à49Cittàfigure.4


Università degli Studi di Padova
Corso di Laurea Magistrale in Informatica

Sistemi Concorrenti e Distribuiti

# Traffico città

- Versione 1
- Versione 2

*Modifiche:*

- -

*Aggiunte:*

- Stesura documento

# Indice



# Elenco delle tabelle

# Elenco delle figure

# 1 Introduzione

Questo documento ha lo scopo di presentare il lavoro svolto per il corso di Sistemi Concorrenti e Distribuiti per il corso di Laurea Magistrale in Informatica.

Verrà presentata inizialmente una breve descrizione del problema, seguita poi dall'analisi di tale problema e dalla presentazione di una delle possibili implementazioni.

## 1.1 Problema

Il problema consiste nello studio e nell'implementazione di sistema rappresentante una città con i suoi residenti. La città deve gestire in maniera corretta tutti gli aspetti di concorrenza e di distribuzione. Requisiti obbligatori sono dunque la distribuzione e la scalabilità.

# 2 Esecuzione

Prima di illustrare però lo studio fatto, verrà illustrato brevemente il software necessario per la corretta esecuzione dell'applicazione.

## 2.1 SBT

SBT è un build tool open source usato per progetti Scala e Java. Al momento, nonostate varie critiche, è il build tool più utilizzato per i progetti realizzati in Scala. Per poter eseguire il progetto, sono stati offerti due diversi metodi.

Il primo consente di eseguire tutto senza fornire alcun parametro. Basterà semplicemente avviare l'esecuzione tramite il comando *sbt run* come nella figura sotto riportata. In questo modo sarà l'applicazione verrà eseguita con una sola JVM.

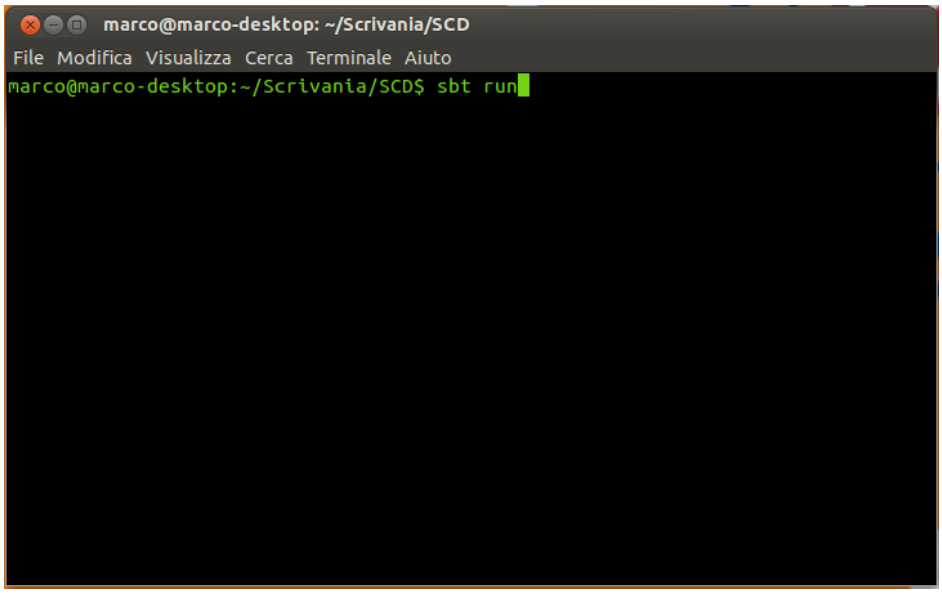


Figura 1: Unica JVM

Il secondo consente invece di specificare l'identificativo e la porta desiderata per ogni zona della città, la quale verrà quindi eseguita in una propria JVM differente. Questa modalità è molto utile in quanto consentirà di 'rimuovere' e 'aggiungere' zone a piacimento, ma questo verrà spiegato meglio in seguito.

