UNIVERSITÀ DI BOLOGNA

PCD1819 Assignment 3 - Smart Positioning and Map Monitor

Bombardi Tommaso, Mascellaro Maria Maddalena, Ragazzi Luca

June 15, 2019

1 ANALISI DEL PROBLEMA

1.1 Esercizio 1: Smartpositioning

Per quanto riguarda il primo esercizio, nella fase di analisi del problema sono state riprese le valutazioni fatte nel primo assignment. Viste le specifiche aggiuntive, oltre ai vincoli precedentemente definiti, dovranno essere rispettate altre condizioni. In particolare:

- La simulazione dovrà avere una modalità passo passo, e quindi poter essere messa in pausa e poter eseguire uno step per volta. A livello di progettazione, la simulazione potrebbe essere considerata come un insieme di passi (step) che possono essere eseguiti sia insieme sia singolarmente, ma sempre secondo il loro ordine predefinito.

- Dovrà essere possibile aggiungere e togliere corpi dinamicamente durante l'esecuzione della simulazione (anche quando questa è in pausa). Dato che la posizione di una particella influenza la posizione delle altre nello step successivo, un corpo non potrà essere aggiunto o rimosso dinamicamente durante il calcolo di posizione e velocità delle particelle in un determinato step. Lo step rappresenta infatti "un'istantanea" ad un determinato tempo t in cui, se il corpo è presente, deve obbligatoriamente essere considerato da tutte le particelle presenti nella simulazione (e non solo da alcune).
  Allo stesso tempo il sistema dovrà tenere traccia di tutte le richieste dell'utente (aggiunta o rimozione), anche se queste avvenissero durante il calcolo di uno step, e accoglierle non appena possibile, ossia al termine del calcolo delle posizioni delle particelle.

1.2 Esercizio 2, Esercizio 3: MapMonitor

Per quanto riguarda il secondo e il terzo esercizio, dall'analisi sono emersi i seguenti vincoli (che dovranno essere rispettati per ottenere un sistema distribuito conforme alle specifiche):

- Trattandosi di un sistema distribuito, dal punto di vista del deployment non potranno essere fatte assunzioni sulla locazione delle varie entità (sensori, guardiani e dashboard) all'interno della rete: due qualsiasi entità potrebbero essere in esecuzione sia su nodi

distinti della rete sia sullo stesso nodo. Per questo motivo il sistema dovrà essere considerato come un insieme di nodi all'interno di una rete, e all'interno di ognuno di essi potranno essere in esecuzione una o più entità.

- Dato che il deploy delle varie entità dovrà poter essere eseguito su nodi diversi, sarà necessario non avere memoria condivisa tra le varie entità del sistema (ogni entità dovrà fare riferimento soltanto alla propria memoria e ai messaggi ricevuti dalle altre). Per facilitare l'esecuzione del programma, potrebbe essere utile fornire la possibilità di configurare il sistema specificando i nodi su cui saranno eseguite le varie entità.

- Le varie entità presenti nel sistema saranno distribuite nella rete e potrebbero essere soggette a malfunzionamenti. Per questo motivo, in entrambe le implementazioni del sistema, sarà necessario che le altre entità del sistema riconoscano eventuali guasti o irraggiungibilità causate dalla rete. Ad esempio, se un sensore risulta essere irraggiungibile all'interno della rete, il suo valore non dovrà essere più considerato dai guardiani del patch in cui il sensore si trovava l'ultima volta che ha comunicato con il sistema.

- Non potendo fare assunzioni sul corretto funzionamento delle altre entità presenti nel sistema, ogni entità (sensori, guardiani e dashboard) dovrà essere autonoma e funzionare correttamente indipendentemente dallo stato delle altre.

2 Descrizione della soluzione proposta

2.1 Esercizio 1: Versione di Smartpositioning basata sul modello ad attori

Per quanto riguarda il primo esercizio la soluzione è costituita dai seguenti package: *smartpositioning.common*, che contiene alcuni sorgenti comuni all'implementazione del primo assignment, *smartpositioning.actors*, in cui si trovano il main dell'applicazione e gli attori realizzati, *smartpositioning.messages*, contenente i messaggi scambiati dai vari attori, e *smartpositioning.performance*, in cui sono state inserite una versione semplificata del programma e una semplice implementazione sequenziale (entrambe usate per le prove di performance).
Riprendendo quanto fatto nel primo assignment, si è scelto di mantenere una logica master-worker nell'applicazione ed è stato quindi creato un attore (*ActorManager*) incaricato di gestire la simulazione. Esso delega le operazioni di calcolo sulle varie particelle ad un insieme di worker (*ActorCalculator*), che lui stesso ha il compito di creare. Dato che il modello ad attori disaccoppia la concorrenza logica da quella fisica e che un attore non ha bisogno del proprio thread fisico di controllo (è possibile eseguire un elevato numero di attori su un numero limitato di thread), si è scelto di creare un *ActorCalculator* per ogni particella presente nella simulazione. Gli attori, ovviamente, comunicano tramite scambio di messaggi.
Nell'applicazione è stato inoltre inserito un attore *ActorViewer*, anch'esso creato dall'*ActorManager*, che si occupa di aggiornare la view e consente di rispettare il frame rate (vengono mostrati N step ogni secondo, dove N è una costante predefinita). In questo modo la simulazione riesce ad essere "realistica" e la sua velocità, se viene scelto un frame rate adeguato, non dipende dalla potenza computazionale della macchina su cui è in esecuzione il programma.
Quando si lancia l'applicazione viene istanziato un semplice *Controller*, che crea la *View* e si occupa di gestire gli input dell'utente (click sui pulsanti start, one step, pause, stop e remove e click su un punto qualsiasi del campo di gioco). Quest'ultimo consente all'utente di aggiungere una particella alla simulazione, mentre attraverso il pulsante remove può essere rimosso un corpo (in particolare, viene rimossa la prima particella che era stata inserita nella lista).
Una volta premuto start, il *Controller* istanzia un sistema di attori (se non è ancora stato creato)

