PCD - Report Assignment 3

Esercizio 1

Andrea Giulianelli
andrea.giulianelli4@studio.unibo.it

Nel primo esercizio dell'Assignment 3 si richiede di realizzare una soluzione al medesimo problema affrontato nell'Assignment 1 basata in questo contesto sullo scambio di messaggi utilizzando come paradigma di riferimento quello ad attori.
Per fare ciò sono stati necessari diversi passi che sono descritti, nei loro punti principali, in questo report.

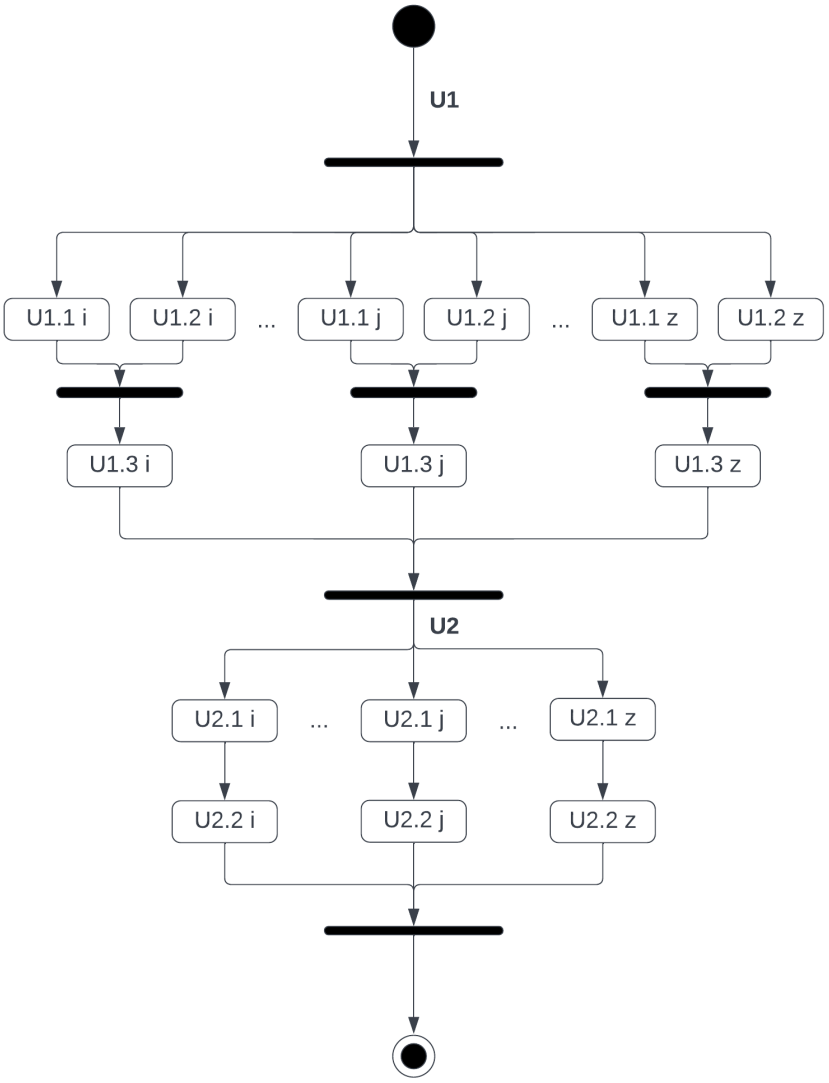
1-Analisi del problema

L'analisi del problema rimane invariata considerando che il problema è il medesimo del primo Assignment e che l'analisi deve astrarre da qualsiasi forma di risoluzione, strategia o tecnologia adottata.
Per comodità riporto la suddivisione in unità e task al quale si era arrivati:

1. **(U) Calcolo forze**: calcolo delle forze a cui è sottoposto ogni corpo *i*.
   - 1.1. **(U) Calcolo forze repulsive *i***: calcola la somma di tutte le forze repulsive a cui il corpo "*i*" è sottoposto.
     - 1.1.1. **(T) Calcolo forza repulsiva *ji***: calcolo della singola forza repulsiva che il corpo *"j"* esercita sul corpo *"i"*.
   - 1.2. **(U-T) Calcolo forza d'attrito *i***
   - 1.3. **(U-T) Calcolo accelerazione *i***: calcolo dell'accelerazione del corpo "*i*" risultante da tutte le forze a cui è sottoposto.
2. **(U) Calcolo nuove posizioni**: calcolo della nuova posizione di ogni corpo *i.*
   - 2.1. **(U) Calcolo posizione *i***
     - 2.1.1. **(T) Calcolo velocità *i***
     - 2.1.2. **(T) Calcolo posizione *i***: calcolo effettivo della posizione.
   - 2.2. **(U) Calcolo posizione bounded *i***
     - 2.2.1. **(T) Check confini del mondo**: le posizioni calcolate devono essere corrette in base ai confini del mondo simulato.

(U) - unità
(T) - task

Questi appena descritti sono le unità e i task che vengono eseguiti ad ogni iterazione della simulazione.
Ovviamente, anche per quanto riguarda le dipendenze valgono le medesime considerazioni fatte per l'Assignment 1 e 2.1.
Si riporta quindi lo stesso diagramma delle dipendenze espresso tramite una notazione ispirata al diagramma di attività in UML 2:



2-Design

Considerando la necessità di adottare un approccio basato sullo scambio di messaggio utilizzando come paradigma di riferimento quello ad attori, in questa sezione si fornirà una descrizione della soluzione adottata in termini di descrizione della strategia risolutiva e di architettura adottata.

Durante la progettazione si è cercato di adottare un approccio il più possibile idiomatico seguendo il principio “everything is an actor” astraendo completamente dalla gestione dei thread ragionando sempre a livello logico e di dominio. Inoltre, si è evitata qualsiasi forma di memoria condivisa, monitor, locks o semafori; è stato adottato unicamente lo scambio di messaggi (immutabili) tra attori. In questo modo, è stato possibile incapsulare il flusso di controllo e decentralizzare le responsabilità tra le parti che interagiscono. Considerando ciò, tutti i meccanismi di coordinazione sono stati creati mediante protocolli basati sullo scambio di messaggio.

L’approccio descritto in questo Assignment non cerca, come primo obiettivo, di raggiungere prestazioni superiori a quelle ottenute nel secondo Assignment, ma cerca di modellare il problema con un approccio ad attori il più possibile puro facendo poi considerazioni relativamente alle performance ottenute e possibili miglioramenti.

Descrizione della strategia scelta

La strategia scelta, basata sul paradigma ad attori, sfrutta tre diversi tipi di attori:

- *Coordinator*: è l'attore che gestisce ad alto livello il *simulatore*. Si occupa della gestione dell'avvio e della pausa del simulatore oltre che della gestione dei risultati ottenuti ad ogni iterazione interagendo con gli attori esterni ad esso, come ad esempio l'attore che gestisce la View (descritto in seguito).
- *Simulation*: è l'attore che si occupa della gestione della *simulazione*. Quindi si occupa della creazione dei vari corpi (e quindi dello *spawn* dei vari *ActorBody*), dell'aggregazione dei vari risultati e dell'avanzare delle varie iterazioni fino alla terminazione, interagendo con il *Coordinator*.
- *ActorBody*: ogni body della simulazione è stato modellato come un attore. Essi interagiranno tra loro al fine di evolversi. In questo modo, a mio avviso, rispetta maggiormente il principio "*everything is an actor*" evitando meccanismi di memoria condivisa.

In questa lista si astrae dalla gestione della view che verrà descritta nella sotto-sezione "Architettura".