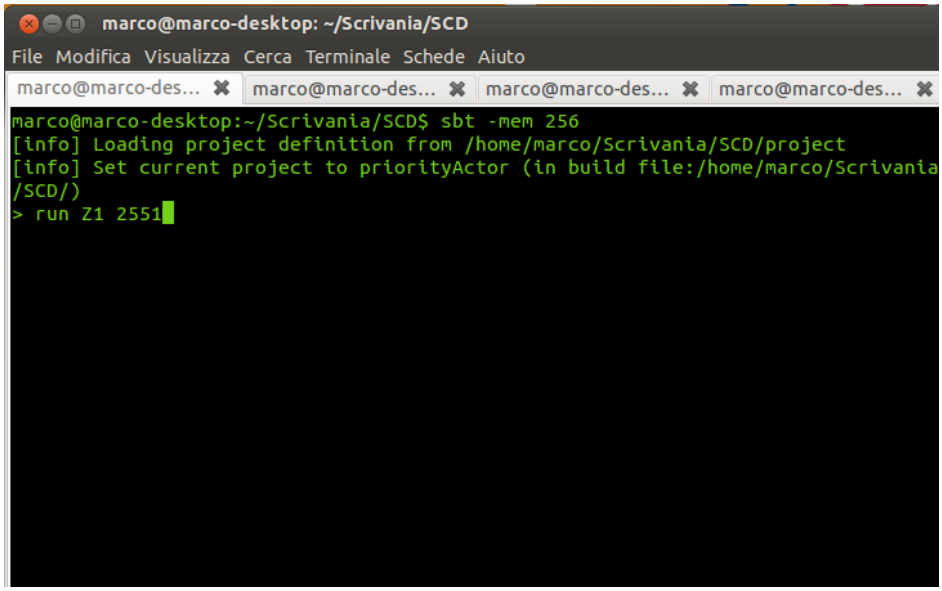


Figura 2: Multiple JVM

## 2.2 Scala ed Akka

Il linguaggio scelto per realizzare il progetto è Scala. Per l'installazione del toolkit Akka basterà semplicemente specificare nel file di configurazione (*build.sbt*) del progetto le 'dipendenze' . In questo modo, alla prima esecuzione, SBT scaricherà tutto il necessario per la corretta esecuzione.

```
// Add multiple dependencies
libraryDependencies ++= Seq(
        "com.typesafe.akka"             %%      "akka-actor"            %       "2.4-SNAPSHOT",
        "org.scala-lang.modules"        %%      "scala-xml"             %       "1.0.2",
        "com.typesafe.akka"             %%      "akka-cluster"          %       "2.3.5",
        "com.assembla.scala-incubator"  %%      "graph-core"            %       "1.9.4"
)
```

Figura 3: Build.sbt

# 3 Studio del problema

## 3.1 Entità

Il primo obbiettivo del progetto era decidere cosa bisognava distribuire e cosa no. Per poter raggiungerlo correttamente, è stata creata una gerarchia delle entità che formano una città.

- Zone
- Incroci
- Strade/Marciapiedi
- Fermate
- Mezzi
- Persone

Si è dunque studiato quale delle entità sopra elencate potesse essere distribuita.

- Zone: possono gestire molte cose internamente e comunicare solo per le cose essenziali
- Incroci, fermate, strade, marciapiedi, mezzi e persone: richiedono molta comunicazione rispetto alle zone in quanto devono eseguire un insieme limitato di compiti

Distribuire le zone è sembrata dunque la scelta migliore.

Entrando nel dettaglio, sono poi state definite le varie entità. Dato che il lavoro è stato svolto da una singola persona, sono state fatte delle scelte atte a non aumentare troppo la difficoltà del progetto. In primo luogo, la città è stata pensata come una griglia. Ogni cella di tale griglia contiene le varie strade, i vari marciapiedi ed eventualmente un incrocio. Tale città non verrà generata in maniera casuale, ma verrà creata all'inizio dall'utente.