e al suo interno esegue un *ActorManager*, che a sua volta istanzia un *ActorViewer* e un *ActorCalculator* per ogni particella presente nella simulazione. I vari input dell'utente vengono comunicati dal *Controller* all'*ActorManager* attraverso i seguenti messaggi: *SimulationStartMsg*, usato quando viene avviata la simulazione, *SimulationPauseMsg*, per mettere in pausa la simulazione, *SimulationAddMsg* e *SimulationRemoveMsg*, usati per comunicare rispettivamente l'aggiunta e la rimozione di una particella. La richiesta di eseguire un singolo step viene comunicata con due messaggi: *SimulationStartMsg* seguito da *SimulationPauseMsg*.
In presenza del click sul pulsante stop, *ActorManager* viene stoppato e quest'ultimo si occupa di stoppare *ActorViewer* e i vari *ActorCalculator*. La medesima cosa avviene quando gli step della simulazione sono terminati e, in questo modo, quando la simulazione non è attiva (può essere soltanto lanciata una nuova simulazione) nessun attore sarà in esecuzione.
Come detto in precedenza, l'*ActorManager* è quello che si occupa di gestire la logica della simulazione. Per questo attore sono stati previsti due comportamenti: *NormalBehaviour*, che costituisce il normale comportamento dell'attore, e *ComputingBehaviour*, ossia il comportamento dell'attore durante il calcolo della posizione delle particelle in un determinato step.
Quando si trova in *NormalBehaviour*, l'*ActorManager* gestisce i messaggi di input dell'utente (*SimulationStartMsg*, *SimulationPauseMsg*, *SimulationAddMsg* e *SimulationRemoveMsg*) e il messaggio che indica il completamento di uno step (*UpdatedStepMsg*). Quando inizia la computazione di uno step, il comportamento dell'attore cambia in *ComputingBehaviour* e l'*ActorManager* invia un messaggio a tutti i worker (*ActorCalculator*) con cui chiede ad essi di calcolare la nuova posizione di una particella (nel messaggio viene passata una lista di backup). Una volta che tutte le particelle sono state aggiornate (e quindi lo step è stato completato), l'*ActorManager* invia un messaggio all'*ActorViewer* richiedendo l'aggiornamento della view e torna ad assumere il suo comportamento normale. Nell'*ActorManager* è stato utilizzato lo stash (estendendo la classe *AbstractActorWithStash*), che consente all'attore di bufferizzare dei messaggi che non devono essere gestiti usando il comportamento corrente ma che non devono neanche essere ignorati. Tutti i messaggi inattesi ricevuti durante il *ComputingBehaviour* vengono messi nello *stash* e recuperati non appena il comportamento torna normale (con la funzione *unstashAll()*), in modo da considerare anche gli input dati dall'utente durante il calcolo di uno step.
Nella figura di seguito è stato rappresentato il comportamento dell'*ActorManager* attraverso uno statechart, in cui ogni stato rappresenta un comportamento dell'attore e ogni messaggio un evento (che genera una particolare transizione).

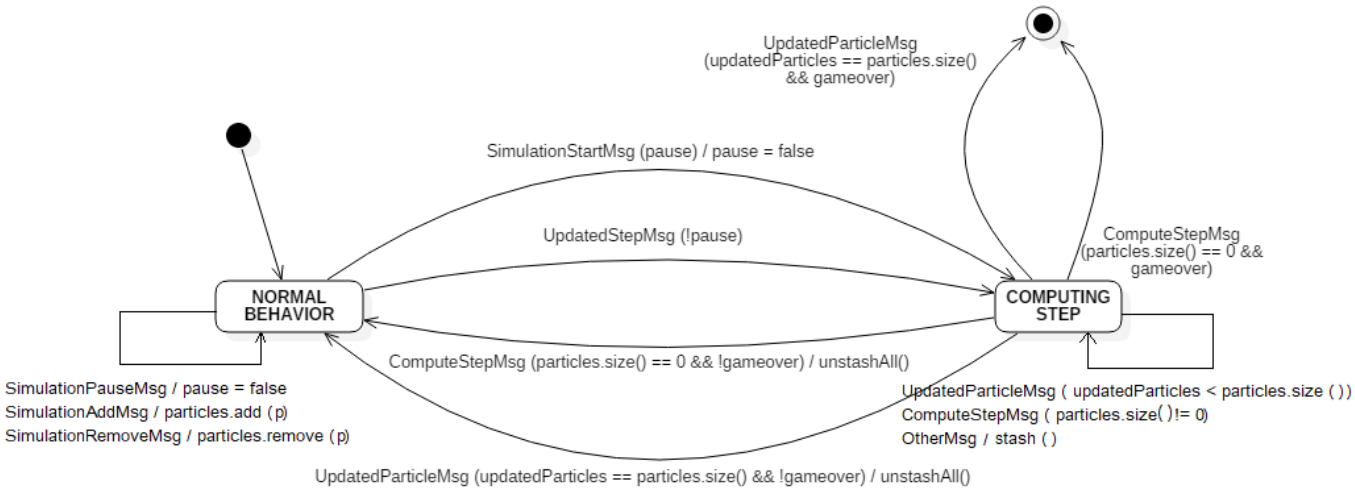

Figure 2.1: Statechart che rappresenta il comportamento dell'*ActorManager*

Per quanto riguarda gli altri attori realizzati, l'*ActorCalculator* è un semplice worker che reagisce ad un solo tipo di messaggio (*ComputeParticleMsg*, in cui sono presenti la lista di backup delle particelle e l'indice della particella da aggiornare) rispondendo al mittente con la particella aggiornata (inviata tramite un messaggio *UpdatedParticleMsg*).
Infine, l'*ActorViewer* si occupa di aggiornare la *View* in accordo con i messaggi ricevuti dall'*ActorManager*. *ActorViewer* reagisce a messaggi di tipo *ShowChangesMsg* e *ShowStepMsg*, che contengono entrambi uno snapshot delle particelle da mostrare, ma il primo è generato dall'aggiunta/rimozione di una particella mentre il secondo si ha quando viene calcolato un nuovo step.
Per mantenere costante il numero di step mostrati nell'unità di tempo (ogni secondo il programma mostra un numero di step pari all'unità di tempo divisa per il frame rate), è stato utilizzato un *AbstractActorWithTimer*. In particolare gli snapshot ricevuti vengono mostrati immediatamente ma, se lo snapshot è ricevuto in seguito al calcolo di uno step, viene inviato un messaggio all'*ActorManager* soltanto quando "arriva il tick" del timer.
Questa scelta, pur bloccando l'*ActorManager* (e di conseguenza gli *ActorCalculator*) in attesa del frame rate, ha consentito di ottenere una buona reattività relativamente all'aggiunta e alla rimozione di particelle: se sopraggiunge una richiesta di aggiunta/rimozione questa sarà considerata al completamento dello step corrente mentre, se l'*ActorManager* avesse continuato con il calcolo degli step successivi, la particella sarebbe stata visibile soltanto dopo aver mostrato tutti gli step calcolati prima dell'input (questo avrebbe causato un comportamento poco reattivo del programma, specialmente nel caso fosse stato scelto un frame rate basso).

