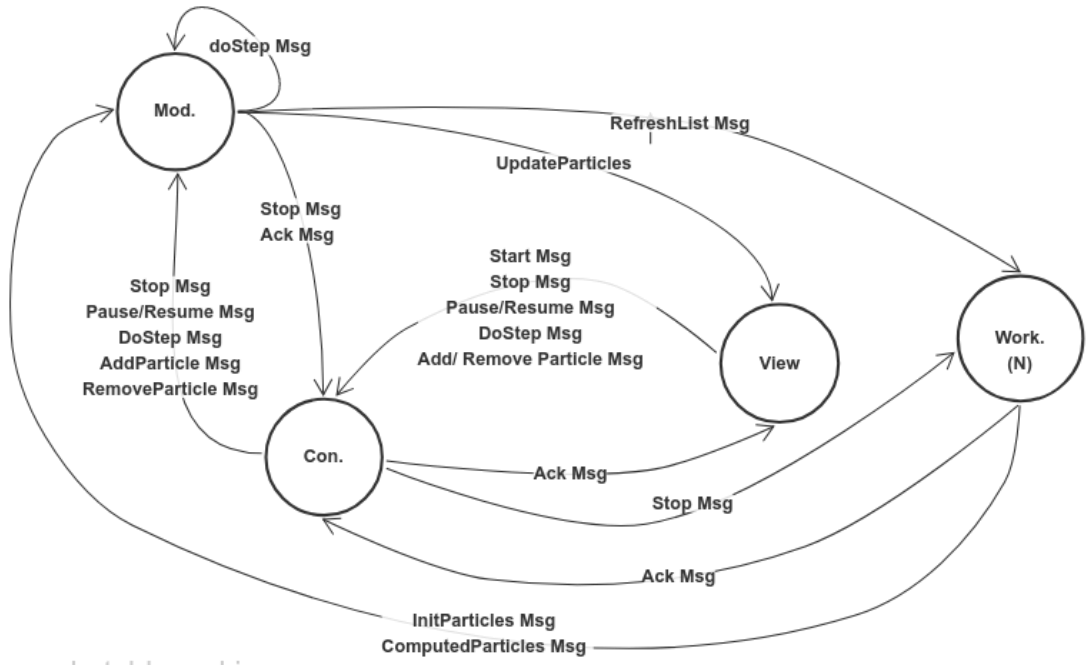
Analisi e progettazione

Durante la fase di analisi sono stati individuati i seguenti attori:

- **View Actor**: è l'attore che si occupa di comunicare al controller le informazioni provenienti dall'interfaccia grafica e di aggiornare quest'ultima in reazione ai messaggi ricevuti dal Model.
- **Model Actor**: è l'attore che si occupa della parte logica del programma. Comunica con i Worker Actor, delegandogli il lavoro di calcolo (distribuito in maniera equa ad ogni passo) e gestisce la logica degli step.
- **Controller Actor**: crea gli altri attori e comunica con il model, notificandogli gli eventi di Stop, Pause/Resume, aggiunta/rimozione di una particella e step (nel caso in cui si sia nella modalità step by step)
- **Worker Actor**: sono N attori che si occupano di calcolare le nuove posizioni delle particelle. Inoltre, nella fase iniziale, essi generano casualmente le particelle di partenza.

La scelta degli attori poteva essere più granulare, ad esempio si poteva modellare come attore la singola particella, tuttavia si è deciso di optare per questa soluzione in maniera tale da non aggiungere troppa complessità ed evitare un eccessivo overhead.

Lo schema seguente mostra le interazioni tra gli attori, evidenziando i messaggi che essi si scambiano. Si ha un'istanza per ogni attore, eccezion fatta per i WorkerActor che sono in numero variabile.