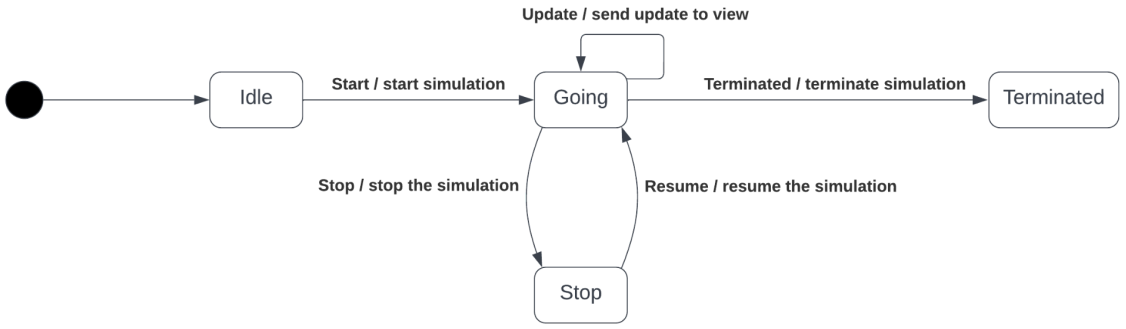
La strategia, ad alto livello, può essere descritta nel seguente modo:

1. Il *Coordinator*, informato della necessità di iniziare la simulazione dalla View (descritta in seguito), crea ed avvia l'attore Simulation.
2. L'attore *Simulation* successivamente crea i vari *ActorBody* e spedisce ad ognuno di essi un messaggio di *Start* al fine di avviarli.
3. Ogni *ActorBody* spedisce un messaggio contenente una tupla formata da <posizione e massa> a tutti gli altri *ActorBody*. Posizione e massa sono gli elementi necessari al calcolo delle forze, come si può notare nella descrizione del problema e nella sua analisi.
4. Ogni *ActorBody*, per ogni tupla (posizione, massa) ricevuta aggiorna la somma delle forze repulsive.
5. Ogni *ActorBody*, una volta calcolate tutte le forze repulsive, calcola la forza d'attrito e aggiorna la propria accelerazione.
6. Dopodiché ogni *ActorBody* invia a tutti gli altri *ActorBody* un messaggio per informare del proprio completamento del calcolo delle forze, al fine di rispettare i vincoli descritti in fase di analisi.
7. Ricevuti tutti i messaggi, gli *ActorBody* calcolano i dati rimanenti, cioè velocità e posizione (con i rispettivi controlli rispetto ai boundary).
8. Ogni *ActorBody* dopo aver calcolato la posizione invia all'attore *Simulation* la posizione aggiornata.
9. Dopodiché ogni *ActorBody* invia a tutti gli altri *ActorBody* un messaggio per informare del proprio completamento del calcolo delle posizioni al fine di sincronizzarsi, rispettando i vincoli descritti in fase di analisi. Inoltre, in questo caso, gli *ActorBody* rimangono in attesa dello stesso messaggio di conferma anche da parte dell'attore *Simulation* al fine di confermare la gestione safe delle nuove posizioni.
10. L'attore *Simulation*, una volta aggregati tutti i risultati dell'iterazione corrente, li spedisce all'attore *Coordinator* il quale gestisce poi l'interazione con gli attori esterni al dominio del problema (come ad esempio la View, separando in questo modo le responsabilità).
11. Si riparte dal punto 3.

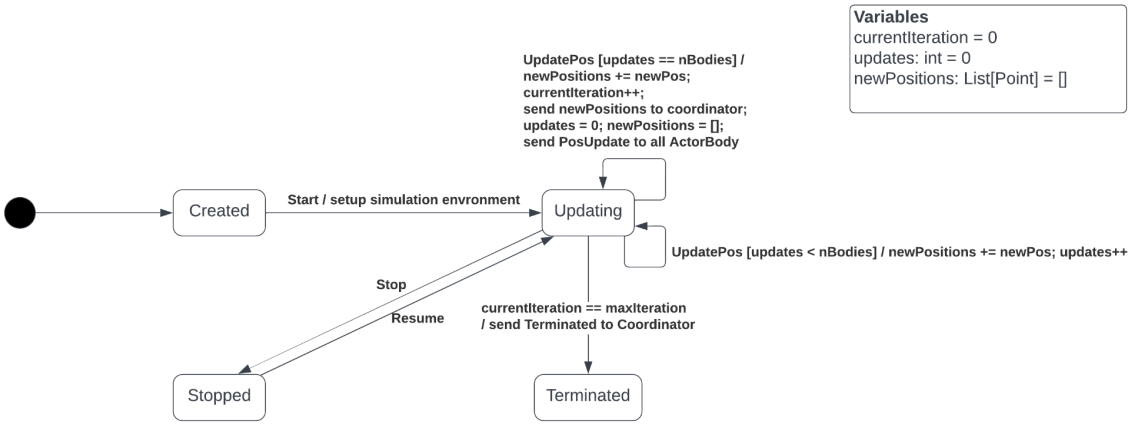
Nei punti 6 e 9 si nota che gli attori adottano un protocollo di coordinazione che simula il comportamento di una barriera in cui ogni *ActorBody* invia un messaggio a tutti gli altri *ActorBody* di cui ha il riferimento. Ho preferito questo protocollo ad un attore che modellava esplicitamente la barriera in quanto introduce meno colli di bottiglia a discapito di una quantità maggiore di messaggi scambiati.
In aggiunta a quanto descritto precedentemente, anche la Pausa e la Ripresa della simulazione sono state gestite a livello di protocollo di scambio di messaggio. Infatti, la View segnala all'attore *Coordinator* l'intenzione di mettere in pausa/riprendere la simulazione. Dopodiché, il *Coordinator* lo segnala all'attore *Simulation* il quale, durante la pausa, non parteciperà al protocollo di fine iterazione (barriera del punto 9) mettendo in pausa l'avanzare della simulazione.