2.2 Esercizio 2: Versione di MapMonitor basata sul modello ad attori

Per quanto riguarda il secondo esercizio la soluzione è costituita dai seguenti package: *mapmonitor.common*, che contiene alcuni sorgenti comuni all'implementazione del terzo esercizio (*View* relative alle varie entità e alcune classi di utility), *mapmonitor.actors*, in cui si trovano gli attori realizzati e tre main che consentono di lanciare rispettivamente sensori, guardiani e dashboard, e *mapmonitors.messages*, contenente i messagi scambiati dai vari attori.
Per utilizzare il modello ad attori nel distribuito si è scelto di utilizzare *Akka Clustering*, che fornisce la possibilità di creare un cluster all'interno del quale uno o più nodi sono collegati tramite un servizio di membership. In particolare, in *Akka Clustering* un nodo è definito da "hostname:port:uid" ed è costituito da un actorsystem che si è unito al cluster. Per creare un cluster vanno specificati uno o più seed node, che consentono di gestire le richieste di join (ossia di partecipazione) al cluster. Specificando gli stessi seed node per ogni nodo del cluster, il nodo potrà unirsi (join) al cluster e comunicare con gli altri nodi del cluster.
Nel nostro caso, abbiamo scelto di creare un cluster a cui si unissero tutte le entità del sistema.
Il clustering fornisce un altro importante servizio che è stato utilizzato in questo esercizio, ossia il protocollo di gossip. Esso permette ai nodi del cluster, in caso di un'opportuna sottoscrizione, di ricevere informazioni riguardanti lo stato dei vari membri del cluster. Sfruttando il gossip è infatti possibile, per esempio, rilevare quando un membro si unisce al cluster oppure quando un altro membro viene rimosso dal cluster o risulta essere irraggiungibile.
Due attori in esecuzione all'interno dello stesso cluster possono comunicare, ovviamente tramite scambio di messaggi, se uno sa in quale nodo è in esecuzione l'altro. Nel nostro caso questo non è sempre possibile e, perciò, si è scelto di utilizzare il *Distributed Publish Subscribe*. Esso fornisce un attore di tipo *DistributedPubSubMediator*, che viene avviato su tutti i membri del cluster e attraverso il quale è possibile inviare messaggi a tutti gli attori del cluster che hanno registrato interesse per un argomento (senza sapere su che nodo sono in esecuzione).
Utilizzando *Akka Clustering*, sfruttando il protocollo di gossip e il *DistributedPubSubMediator*, abbiamo implementato tre tipologie di attori (una per ogni entità definita nelle specifiche). Essi saranno eseguiti all'interno del cluster e saranno in grado di comunicare tra loro.

Innanzitutto è stato implementato *ActorSensor*, l'attore che rappresenta un sensore del sistema. Si tratta di un attore che estende la classe *AbstractActorWithTimers*, poiché sfrutta le funzionalità messe a disposizione dai timer per gli attori di Akka. Per simulare l'andamento di un sensore, che si muove all'interno di una mappa e il cui valore cambia periodicamente, sono state settate casualmente una posizione e un valore (scelti rispettivamente all'interno della mappa e all'interno del range predefinito di valori). Ogni volta che il timer interno al sensore invia ad esso un messaggio, il valore e la posizione vengono aggiornate secondo un andamento regolare in modo da simulare il comportamento di un sensore. Quando un sensore esce dalla mappa, viene fatto tornare all'interno di essa per "non perdere sensori" durante la simulazione, ma a livello di implementazione è gestito anche il caso in cui un sensore esca dalla mappa (se si trattasse di un sensore reale, potrebbe uscire dalla mappa, rimanere fuori per un certo tempo, rientrare successivamente e il sistema si comporterebbe come previsto).
L'*ActorSensor*, ad ogni tick del timer, aggiorna infatti la sua posizione ed il suo valore. Successivamente calcola il patch in cui si trova e, se è diverso dal patch relativo alla posizione precedente, comunica l'uscita dal patch in cui si trovava inviando un messaggio sfruttando il *DistributedPubSubMediator* e specificando come topic "SensorOutFromPatch" seguito dal numero del patch interessato. A questo topic saranno iscritti tutti i guardiani associati a quel patch, che riceveranno il messaggio. Se il sensore esce dalla mappa, dovrà inviare un messaggio a tutte le dashboard per comunicarlo. Se invece rimane all'interno della mappa, dovrà inviare la sua posizione a tutte le dashboard e il suo valore ai guardiani del patch. Anche in questi casi, il sensore riuscirà ad inviare messaggi a particolari entità (dashboard o guardiani di un patch) usando il *Distributed Publish Subscribe* e specificando un topic adatto.
Per poter fare quanto appena descritto, il sensore dovrà essere in grado di determinare il patch in cui si trova. Questo è possibile poiché, alla sua creazione, vengono indicate le dimensioni della mappa (valore minimo/massimo delle ascisse e valore minimo/massimo delle ordinate) e le costanti M ed N e si assume che i patch siano sempre numerati da sinistra verso destra e dall'alto verso il basso. Con queste assunzioni, risulta essere semplice per un sensore determinare il patch in cui si trova. Si mostra quindi una semplice mappa in cui M=3 e N=3.