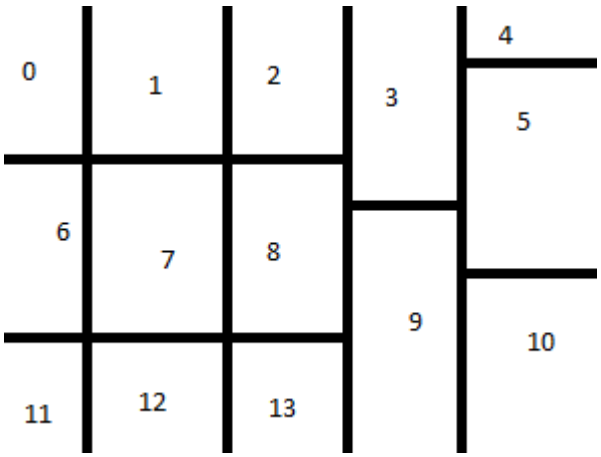


Figura 4: Città

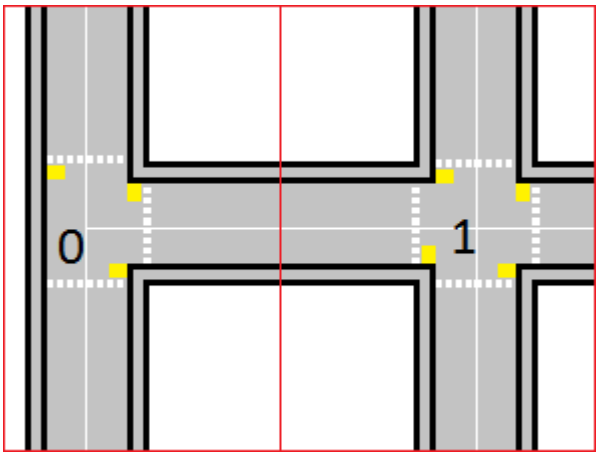


Figura 5: Zone

Come si vede dall'immagine qui sopra, si è deciso di permettere ai pedoni di attraversare la strada solamente agli incroci. Per fare ciò ovviamente sarà necessaria una corretta gestione dei semafori, i quali funzioneranno nel seguente modo:

- 1: Semafori rossi per i pedoni; Semafori verdi per alcune strade.
- 2: Semafori verdi per i pedoni; Semafori rossi per tutte le strade.
- 3: Semafori rossi per i pedoni; Semafori verdi per le altre strade strade.

- 4: Semafori verdi per i pedoni; Semafori rossi per tutte le strade.
- 1: Semafori rossi per i pedoni; Semafori verdi per alcune strade.
- 2: E così via...

Un'altra limitazione aggiunta consiste nel mettere dei versi alle strade e ai marciapiedi. Le macchine ed i pedoni potranno procedere solamente nelle direzioni indicate nella figura sottostante.

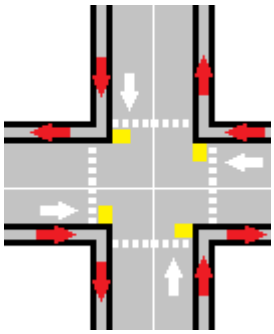

Figura 6: Direzioni

Si è deciso di creare all'inizio anche i percorsi che dovranno seguire mezzi e pedoni. Le posizioni dei mezzi e dei pedoni verranno mostrate all'arrivo negli incroci, all'arrivo in una zona e all'inizio di una strada/marciapiede.

## 3.2 Stati

Sono stati ora analizzati gli stati assunti dalle varie entità.

- Zona:
  - 1. Creazione
  - 2. Ricezione mezzi
  - 3. Ricezione pedoni
  - 4. Attesa
  - 5. Instradamento mezzi
  - 6. Instradamento pedoni
- Incrocio:
  - 7. Gestione incrocio
  - 8. Ricezione mezzi
  - 9. Ricezione pedoni

  - 10. Invio mezzi
  - 11. Invio pedoni
- Corsia:
  - 12. Invio mezzo
  - 13. Ricezione mezzo
  - 14. Attesa
- Marciapiede:
  - 15. Invio pedone
  - 16. Ricezione pedone
  - 17. Attesa
- Fermata:
  - 18. Arrivo persona
  - 19. Arrivo autobus
  - 20. Partenza
  - 21. Attesa
- Autobus:
  - 22. In movimento
  - 23. In sosta
- Automobile:
  - 24. In movimento
  - 25. In sosta
- Pedone:
  - 26. In movimento
  - 27. In sosta
- Tratto:
  - 28. Invio mezzi
  - 29. Ricezione mezzi
  - 30. Attesa
- Striscia pedonale:
  - 31. Invio pedone
  - 32. Ricezione pedone
  - 33. Attesa

## 3.3 Azioni

- Tabellona

## 3.4 Sincronizzazione

- Non so tuttora se sia giusto

# 4 Implementazione

## 4.1 Progettazione

### 4.1.1 Linguaggio

Il linguaggio scelto per realizzare il progetto è Scala. Tramite l'uso del toolkit Akka, è infatti possibile applicazioni concorrenti, distribuite e resistenti agli errori. Il modello ad attori permette di gestire in maniera più semplice ed efficiente i problemi di concorrenza.