Il comportamento delle tre tipologie di attori è stato modellato attraverso l'utilizzo di FSM (Finite State Machine) permettendo di progettare gli attori stessi con una maggiore estendibilità e con un maggior controllo del comportamento. In generale, il design proposto, astrae, come anticipato, da eventuali tecnologie o librerie utilizzate nelle fasi implementative, cercando di rimanere legati al paradigma ad attori.

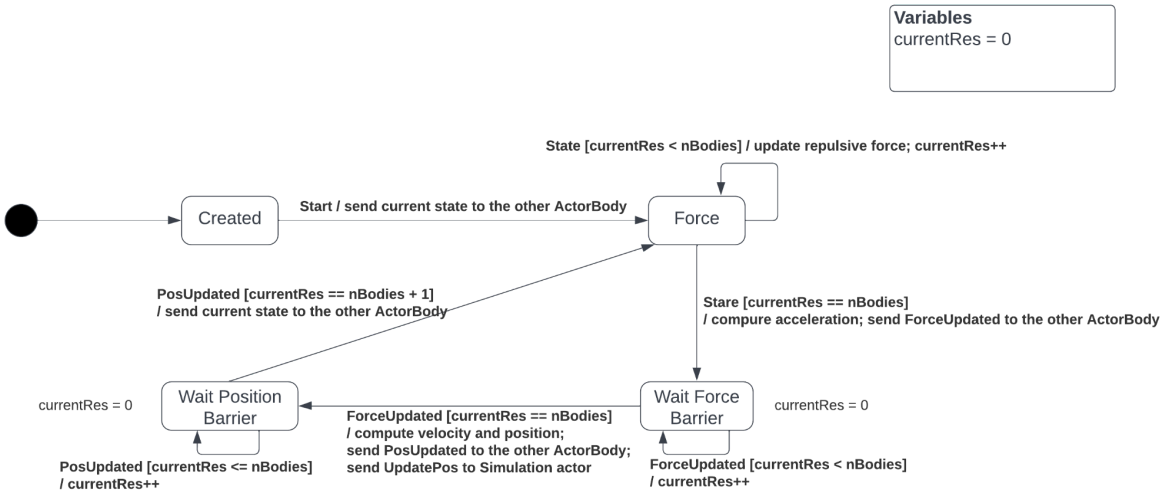
Coordinator



Simulation



ActorBody



I tre attori sono stati descritti attraverso l'utilizzo di EFSM (quindi sfruttando l'ausilio di variabili) in cui gli stati rappresentano gli stati in cui si trovano i rispettivi attori e le transazioni sono descritte da guardie rappresentate dal messaggio ricevuto e da azioni che indicano l'azione eseguita prima di entrare nello stato successivo.
In queste modellazioni si astrae completamente dai valori iniziali (numero di iterazioni, creazione body ecc..), ma ci si concentra solamente sui cambi di stato (*become*) eseguite dagli attori.
Gli attori sono stati modellati come componenti attivi, ma soprattutto reattivi (non hanno un comportamento proattivo che automaticamente viene eseguito), nel rispetto del paradigma.
Inoltre, è possibile notare, considerando le caratteristiche del paradigma ad attori, che il caso in cui i messaggi arrivino fuori ordine o in uno stato non adatto a gestirli è possibile, ma non è stato rappresentato all'interno della modellazione tramite FSM per non appesantire la notazione. Essi saranno gestiti a livello implementativo sfruttando le potenzialità messe a disposizione dalla libreria utilizzata.