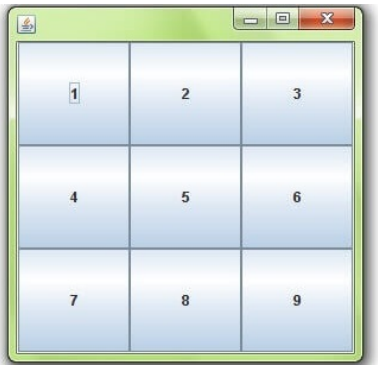

Figure 2.2: Suddivisione dei patch che mostra com'è strutturata una mappa in cui M=3 e N=3

Come già accennato in precedenza descrivendo il sensore, è stato realizzato anche un *ActorGuardian*. Esso ha memorizzato internamente i valori dei sensori, all'interno di *Map<String, Map<ActorRef, Double»*. In questa mappa la stringa rappresenta il nodo del cluster (hostname e porta), il riferimento all'attore identifica l'*ActorSensor* e il double il valore. Per ogni nodo del cluster sono infatti presenti uno o più attori e ad ognuno di essi è associato il proprio valore. Questa struttura dati consente di rimuovere sia un singolo attore, ad esempio quando esce dal patch, sia tutti gli attori associati ad un determinato nodo del cluster.
Per l'*ActorGuardian* sono stati previsti tre comportamenti: *NormalBehavior*, *CheckBehavior* e *DangerBehavior*. All'interno del *NormalBehavior*, l'attore reagisce ai messaggi che generano

un cambiamento dei valori associati ai sensori: *DetectedValueMsg*, *DetectedOutMsg*, *MemberRemoved* e *UnreachableMember*. In seguito ad ognuno di essi, vengono modificati i valori associati ai sensori e viene chiamata una funzione *evaluateSensorValues()* che ha il compito di valutare la media dei valori dei sensori. Se questa rimane sopra ad una certa soglia per un determinato numero di secondi (verificato utilizzando un timer), il guardiano andrà in pre-allerta e invierà un *ConsensusWarningMsg* a se stesso e a tutti gli altri guardiani dello stesso patch.
La ricezione di questo messaggio farà in modo che il guardiano cambi comportamento passando da *NormalBehavior* a *CheckBehavior*. Nel *CheckBehavior* il guardiano dovrà verificare se la maggioranza dei guardiani associati allo stesso patch sono in pre-allerta da un certo numero di secondi. Per farlo si è scelto di utilizzare l'algoritmo del crash model, che è un algoritmo di consenso. Esso esegue f + 1 round, dove f è il numero di processi che possono fallire (nel nostro caso f è il numero di guardiani che possono fallire tra i guardiani che monitorano un certo patch). Per ogni round dell'algoritmo, ogni processo mantiene un insieme di valori V (nel nostro caso questi costituiscono gli stati dei guardiani) e invia i valori di questo insieme che non ha ancora inviato a tutti gli altri gurdiani. Essi rimangono in attesa di valori fino ad un timeout (ottenuto grazie a un timer) e, quando questo sopraggiunge, passano al round succesivo dell'algoritmo. Quando tutti i round dell'algoritmo terminano, i guardiani valutano i valori presenti in V e decidono se entrare o meno in allerta. Dato che l'algoritmo fa sì che i guardiani concordino sulla decisione da prendere, il patch viene considerato in allerta se un solo guardiano è entrato in allerta (perché dovranno essere in allerta anche gli altri).
Quando un guardiano entra in allerta, il suo comportamento cambia in *DangerBehavior*. Mentre nel *CheckBehavior* venivano considerati anche messaggi non relativi all'implementazione dell'algoritmo del consenso (questi altri messaggi venivano stashati e poi recuperati all'uscita da questo comportamento), nel *DangerBehavior* vengono ignorati tutti i messaggi ad eccezione del *RecoveryPatchMsg*, che viene ricevuto da tutti i guardiani di un patch quando una dashboard riceve un click sul patch in allerta. Questo messaggio fa in modo che i guardiani si "sblocchino" e ritornino al loro comportamento normale. Tutti i valori relativi ai sensori, memorizzati in precedenza, vengono eliminati non appena si entra nello stato di allerta. Di conseguenza, quando un guardiano esce dallo stato di allerta è completamente resettato.