È permessa inoltre una buona gestione degli errori. Ogni attore è supervisionato da un altro attore. Ogni attore che provocherà un errore dovrà infatti riferire il problema al suo supervisore, il quale dovrà reagire in maniera appropriata.

Akka permette anche di realizzare applicazioni distribuite. È possibile infatti creare attori in diversi nodi all'interno di un cluster. Tali attori potranno comunicare senza alcun problema nonostante vengano eseguiti in nodi diversi.

### 4.1.2 Concorrenza

Ogni componente del progetto è rappresentata da un attore. Queste entità comunicano tra di loro in modo diretto tramite scambio di messaggi. In questo modo ho più comunicazione, ma è meno probabile che si verifichino situazioni pericolose o deadlock. Di conseguenza non saranno presenti variabili condivise tra gli attori: ogni attore avrà delle proprie variabili modificabili solamente da se stesso in seguito alla ricezioni di certi messaggi. Non vi è nemmeno bisogno di utilizzare synchronized, wait() o notify() per la gestione di eventuali liste. Ogni messaggio infatti verrà gestito singolarmente, mentre tutti gli altri rimarranno in attesa di essere letti.

Come detto in precedenza, non possono verificarsi dei deadlock non verificandosi tutte e 4 le condizioni relative al deadlock. Non ci sono infatti situazioni di accumulo di risorse necessarie ad altri processi (ogni attore riceve mezzi o persone, ma non deve bloccarsi per aspettarle). Non sono presenti quindi situazioni di attesa circolare che potrebbero portare ad uno stallo.

Non si può verificare nemmeno una situazione di starvation; i vari attori infatti non avranno priorità diverse. L'unica situazione in cui vengono gestite priorità diverse è quella della ricezione di messaggi relativi ai semafori. All'arrivo di un messaggio 'semaforo rosso' o 'semaforo verde' , questi saranno letti subito dopo il messaggio attuale. Akka però garantisce l'ordinamento dei messaggi con la stessa priorità provenienti dallo stesso mittente, quindi tutti i messaggi relativi a mezzi o pedoni saranno letti prima o poi. Tale gestione delle priorità avverrà nel seguente modo: verranno usati due stati (due metodi 'receive' ) per poter gestire correttamente gli invii, uno da usare quando il semaforo è rosso e l'altro da usare quando il semaforo è verde. Il primo riceverà

i mezzi e li salva nella coda. Appena verrà ricevuto un messaggio (semaforo verde), lo stato verrà cambiato. Il secondo dunque invierà i mezzi in coda ed eventuali nuovi mezzi ricevuti finché non riceverà un certo messaggio (semaforo rosso). In questo caso si ritornerà al primo metodo 'receive' . Per gestire correttamente i semafori e quindi fermare subito il trasferimento dei mezzi, l'attore necessita di una mailbox particolare. Sarà infatti necessario l'utilizzo di una 'BoundedStablePriorityMailbox' in quanto essa gestisce messaggi con priorità, garantendo al tempo stesso l'ordinamento corretto di messaggi con la stessa priorità.

Di seguito si potrà vedere uno schema che dimostra la correttezza dell'idea appena presentata.

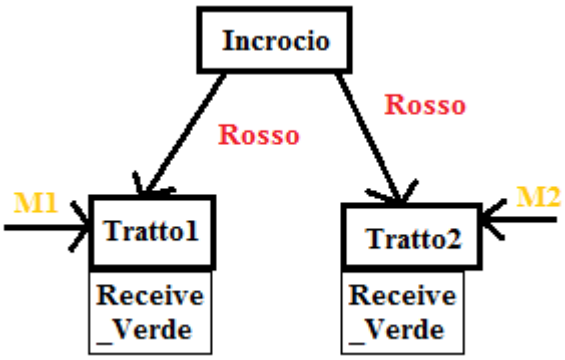


Figura 7: Semaforo 1

Un attore inizia con il 'receive' relativo al semaforo verde. Questo legge un mezzo ricevuto e lo rimanda al tratto o alla strada di destinazione. Supponiamo ora che l'attore stia ricevendo dei mezzi ed un messaggio 'Semaforo rosso' . All'arrivo di un messaggio 'Semaforo rosso' , questo verrà letto prima dei vari mezzi in arrivo.