Nella modellazione delle FSM si è astratto inoltre dalla struttura dettagliata dei messaggi. La struttura è descritta di seguito considerando le varie interazioni tra attori.

Coordinator → Simulation

```
- Start(iterations: int, numberOfBodies: int, boundary: (double,
  double, double, double))
- Stop
- Resume
```

Simulation → Coordinator

```
- Update(iteration: int, virtualTime: double, positions:
  List[Point])
- Terminated
```

Simulation → ActorBody

```
- Start(actorBodyRefs: List[Ref])
- PosUpdated
```

ActorBody → Simulation

```
- UpdatePos(newPosition: Point)
```

ActorBody → ActorBody

```
- State(position: Point, mass: double)
- ForceUpdated
- PosUpdated
```

Per gestire la ricezione di messaggi di risposta è stato utilizzato l'interaction pattern "Adapter".
Quindi tutti i meccanismi di sincronizzazione/coordinazione sono stati ottenuti mediante i suddetti protocolli di scambio di messaggio utilizzati all'interno del comportamento modellato tramite FSM. Questo ha permesso di rispettare i vincoli del problema.

Architettura adottata
Al fine di sfruttare il paradigma ad attori anche nell'architettura del software, è stata utilizzata una versione di MVC adattata al paradigma stesso la quale prevede:

- Model: è il model tradizionale, con le classi che modellano i concetti del dominio.
- Controller formato dai tre attori: *Coordinator, Simulation* e *ActorBody* descritti precedentemente. Essi infatti rappresentano tutte le parti di logica e controllo dell'applicazione. Come anticipato, la decisione di separare il controllo in tre diversi attori è al fine di separare le responsabilità. In particolare, solamente l'attore *Coordinator* è consapevole dell'esistenza della View. In questo modo la simulazione e il suo controllo è confinata ed indipendente dal resto.
- View: al fine di gestire la view è stato creato un ulteriore attore *ViewActor* che consente alla parte ad attori di comunicare con la parte non ad attori, quindi gestisce il flusso *Coordinator -> View*.
  Il protocollo e la struttura dei messaggi è la seguente:

  View → Coordinator

  ```
  - Start
  - Stop
  - Resume
  ```

  Coordinator → ViewActor

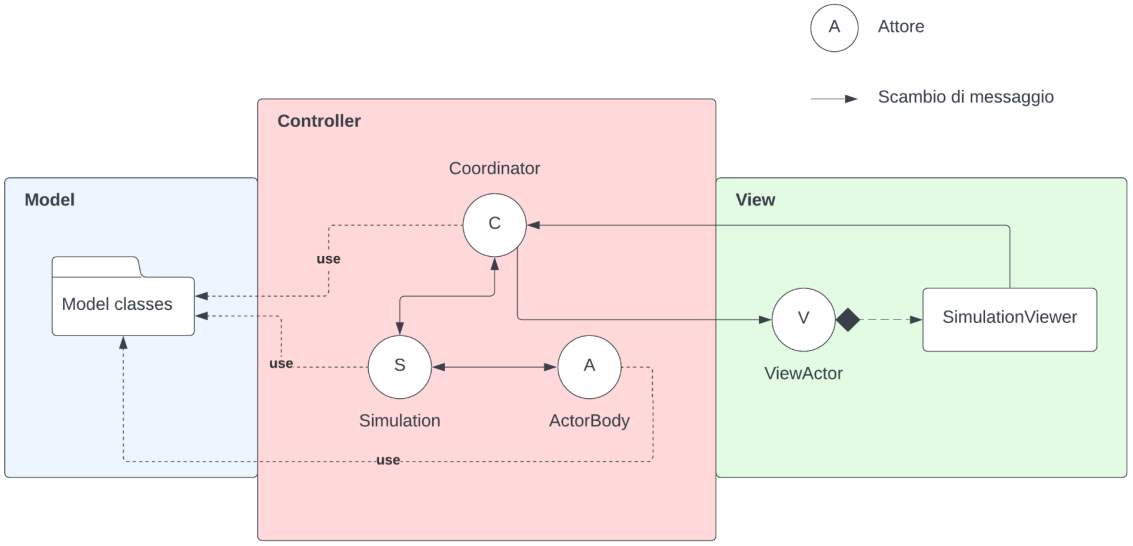
  ```
  - Update(positions: List[Point], virtualTime: double,
    iteration: int, boundary: (double, double, double, double))
  - Terminated
  ```