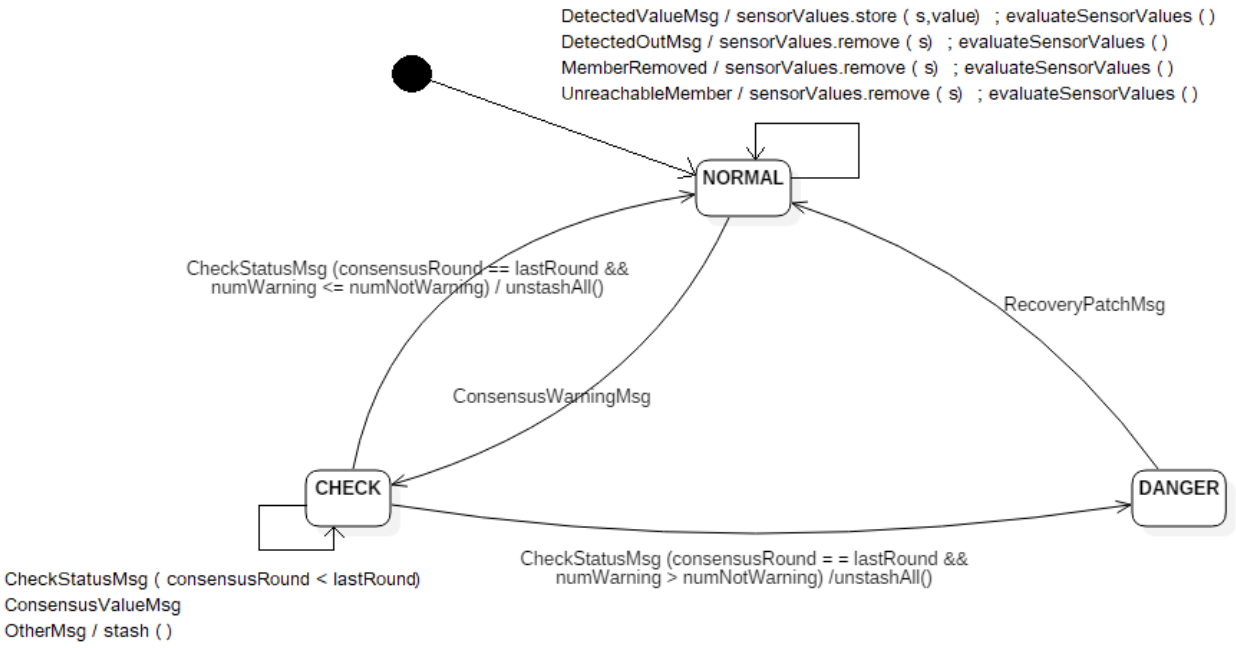

Figure 2.3: Statechart che rappresenta il comportamento dell'*ActorGuardian*

Oltre agli attori già citati è stato realizzato un *ActorDashboard*, che rappresenta l'ultima entità presente nel sistema. In modo analogo a quanto fatto nell'*ActorGuardian*, questo attore memorizza internamente le posizioni dei sensori e gli stati dei guardiani. Il comportamento dell'attore è sempre il medesimo e vengono accettati i seguenti messaggi: *DetectedPositionMsg*, che fa sì che venga aggiornata la posizione relativa ad un sensore, *GuardianStateMsg*, che fa sì che venga aggiornato lo stato relativo ad un guardiano, *DetectedOutMsg*, che elimina la posizione relativa ad un sensore poiché questo è uscito dalla mappa, *MemberRemoved* e *UnreachableMember* , che eliminano i valori relativi ad alcuni guardiani e/o sensori che risultano essere irraggiungibili. I guardiani inviano i propri stati (lo stato di un guardiano può essere: normale, pre-allerta da almeno N secondi e allerta) quando vengono creati, quando un nuovo nodo entra a far parte del cluster (messaggio *MemberUp*) e ogni volta che cambia il loro stato. In questo modo le dashboard sono sempre a conoscenza dello stato dei guardiani.
Ogni volta che l'utente clicca su un patch, l'*ActorDashboard* invia a tutti i guardiani del patch un messaggio di tipo *RecoveryPatchMsg*. In questo modo, se il patch è in allerta, potrà ritornare allo stato normale. Anche in questo caso, i messaggi vengono scambiati all'interno del cluster con il *DistributedPubSubMediator* messo a disposizione dal *Distributed Publish Subscribe*.

2.2.1 Esercizio 2: Guida Utente

Per eseguire la versione di MapMonitor basata sul modello ad attori sono stati creati tre main: uno per l'entità *Guardiano*, uno per l'entità *Sensore* e uno per l'entità *Dashboard*. Ogni main, per poter essere eseguito, richiede che vengano specificati i seguenti argomenti:

```
Quantity Host Port SeedNodes
```

La quantità specifica il numero di guardiani, sensori o dashboard che si vogliono eseguire su un certo nodo, l'host e la porta identificano l'indirizzo ip su cui verranno eseguite le entità (ossia quello della macchina da cui viene lanciato il main) e i seed node identificano i nodi seme del cluster. Questi nodi consentono ad altri nodi di unirsi al cluster ed è perciò importante che, almeno alcuni di essi, rimangano attivi mentre il sistema distribuito è in esecuzione. Per ogni seed node andrà specificato un indirizzo ip nella forma "host:porta".
I tre main presentati non consentono di creare entità di tipo differente (ad esempio due sensori e due guardiani) nello stesso nodo, perché è più intuitivo dividere le entità per tipo, ma potrebbe essere realizzato un altro main che lo fa e il sistema si comporterebbe correttamente.
Le costanti del sistema, come ad esempio M e N, sono contenute nel file *UtilityValues* (che si trova nel package *mapMonitor.common*) in modo da poter essere modificate facilmente.
I guardiani, quando vengono creati, devono essere assegnati ad un determinato patch (quello che dovranno monitorare) e, nel *MainGuardian* realizzato, vengono assegnati ognuno ad un patch differente (a partire dal primo). Se il numero dei guardiani è maggiore del numero di patch, una volta assegnato un guardiano ad ogni patch il "giro" ricomincia.
Ad esempio, per eseguire 14 guardiani su due host diversi potranno essere lanciati due *MainGuardian* (sui giusti host) specificando le seguenti configurazioni:

```
8 192.168.1.4 2550 192.168.1.4:2550 192.168.1.5:2550

6 192.168.1.5 2550 192.168.1.4:2550 192.168.1.5:2550
```

I due nodi contenenti i guardiani sono stati indicati come nodi seme del cluster e, quindi, potranno essere lanciate altre entità che si uniranno al cluster se verranno specificati gli stessi seed node. Ad esempio, con la configurazione riportata sotto, sarà possibile lanciare due sensori o due dashboard all'interno dello stesso MapMonitor (con *MainSensor* e *MainDashboard*).

```
2 192.168.1.8 2551 192.168.1.4:2550 192.168.1.5:2550
```