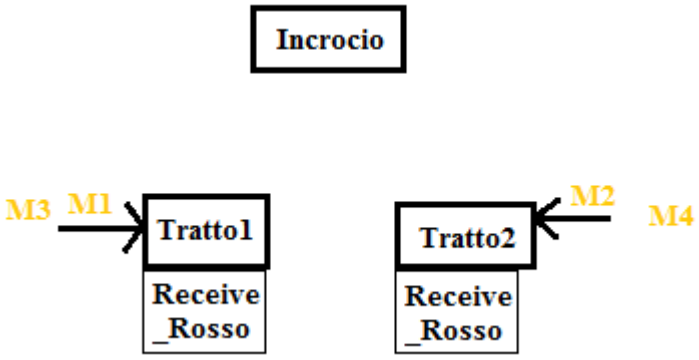


Figura 8: Semaforo 2

Quel messaggio farà cambiare mailbox all'attore. Gli altri messaggi non ancora gestiti verranno quindi letti con la nuova mailbox.

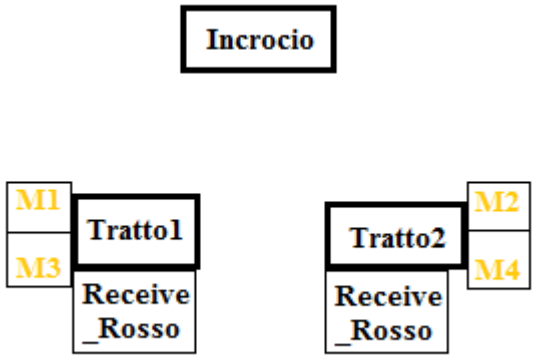


Figura 9: Semaforo 3

Tale mailbox ora ha il compito di salvare tutti i mezzi in arrivo, simulando quindi la coda che si forma quando c'è un semaforo rosso.

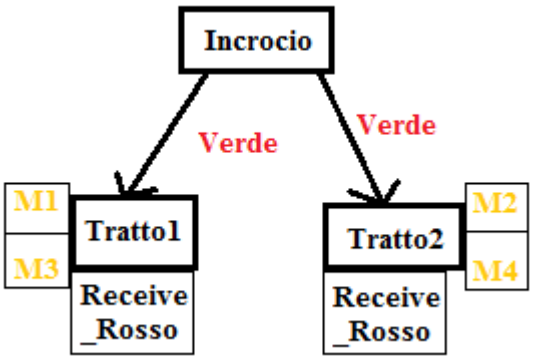


Figura 10: Semaforo 4

Supponendo ora che arrivi un messaggio 'Semaforo verde' , lo stato dell'attore cambierà nuovamente.

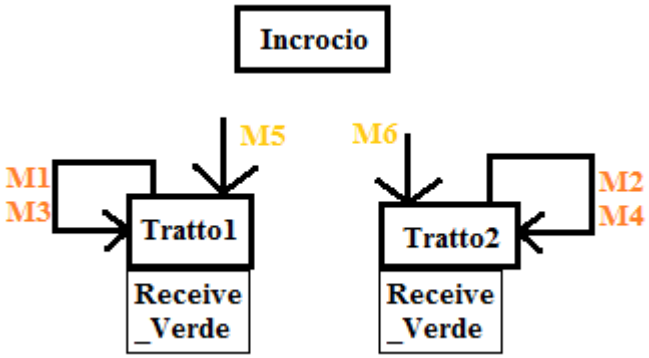


Figura 11: Semaforo 5

Siccome i messaggi già letti non possono più essere letti dalla prima mailbox, dovranno essere rimandati. Per fare funzionare il tutto, i mezzi già accodati verranno rispediti con una priorità un po' più alta. Dunque prima di terminare la gestione del messaggio relativo al semaforo, si dovrà inviare all'attore stesso TUTTI i mezzi accodati. Nell'esempio dunque, 'Tratto1' gestirà prima 'M1' ed 'M3' in quanto hanno priorità media ('M1' verrà gestito per primo in quanto una 'BoundedStablePriorityMailbox' garantisce il corretto ordinamento per i messaggi con la stessa priorità ricevuti dallo stesso mittente) per poi gestire 'M5' . Nel caso arrivasse un nuovo messaggio 'Semaforo rosso' , la situazione verrà