L'attore *ViewActor*, essendo anch'esso un componente reattivo, è avviato attraverso il seguente messaggio:

```
- Start(width: int, height: int, coordinator: Ref)
```

Come si può notare, nel messaggio di start viene passato l'id dell'attore *Coordinator*. Questo viene utilizzato al fine di permettere alla parte non ad attori (la View) di comunicare con il gestore del simulatore, il *Coordinator*, al fine di gestire l'avvio, la pausa e la ripresa della simulazione.

Di seguito, lo schema architetturale che riassume le interazioni principali:



L'attore *ViewActor* una volta ricevuti i dati dell'iterazione corrente dal *Coordinator* si occupa di gestire l'update della GUI sfruttando il *SimulationViewer* interno.
Considerando le caratteristiche del paradigma ad attori, è possibile, in questa modellazione, che l'attore *ViewActor* riceva i messaggi di update dal *Coordinator* fuori ordine (quindi che un iterazione x arrivi prima di una y, con x > y). Questo è stato considerato nel design e la sua gestione verrà descritta a livello implementativo.

3-Implementazione

Il Simulatore è stato implementato sfruttando il linguaggio di programmazione *Scala* e il toolkit *Akka* per utilizzare il paradigma ad attori. Considerando le caratteristiche del linguaggio utilizzato, si è preferito adottare l'approccio funzionale. Al tempo stesso, come suggerito dalla documentazione di Akka, in alcuni casi è conveniente mettere assieme allo stile funzionale qualche elemento dello stile ad oggetti.
Seguendo lo stile indicato da Akka, ogni attore è stato modellato interamente all'interno di un *object* di *Scala* in cui sono descritti i messaggi che esso può gestire e la factory per crearlo. L'implementazione di esso così rimane incapsulata al suo interno permettendo una maggior agilità di modifica.

Per i messaggi del protocollo utilizzati solo internamente all'attore, la visibilità è stata ridotta all'attore stesso (come nel caso di *ActorBody* in cui i messaggi *State* e *ForceUpdated* sono *privati*). Inoltre, al fine di gestire i messaggi di risposta e quindi effettuare le traduzioni dei vari protocolli utilizzati dagli attori è stato sfruttato l'interaction pattern *Adapter* reso particolarmente agile dal toolkit *Akka*. In questo modo si riducono i rischi legati all'utilizzo errato del protocollo progettato.

Il comportamento degli attori è stato modellato attraverso l'utilizzo di FSM. Questa modellazione trova spazio anche all'interno dell'implementazione in quanto l'API di *Akka Typed* permette di modellare naturalmente le FSM. Infatti:

- I vari stati della FSM diventano diversi *Behavior*.
- Ogni *Behavior* (il quale rappresenta uno stato) viene modellato attraverso un metodo separato, memorizzando così lo stato all'interno dei parametri del metodo stesso.
- Il Behavior ritornato rappresenta il prossimo stato. Perciò le transazioni della FSM sono abilitate dal valore di ritorno dei suddetti metodi e dalle reazioni dei *Behavior* che gestiscono i messaggi ricevuti.