2.3 Esercizio 3: Versione di MapMonitor basata su distributed object computing

Per quanto riguarda il terzo esercizio la soluzione è costituita dai package *mapmonitor.common*, che contiene alcuni sorgenti comuni all'implementazione del secondo esercizio (*View* relative alle varie entità e classi di utility), e *mapmonitor.rmi*, in cui si trovano i file sviluppati per l'implementazione basata su distributed object computing e realizzata con *Java RMI*.
A differenza di *Akka Clustering*, in cui un qualsiasi tipo di nodo può unirsi dinamicamente al cluster, in *Java RMI* è necessario conoscere quali sono le componenti remote del sistema per potervi comunicare (si tratta di oggetti remoti, su cui possono essere chiamati dei metodi, ma per farlo è necessario recuperare il riferimento al loro *stub*).
La tipica architettura di un'applicazione realizzata con *Java RMI* prevede la suddivisione del sistema in una o più componenti server, realizzate istanziando ed esponendo in rete uno o più oggetti remoti, e in una o più componenti client, che hanno il compito di interagire con gli oggetti remoti. Tuttavia, viste le specifiche che il sistema MapMonitor deve rispettare, non è stato possibile considerare ogni entità semplicemente come client o server.
Per avere conferma di questo è sufficiente pensare ai guardiani: un qualsiasi guardiano potrebbe entrare in pre-allerta e rimanerci per il numero di secondi prestabilito, a quel punto dovrebbe verificare in che stato si trovano gli altri guardiani dello stesso patch e, per farlo, dovrebbe eseguire una chiamata di metodo remota su di essi. Quindi un guardiano dovrà essere un oggetto remoto (che implementa un'interfaccia remota), ossia un server, ma allo stesso tempo dovrà anche essere un client per poter eseguire chiamate di metodi su altri guardiani.
Partendo da questa considerazione e tenendo conto del fatto che, per un corretto funzionamento del sistema, i guardiani devono essere aggiornati sul valore dei sensori presenti nel sistema e le dashboard devono essere aggiornate riguardo la posizione dei sensori e lo stato dei guardiani, si è scelto di realizzare le tre entità come descritto in seguito.
Il sensore è stato realizzato come un semplice oggetto remoto, implementa l'interfaccia *SensorRemote* ed espone quindi due metodi *getPosition()* e *getValue()*, che saranno richiamati rispettivamente dalle dashboard e dai guardiani. Per ogni sensore è stato inoltre creato un thread, che simula l'andamento di posizioni e valori aggiornandole come fatto dall'*ActorSensor* nell'esercizio 2. Questo thread è stato inserito nell'applicazione solo ed esclusivamente per simulare l'andamento del sensore, in un caso reale potremmo pensare ad un sensore come un oggetto remoto che "misura" il suo valore e la sua posizione ogni volta che gli vengono richieste.
La classe *SensorImpl*, oltre all'interfaccia remota, implementa anche l'interfaccia *Sensor*, in cui troviamo i metodi che possono essere richiamati dal thread che gestisce la simulazione. Dato che *SensorImpl* deve gestire gli accessi concorrenti di questo thread e dei processi che chiamano metodi remoti sull'oggetto, questa classe è stata implementata come monitor.
Come anticipato in precedenza, per ogni guardiano dovrà essere creato un oggetto che implementa l'interfaccia remota *GuardianRemote*. Essa mette a disposizione tre metodi: *getState()*, che ritorna lo stato corrente del guardiano e che sarà richiamato dagli altri guardiani dello stesso patch e dalla dashboard, *notifyRecovery()*, che sarà richiamato dalla dashboard per notificare la pressione su un determinato patch, e *notifyWarning()*, che sarà richiamato da un altro guardiano dello stesso patch che si trova in pre-allerta per almeno N secondi.
Oltre all'oggetto remoto, per ogni guardiano è stato creato un thread che si occupa di monitorare i valori dei sensori posizionati all'interno del patch del guardiano e lo stato dei guardiani a cui è stato assegnato lo stesso patch. Il thread creato dai guardiani costituisce la sua componente "client". Per comunicare con i sensori e i guardiani, è necessario che il thread possa accedere allo stub relativo ad essi. Perciò, quando il thread viene creato, si fornisce un insieme di coppie host/nome che consentono di recuperare gli stub relativi ai sensori e ai guardiani. Ogni volta che deve essere eseguita una chiamata ad uno di questi due oggetti remoti, si ottiene il registro relativo all'host e, attraverso il nome dello stub, si risale allo stub su cui è possibile chiamare il metodo. Per semplicità, su ogni host è stato considerato soltanto il registro relativo

alla porta di default (per questo motivo una coppia host/nome identifica un singolo stub, dato che per ogni registro vi è un solo stub associato a un certo nome).
Per quanto riguarda la logica del loop, in linea di massima il *GuardianThread* segue (anche se in modo semplificato) quanto fatto nell'*ActorGuardian*. La principale differenza è costituita dal fatto che, in questo caso, il guardiano non attende un dato dai sensori ma fa polling su essi, eseguendo periodicamente delle chiamate remote. Si è scelto di utilizzare questo approccio perché, se fosse stato il sensore ad inviare i valori misurati al guardiano, quest'ultimo non avrebbe avuto modo di riconoscere, ad esempio, un sensore irraggiungibile.
Come nel caso del sensore, l'oggetto remoto *GuardianImpl* esponde due interfacce: una remota e una accessibile al thread relativo allo stesso guardiano. Onde evitare corse critiche tra questo e i processi remoti, si è scelto di implementare anche *GuardianImpl* come monitor.
Infine le dashboard sono state create come semplici client perché, dato che per conoscere posizione dei sensori e stato dei guardiani è necessario eseguire una chiamata di metodo remoto su di essi, nessuna entità dovrà chiamare metodi su una dashboard. Per ogni dashboard si ha un *DashboardThread*, che conosce le coppie host/nome necessarie per recuperare gli stub di tutti i sensori e dei guardiani relativi a tutti i patch. Questo thread esegue polling, e quindi chiamate periodiche, sui sensori e sui guardiani per essere aggiornato rispettivamente riguardo la loro posizione e il loro stato (e poter mostrare i dati attraverso la *View*).
Il thread realizzato per la dashboard condivide con la *View* relativa alla dashboard un monitor (*DashboardPressed*) che viene controllato ad ogni step nel loop del thread e che consente al thread di capire su quali patch ha premuto l'utente che interagisce con la dashboard. In presenza di una pressione su un patch, il *DashboardThread* si occupa di eseguire una chiamata remota su tutti i guardiani relativi a quel patch (si chiama il metodo *notifyRecovery()*).