gestita come già spiegato all'inizio. Tutti i mezzi non ancora gestiti verranno quindi letti con la seconda mailbox e verranno accodati.

La gestione dei semafori inoltre risulta molto semplice. Basta solo impostare un timer per poter inviare a se stesso un messaggio per fare scattare i semafori. Tale messaggio però dovrà essere letto subito (e dovrà quindi avere una maggiore priorità rispetto a mezzi e pedoni), quindi ci sarà bisogno di una 'PriorityMailbox' anche per l'incrocio.

La progettazione è stata fatta in modo tale da rendere il sistema più scalabile possibile. Esso infatti funziona con una quantità qualsiasi di strade, di fermate, di mezzi e di persone. Questo perché ogni componente dovrà comunicare solamente con un'altra entità.

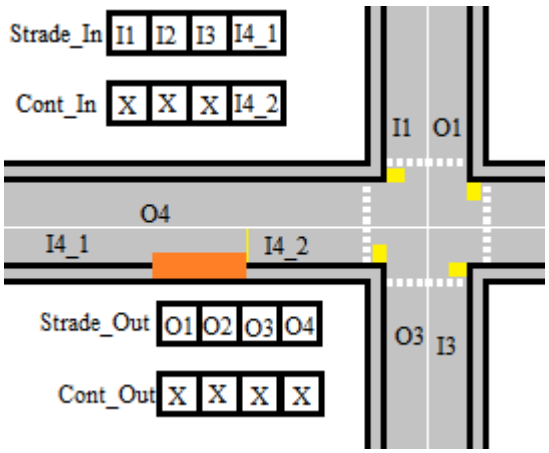


Figura 12: Strade 2

Ogni strada infatti dovrà solo guardare se nella STESSA posizione della lista Cont_In o Cont_Out c'è o meno un riferimento. Ovviamente ci dovrà essere un controllo per evitare che la strada 'I4_2' continui a mandare un messaggio sempre a se stessa. Dato che si era deciso di permettere al massimo una divisione per una strada, basterà controllare, ad esempio, se l'identificativo della strada contiene '_2' alla fine. In tal caso comunicherà con quella strada, altrimenti comunicherà con l'incrocio o con la zona vicina a seconda se la strada è 'entrante' oppure 'uscente' . Basterà quindi, durante la creazione delle strade, impostare correttamente i destinatari dei messaggi. Ogni zona potrà quindi avere un numero arbitrario di strade e ci potrà essere fino ad una fermata per ogni strada.

Come già detto inoltre, non è necessario l'utilizzo di liste per poter salvare mezzi e pedoni. Tali code saranno infatti gestite automaticamente dalle mailbox dei vari attori (che gestiscono in maniera automatica l'ordinamento. Akka infatti garantisce la lettura ordinata dei messaggi ricevuti dallo stesso mittente).

Tale progettazione inoltre non comporta accodamenti forzati. Nessuna componente del sistema infatti deve aspettare che altre componenti finiscano il loro lavoro per poter continuare il proprio (ad eccezione dell'incrocio con la gestione delle precedenze).

### 4.1.3 Distribuzione

- Sincronizzazioni iniziali
- Caduta zone

Per la realizzazione del sistema si vuole utilizzare il Clustering di Akka. Ogni cluster rappresenta un insieme di nodi (la cui posizione non è necessariamente nota al programmatore). L'Akka Clustering è stato preferito rispetto all'Akka Remoting in quanto, con quest'ultimo, ci sarebbe l' obbligato a conoscere gli indirizzi IP di tutte le zone. Nonostante ogni zona venga assegnata ad un nodo diverso, tutte le zone fanno parte dello stesso Actor System. In questo modo, ogni zona può richiedere (tramite un semplice 'contex.actorSelection()' ) e ricevere i riferimenti alle zone vicine. Tali riferimenti verranno quindi utilizzati dalle varie componenti per permettere l'invio di mezzi e persone.