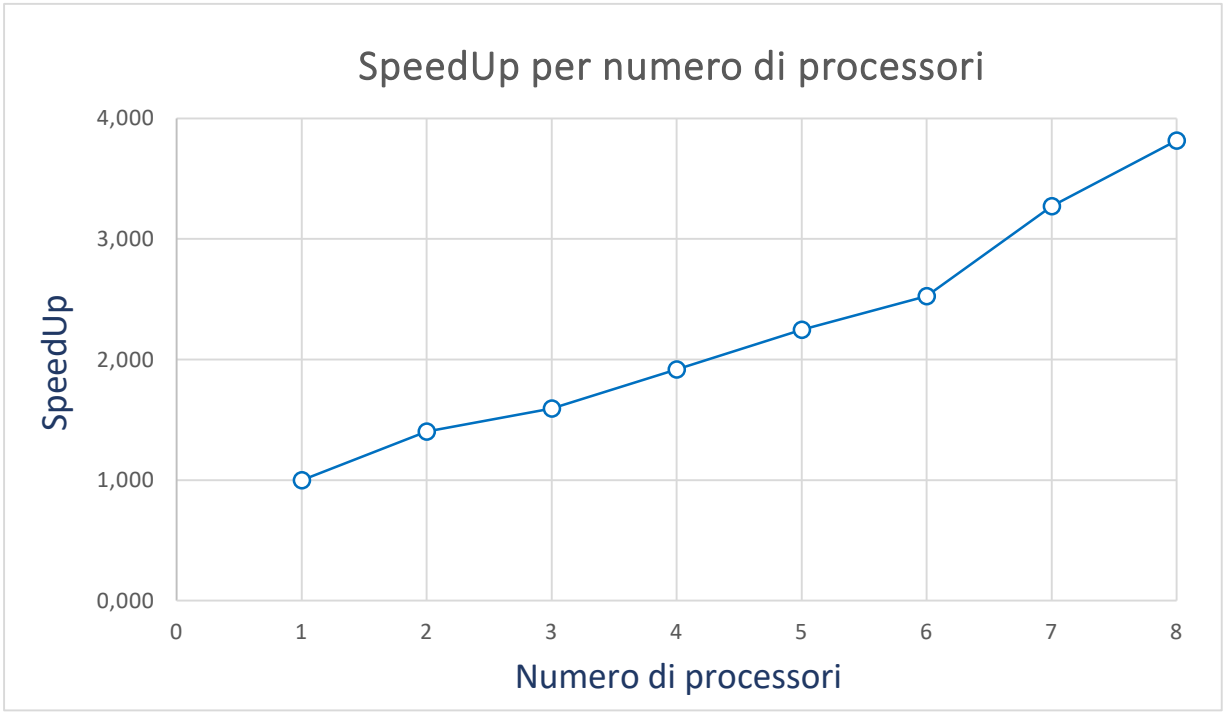


Breve descrizione dei messaggi:

- **Start Msg:** contiene le informazioni necessarie al setup iniziale del programma, come ad esempio il numero di particelle e il numero di passi
- **InitiParticlesMsg:** contiene i valori relativi alle particelle, generati in maniera casuale dagli WorkerActor in fase di pre-start.
- **DoStep Msg:** la ricezione di questo messaggio (vuoto), innesca nel ModelActor l'esecuzione di un nuovo step. In modalità continua, il Model Actor lo invia a sé stesso alla fine di ogni step, mentre in modalità step by step viene inviato dal ViewActor dopo che è stato premuto il tasto "nex step" sulla GUI.
- **ComputedParticles Msg:** contiene la lista delle particelle assegnate ad un worker, con i valori aggiornati. Viene inviata da ciascun Worker Actor al termine di ciascuna computazione.
- **RefreshListMsg:** contiene la lista di tutte le particelle attualmente presenti, sulla base della quale il Worker Actor ricevente dovrà calcolare le nuove posizioni della porzione di particelle di sua competenza.
- **Pause/Resume Msg:** è un messaggio vuoto che ripristina il programma se questo è in pausa e lo mette in pausa in caso contrario, facendolo passare alla modalità step by step.
- **Add Particle Msg:** innesca l'aggiunta di una nuova particella, i cui dati vengono generati casualmente. La particella sarà visibile allo step successivo.
- **Remove Particle Msg:** rimuove l'ultima particella creata. La rimozione è effettiva dallo step successivo.
- **Stop Msg – Ack Msg:** sono messaggi vuoti che vengono utilizzati per terminare l'applicazione. L' Ack Msg serve al Controller Actor per capire quando tutti hanno ricevuto il messaggio di stop

Valutazione delle performance

Questa soluzione è molto più lenta se paragonata alla soluzione presentata per il primo assignment, anche se risulta essere più versatile. Inoltre il paradigma ad attori rende il progetto di più facile realizzazione. Il grafico seguente mostra lo speedup.