2.3.1 Esercizio 3: Guida Utente

Per eseguire la versione di MapMonitor basata su distributed object computing sono stati creati tre main: uno per l'entità *Guardiano*, uno per l'entità *Sensore* e uno per l'entità *Dashboard*. Per lanciarli è però necessario aver configurato correttamente i file *mapmonitor.rmi.guardians.json* e *mapmonitor.rmi.sensors.json* (inseriti nel progetto per facilitare il deploy).
Il file *mapmonitor.rmi.guardians.json* include i dati necessari per creare i guardiani: per ogni patch vengono specificate una lista di coppie host/nome dello stub che specificano l'host e il nome dello stub all'interno del registro di default relativo all'host. Per registro di default si intende quello relativo alla porta di default (ossia 1099), che si è scelto di utilizzare per semplicità. Se per due guardiani è stato specificato lo stesso host, è necessario che il nome dello stub sia univoco (deve identificare l'oggetto remoto all'interno del registro relativo all'host).
Allo stesso modo il file *mapmonitor.rmi.sensors.json* include una lista di coppie host/nome dello stub, che specifica dove saranno in esecuzione tutti i sensori (anche in questo caso, se due sensori fanno riferimento allo stesso host dovranno avere nomi di stub diversi).
Quando vengono eseguiti *MainGuardian* e *MainSensor*, è necessario specificare un host e deve trattarsi dell'host che identifica la macchina da cui viene lanciato il programma. Questi main recupereranno dai file JSON le configurazioni specificate del programma e lanceranno, rispettivamente, tutti i guardiani e tutti i sensori che è previsto siano in esecuzione sull'host dato. Nel caso dei guardiani, il file di configurazione viene utilizzato anche per passare ai vari thread dei guardiani le coppie host/nome dello stub che identificano gli oggetti remoti relativi ai guardiani dello stesso patch e ai sensori. Anche il *MainDashboard*, pur non istanziando nessun oggetto remoto, fa riferimento ai file di configurazione dei guardiani e dei sensori per fornire la possibilità al thread creato per la dashboard di comunicare con essi.

3 PROVE DI PERFORMANCE

La versione di Smartpositioning basata sul modello ad attori (esercizio 1) costituisce una versione concorrente di un problema risolvibile in modo sequenziale. Per questo motivo, nel package *smartpositioning.performance*, sono state inserite una semplice implementazione sequenziale e una semplificazione della versione ad attori realizzata e descritta.
Sono stati misurati e confrontati i tempi d'esecuzione delle due versioni del programma, facendo sempre riferimento ad un numero fisso di step (500). Questo perché nel primo assignment era stato notato come lo speedup aumentasse al crescere del numero di particelle, e non del numero di step (gli step vengono eseguiti in modo sequenziale, dato che uno step può essere eseguito solo dopo il suo precedente). Nelle figure che seguono sono stati riportati i tempi di esecuzione relativi alla versione sequenziale e a quella ad attori, e lo speedup ottenuto

Numero particelle	Tempo esecuzione versione sequenziale (ns)	Tempo di esecuzione con modello ad attori (ns)	Speedup
250	334023015	1386426823	0,240923653
500	1184541442	1706172134	0,694268426
1000	4772316174	2582904509	1,847654901
2000	20993327757	7372555133	2,847496882
4000	77419928243	25033537453	3,092648348
8000	296904025359	102742631880	2,889784113

Figure 3.1: Tempi di esecuzione relativi alla versione sequenziale e a quella ad attori

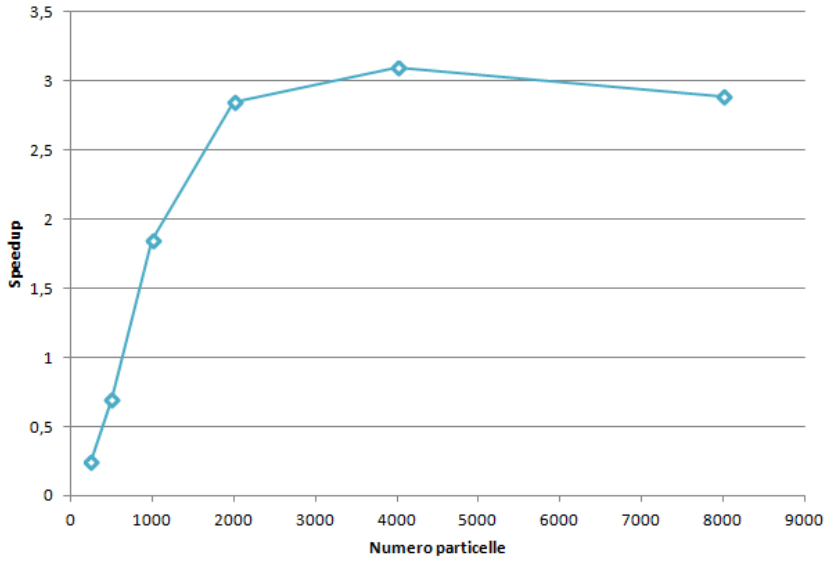

Figure 3.2: Grafico che rappresenta lo speedup ottenuto

4 Risultato ottenuto

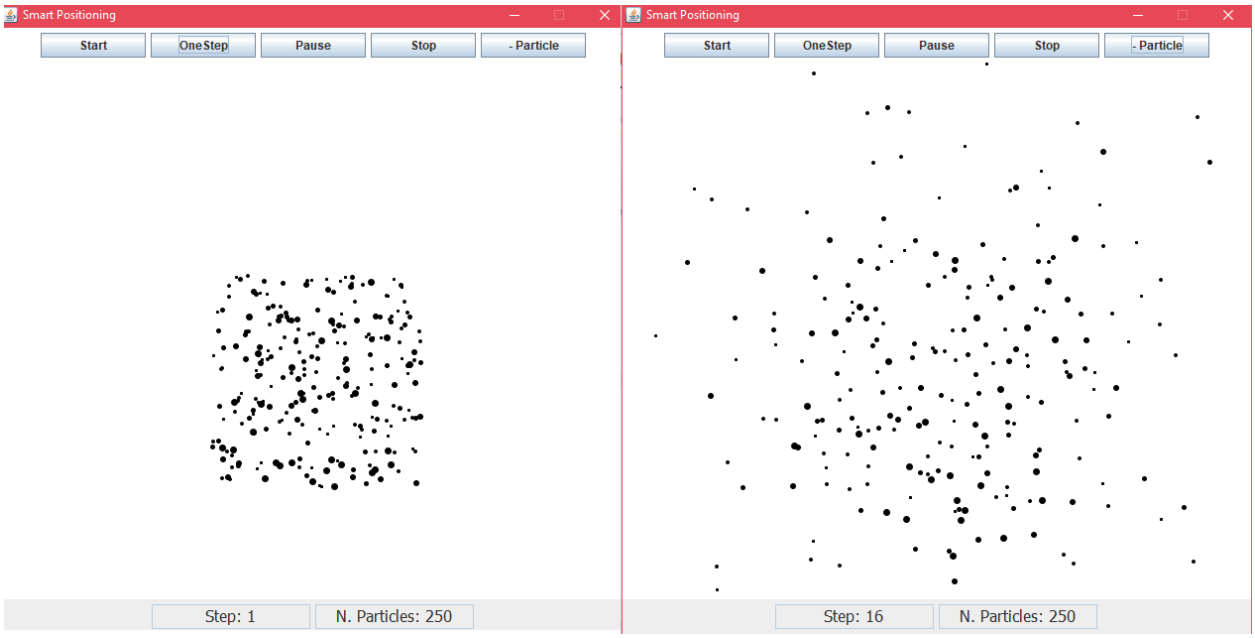

Figure 4.1: *View* relativa a Smartpositioning (esercizio 1)