La normale comunicazione tra le varie zone, dato che avviene tramite messaggi, utilizza una semantica 'at-most-once' . Questo non va bene in quanto non è garantito l'arrivo e la conseguente lettura dei messaggi da parte delle altre zone. Prima di poter dunque eliminare definitivamente un mezzo da una zona, bisogna essere certi che quella vicina l'abbia ricevuto correttamente. Una semplice soluzione sarebbe quella di utilizzare il trait 'AtLeastOnceDelivery' , ma esso comporta la gestione automatica di ulteriori invii in seguito a dei fallimenti, non garantendo più quindi il corretto ordinamento dei messaggi. Si è optato quindi per l'utilizzo dei Futures. Ogni corsia o marciapiede dovrà aspettare una risposta dalla zona di destinazione prima di poter inviare il mezzo successivo. In caso la risposta non venga ricevuta in tempo, si dovrà ritentare con un nuovo invio. Ovviamente ogni zona dovrà essere in grado di scartare eventuali duplicati tramite l'uso delle liste 'mezziRicevuti' e 'personeRicevute' .

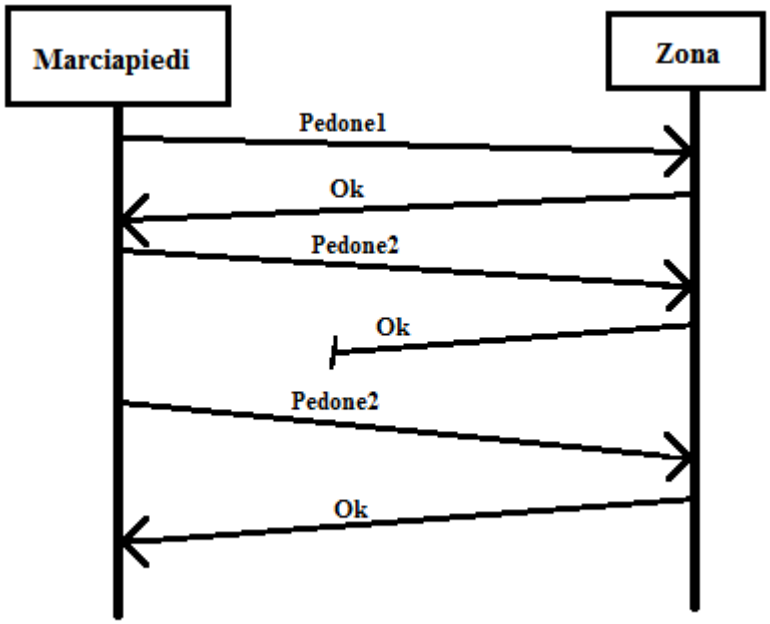


Figura 13: Invio pedoni

Questo può causare dei rallentamenti, ma consentirà al sistema di funzionare correttamente. L'importante è non scegliere un tempo di attesa troppo breve (non voglio inviare troppi messaggi inutili), ma nemmeno troppo esteso (non voglio causare troppi rallentamenti solo per aspettare un messaggio).

La progettazione è stata fatta in modo tale da rendere il sistema più scalabile possibile. Il sistema infatti funziona con una quantità qualsiasi di zone. Come già spiegato per la parte concorrente infatti, basterà passare i riferimenti corretti alle strade uscenti durante la fase di creazione per poter avere un numero arbitrario di zone.

### 4.1.4 Lista e descrizione classi

Aggiorno con VeicoloPipriorit, MezzoDeviato, ecc...