Come anticipato nella parte di design, considerando le caratteristiche del paradigma ad attori e la modellazione adottata, si può verificare il caso in cui i messaggi arrivino fuori ordine o in uno stato non adatto a gestirli. A tal fine i due problemi sono stati affrontati nel seguente modo:

- Per quanto riguarda la ricezione di messaggi in uno stato non adatto a gestirli è stato utilizzato il meccanismo dello *Stash* offerto da *Akka*. Esso permette ad un attore di depositare temporaneamente all'interno di un buffer i messaggi che non possono essere gestiti nel *Behavior* corrente.
  Ad esempio questo problema si verifica nell'attore *ActorBody* nello stato *Created* in cui mentre l'attore aspetta l'arrivo del messaggio di *Start* nel frattempo potrebbero arrivare i messaggi *State* provenienti dagli altri *ActorBody*.
- Invece per quanto riguarda l'ordine dei messaggi nel caso di interazione tra una singola coppia di attori sono state sfruttate le caratteristiche di Akka che assicurano il rispetto dell'ordine di ricezione dei messaggi rispetto all'invio. Questo solamente nel caso di invio di più messaggi sequenziali da parte dello stesso attore. Quindi nel nostro caso consente di risolvere, senza ulteriori interventi, l'ipotetico problema di ricevere i dati di iterazioni nuove prima di quelle vecchie da parte del *ViewActor* (nell'interazione con il *Coordinator*).

Infine, una possibile alternativa allo specificare gli indirizzi degli altri corpi nel messaggio di *Start* di un *ActorBody* è sfruttare l'API *Receptionist* offerta dal toolkit *Akka*.

4-Prove di performance

Sono state eseguite delle prove di performance per confrontare le prestazioni ottenute rispetto alla versione precedente dell'Assigment 2.
Le caratteristiche del sistema su cui sono stati eseguiti i test sono:

- 16 GB di RAM
- CPU Quad-Core, NO Hyper Threading (quindi solo 4 core), da 3.90 GHz.

I test sono stati eseguiti considerando simulazioni con N corpi per un numero di iterazioni Nsteps - con N: 100, 1000 e Nsteps: 1000, 10000, 20000.
I tempi sono indicati in millisecondi.

Sequenziale

	steps		
body	**1000**	**10000**	**20000**
100	98	675	1229
1000	4922	49586	97820

Soluzione Assignment 2

	steps		
body	**1000**	**10000**	**20000**
100	128	747	1482
1000	1834	14556	28631

Soluzione attuale ad attori

	steps		
body	**1000**	**10000**	**20000**
100	2090	18370	37208
1000	214805	2166901	4321737

Si nota un peggioramento sostanziale delle performance.
È stato eseguito un profiling con VisualVM in cui si nota un elevato utilizzo della CPU e della memoria. Nonostante ciò la soluzione non permette di ottenere performance nemmeno vicine alla versione sequenziale. I motivi di ciò sono da ricercare in diversi aspetti:

- Elevata quantità di messaggi scambiati.
  Il protocollo prevede lo scambio di una quantità molto alta di messaggi. Impostando un numero abbastanza elevato di corpi e di iterazioni è possibile arrivare anche a milioni di messaggi scambiati.
- Utilizzo di un linguaggio che sfrutta la JVM. La JVM, soprattutto per la parte concorrente, non è stata progettata per il modello ad attori.

Ho preferito ugualmente proseguire e presentare questa soluzione in quanto essa rappresenta un tentativo di modellare un problema e progettare una soluzione rimanendo all'interno del paradigma ad attori con il tentativo ulteriore di seguire il principio "everything is an actor" e rappresentare ogni meccanismo di coordinazione/sincronizzazione attraverso un protocollo basato sullo scambio di messaggio.

Soluzione alternativa
Una possibile soluzione che avrebbe dato prestazioni migliori rispetto alla soluzione attuale è la modellazione ad attori dell'architettura *Master-Worker*. Infatti, modellando il Master ed

ogni Worker come un attore le interazioni necessarie (da progettare mediante protocolli basati sullo scambio di messaggio) vengono notevolmente ridotte, permettendo un aumento delle performance. Inoltre, le computazioni eseguite nella soluzione corrente da ogni singolo attore sono molto semplici, mentre nella versione Master-Worker ogni Worker possiede una quantità più importante di lavoro che risente meno dell'overhead introdotto dal design della soluzione.
Tuttavia, come anticipato, ho preferito presentare questa soluzione in quanto, a mio parere, è più vicina al modello/paradigma ad attori.