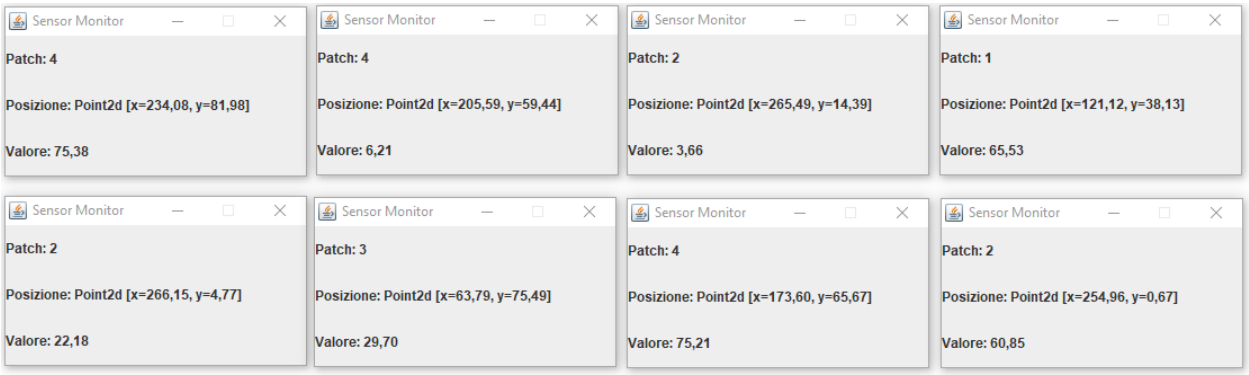

Figure 4.2: *View* che mostra lo stato dei sensori in MapMonitor (esercizio 2/3)

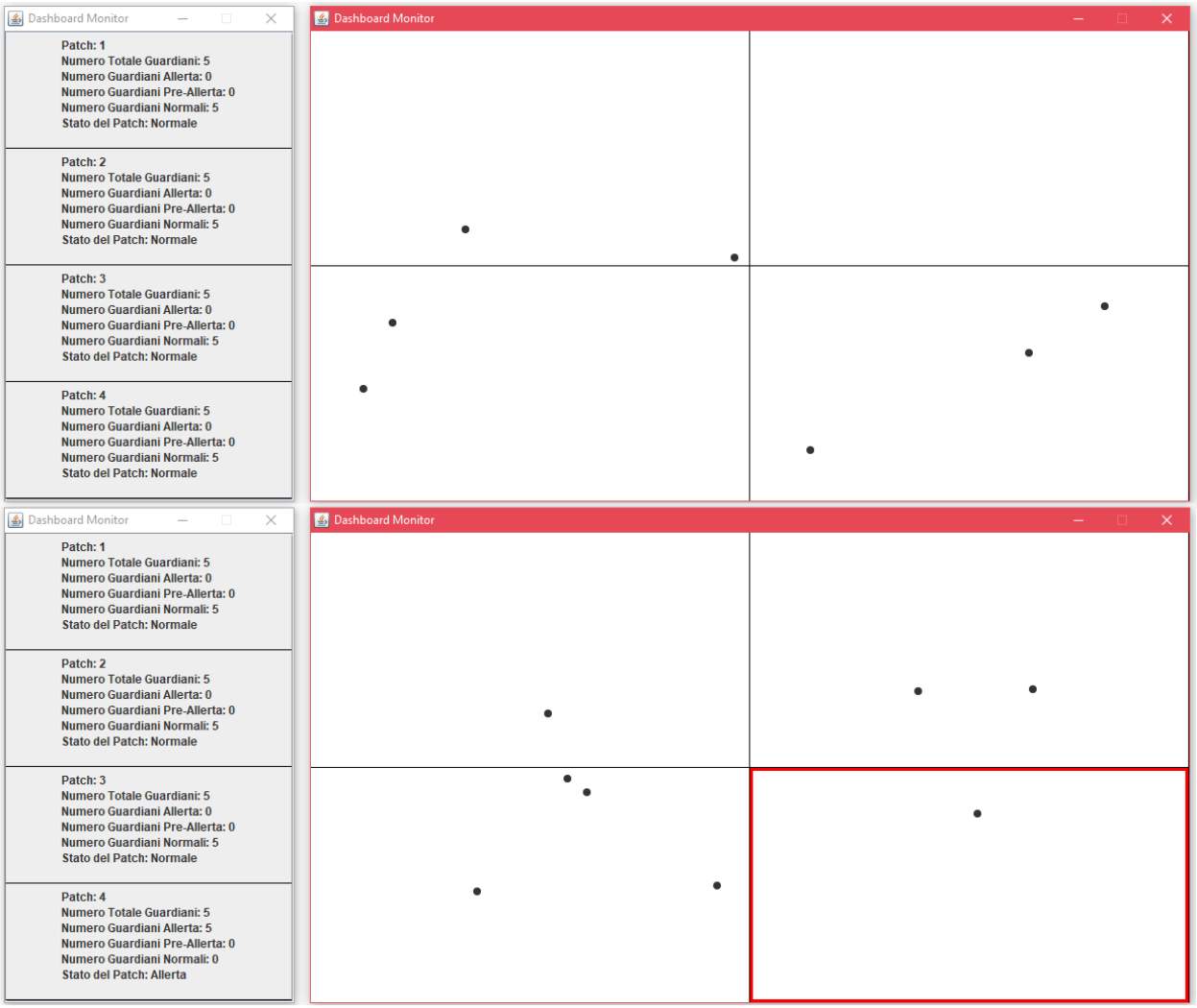

Figure 4.3: *View* che rappresenta la dashboard in MapMonitor (esercizio 2/3)