Assignment 3.2: sistema distribuito ad attori con Akka

Sintesi del problema

Si vuole realizzare un sistema distribuito denominato MapMonitor, che gestisce un numero variabile di **sensori** dinamici che si spostano su una mappa e possono scomparire, apparire o uscire dai confini della mappa. La mappa è suddivisa in sotto-aree denominate **patch**. Ad ognuna di esse sono assegnati dei **guardiani**, che si occupano di calcolare la media dei valori rilevati dai sensori. Una media superiore a una determinata soglia innesca (se dura almeno k secondi), uno stato di **preallerta** nel guardiano. Se la maggior parte dei guardiani è in preallerta allora la patch e i suoi guardiani passano in uno stato di **allerta**, che può essere ripristinato dall'utente, tramite l'interfaccia grafica.

Analisi e progettazione

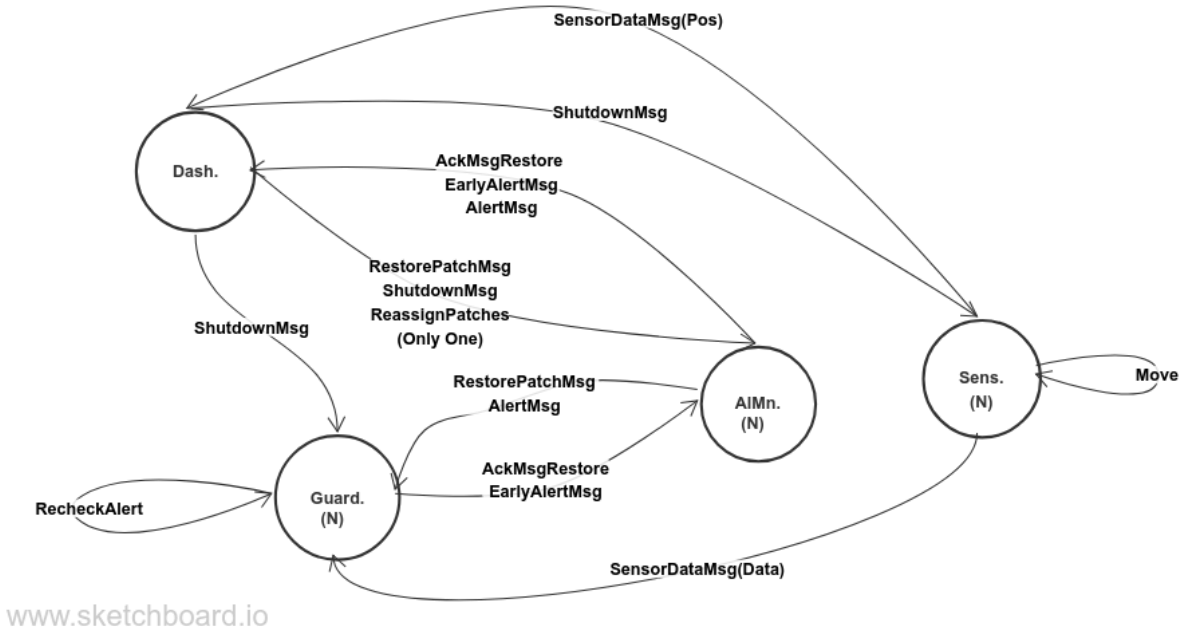
L'analisi del problema ha portato alla progettazione di un sistema in cui il cluster è composto da nodi, i quali Actor System consistono in un unico attore, creando così una corrispondenza 1:1 tra nodo e attore. Per l'interazione tra i vari nodi, nell'implementazione, si è fatto largo uso del modello **publish-subscribe**.

I ruoli ricoperti dai nodi sono i seguenti:

- **Sensor:** i nodi con ruolo di sensori sono composti da un attore "Sensor" che gestisce in autonomia il proprio spostamento e la generazione del valore V(A). Lo spostamento, di una quantità pari a STEP_SIZE, è gestito mediante l'auto-invio di un **Move Msg** contenente una direzione generata casualmente, effettuato ogni MOVING_RATE millisecondi. La rilevazione di V(A) viene invece innescata mediante l'auto-invio di un **DetectNewVal Msg** schedulato ogni DETECTING_RATE millisecondi. I sensori conoscono solo le priopri coordinate; non hanno concezione delle patch e dei loro confini, di conseguenza non sanno quali sono i guardiani a cui devono rivolgersi. Per questo motivo ad ogni rilevazione i sensori, mediante un **SensorData Msg**, pubblicano il loro valore, che verrà poi recuperato dai guardiani interessati, i quali distinguono i messaggi di loro competenza in base alla posizione del sensore (anch'essa inclusa nel messaggio).
- **Guardian:** i guardiani ricevono gli **SensorData Msg** dai sensori e, se le coordinate del mittente appartengono alla propria patch, le aggiungono ad una lista di valori. Periodicamente viene fatta una media di tali valori (innescata dalla ricezione di un **ComputeAvg Msg**, auto-inviato) e, nel momento in cui questa supera una certa soglia, il guardiano accende un flag che indica la possibilità di entrare in preallerta e fa in modo di auto-inviarsi, dopo un tempo k, un **ReCheckAlert Msg**. Se, alla ricezione di tale messaggio, il flag è ancora acceso, entra definitivamente in pre-allerta e lo comunica agli AlertManager. Se in un qualunque momento precedente alla ricezione del ReCheckAlert Msg, il guardiano calcola una media inferiore alla soglia, il flag viene spento.
- **Dashboard:** la dashboard è un nodo seed e si occupa di gestire gli eventi scatenati dall'utente attraverso la GUI e l'aggiornamento dei dati ricevuti dagli altri nodi (la nuova posizione di un sensore, l'allerta di una patch, la preallerta di un guardiano…)
- **AlertManager:** gli AlerManager si occupano di gestire le patch, tenendo traccia per ognuna dello stato e del numero di guardiani in preallerta. Ad ogni AlertManager è assegnata una lista di patch. Quando un AlertManager riceve (da un guardiano) un messaggio di tipo **NotifyAlert Msg**, con il campo "alertState" impostato a "**EARLY_ALERT**", apprende che quel determinato guardiano è andato in preallerta e, se esso si trova in una delle patch di sua competenza, provvede ad aggiornare la lista dei guardiani in preallerta relativi a quella patch. Quando si accorge che più della metà dei guardiani di una patch sono in preallerta, invia un messaggio di tipo **NotifyAlert Msg**, con il campo "alertState" impostato a "**ALERT**" ai guardiani di quella patch, affinché vadano in allerta.

L'alert manager può inoltre ricevere un messaggio di tipo **NotifyAlert Msg**, con il campo "alertState" impostato a "**OK**" dalla Dashboard, nella circostanza in cui l'utente voglia ripristinare lo stato iniziale di una patch in allerta.

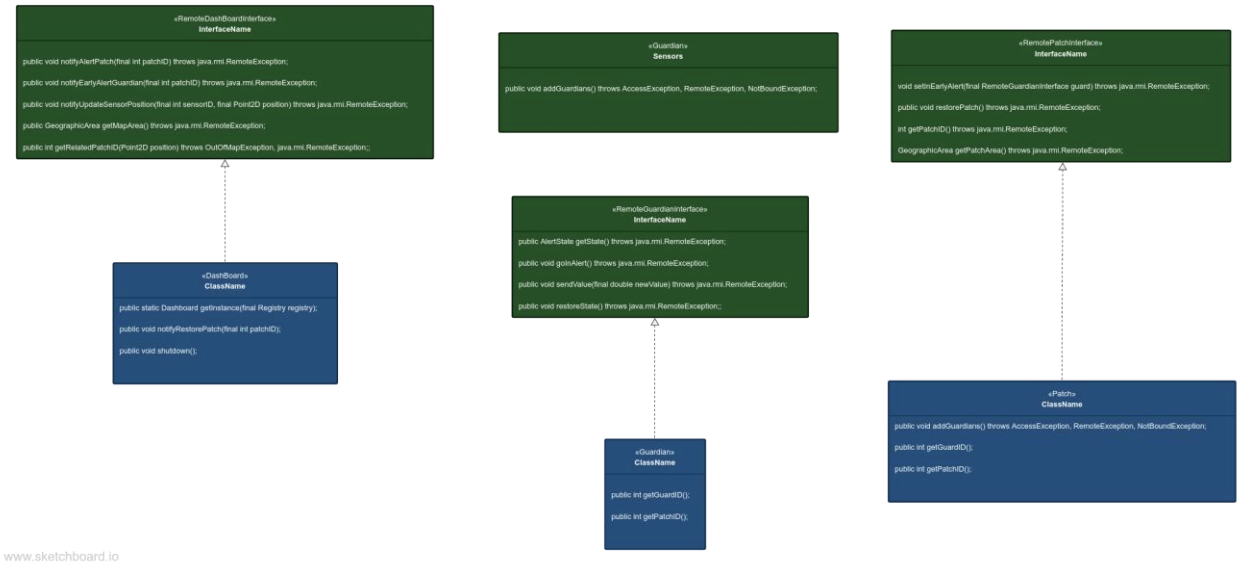
Il seguente schema illustra quelle che sono le interazioni tra i vari nodi:



Fault tollerance

Si è cercato, per quanto possibile, di gestire la dinamicità dei nodi e i casi in cui un nodo, per qualche motivo, smetta di essere reperibile. Le soluzioni trovate, distinte per tipo di nodo, sono le seguenti:

- **Guardian:** la scomparsa o la ricomparsa di un nodo guardiano non influiscono sul funzionamento del sistema. Un guardiano appena entrato riceverà comunque i messaggi dei sensori, poiché questi li inviano in broadcast a tutti i guardiani. Nemmeno gli AlertManager risentono del cambiamento dinamico dei guardiani, poiché ogni volta, per capire se i guardiani in preallerta sono più della metà, usano il numero esatto, prelevandolo dalla lista dei membri. La perdita dei messaggi di notifica della preallerta, inviati agli AlertManager, invece, può avere un'influenza negativa: se vengono persi i messaggi di preallerta di almeno la metà dei guardiani, la patch non va in allerta, anche se avrebbe dovuto.
- **Sensor:** il cambiamento del numero di sensori attivi non influisce sul funzionamento del sistema, né lo fa la perdita dei messaggi inviati da questi ultimi, poiché se ne inviano talmente tanti che la mancanza di alcuni non pregiudica il risultato finale.
- **AlertManager:** ogni AlertManager si occupa di un determinato numero di Patch. Se un nodo AlertManager diventa irraggiungibile, le patch che erano di sua competenza vengono redistribuite tra gli altri AlertManager e il loro stato viene resettato, con l'assunzione che se una patch era in allerta o dei guardiani in preallerta, molto probabilmente la situazione si riverificherà.
- **Dashboard:** la dashboard, purtroppo, è un point of failure. Si è pensato a un solo nodo dashboard, la cui caduta implicherebbe l'impossibilità, per l'utente, di comunicare con il sistema.



Valutazione delle performance

Questa soluzione è decisamente più lenta di quella sviluppata con JavaRMI, ma risulta essere più robusta.

Assignment 3.3: sistema distribuito con Java RMI

Sintesi del problema

Il problema posto è il medesimo dell'esercizio precedente, con la differenza che questa volta si richiede di utilizzare Java RMI.

Analisi e progettazione

L'analisi del problema ha portato a individuare le seguenti entità principali:

- **Sensor:** un sensore ha al suo interno istanze degli oggetti remoti Dashboard e Guardian. Non offre anch'esso un'interfaccia remota poiché nessuna delle altre entità ha bisogno di chiamare metodi su di esso. Il sensore, come nell'esercizio precedente, è autonomo nello spostamento e nella generazione del valore V(A), entrambe azioni che fa periodicamente. Quando rileva un nuovo valore esso lo notifica ai guardiani che ha al suo interno, chiamando su di essi il metodo sendValue(). Ad ogni spostamento, il sensore, notifica alla Dashboard la nuova posizione chiamando su di essa il metodo notifyUpdateSensorPosition() e controlla se è uscito dai confini della propria patch, o , addirittura, da quelli della mappa. Se ha cambiato patch, chiede alla dashboard l'id della nuova patch, mediante il metodo getRelatedPatchID(), tramite esso si fa dare dal registry la nuova patch e aggiorna di conseguenza i suoi campi interni
- **Guardian:** i guardiani hanno al loro interno l'oggetto remoto Patch, con cui interagiscono chiamando, al momento opportuno, il metodo setInEarlyAlert() che serve a notificare il proprio stato di preallerta.
- **Patch:** la patch tiene al suo interno dashboard e guardiani, con cui interagisce per comunicare il suo stato di allerta.
- **Dashboard:** la dashboard ha al suo interno le patch e interagisce con esse con il metodo restorePatch() nel momento in cui l'utente decide di resettare una patch in allerta.

Performance e speedup

Le performance registrate con questo approccio sono decisamente superiori a quelle ottenute con l'approccio precedente, sacrificando però la resilienza.