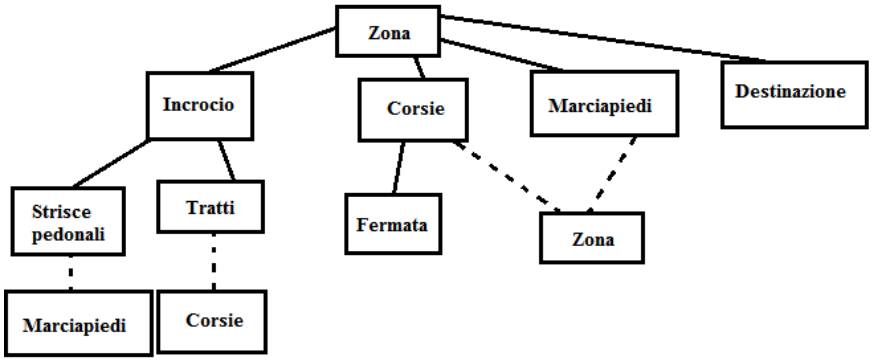


Figura 14: Schema

Di seguito verranno elencate le classi da utilizzare nel progetto e le loro descrizioni.

- Zona: Classe che estende 'Actor' e che rappresenta una zona. Riceve mezzi e persone e li passa all'attore corretto.
- Corsia: Classe che estende 'Actor' e che rappresenta una corsia. Tale corsia può andare da una zona (o da un'altra corsia) ad un incrocio o viceversa. Ogni corsia funziona grazie ad un gestore (uno per ogni corsia). Non appena tali gestori ricevono un mezzo, questi ne controllano il tipo e la destinazione ed inviano il mezzo dove necessario
- Fermata: Classe che estende 'Actor' e che rappresenta la fermata per gli autobus. Alcune classi (come 'Corsia' , 'Marciapiede' e 'Fermata' ) non hanno bisogno di liste di mezzi o di persone per funzionare correttamente. Tali classi infatti rappresentano degli attori con una propria mailbox. Tale attore riceverà molti messaggi, ma questi verranno accodati in automatico nella mailbox ed esaminati uno ad uno. Questo scelta è stata fatta in quanto Akka garantisce l'ordinamento automatico di messaggi che arrivano dallo stesso mittente. In questo caso una corsia riceverà mezzi sempre dallo stesso mittente (o dalla sua zona). Nel secondo caso, ci saranno messaggi provenienti dalla zona alternati ad altri. Questa scelta modella il fatto di avere auto che si immettono in una certa strada. Stesso discorso vale per la fermata, che riceverà mezzi sempre dalla stessa corsia.
- Marciapiede: Classe che estende 'Actor' e che rappresenta un marciapiede. Tale marciapiede può andare da una zona (o da un'altra marciapiede) ad un incrocio o viceversa. I marciapiedi funzionano allo stesso modo delle corsie, con la sola differenza che non è previsto la divisione dei marciapiedi.

- Destinazione: Classe che estende 'Actor' e che rappresenta l'arrivo a destinazione di un mezzo o di un pedone. Per poter simulare l'arrivo a destinazione, si è pensato di utilizzare una classe Destinazione. Essa assomiglia ad una specie di deposito dove i mezzi ed i pedoni arrivati a destinazione verranno inviati. Dopo un po' di tempo questi verranno inviati nuovamente alla zona, che li instraderà correttamente per poter tornare indietro.

- Incrocio: Classe che estende 'Actor' e che rappresenta un incrocio. Funziona allo stesso modo delle zone, ovvero riceve mezzi e persone e li passa all'attore corretto.

- Tratto: Classe che estende 'Actor' e che rappresenta i tratti dell'incrocio. Il secondo pezzo corrisponde ai 4 tratti interni all'incrocio. Essi funzionano in maniera simile alle corsie e alle strisce pedonali, con l'unica differenza che comunicano tutte tra di loro. Non appena ricevono un mezzo, lo manderanno ad un altro tratto. Inoltre 'Tratto' e 'StrisciaPedonale' avranno due stati (due metodi 'receive' ) per poter gestire correttamente gli invii, uno da usare quando il semaforo è rosso e l'altro da usare quando il semaforo è verde. Il primo riceverà i mezzi e li salva nella coda. Appena riceverà un messaggio (semaforo verde), cambierà stato. Il secondo dunque invierà i mezzi in coda ed eventuali nuovi mezzi ricevuti finch non riceverà un certo messaggio (semaforo rosso). In questo caso si ritornerà al primo metodo 'receive' . Per gestire correttamente i semafori e quindi fermare subito il trasferimento dei mezzi, l'attore necessita di una mailbox particolare. Sarà infatti necessario l'utilizzo di una 'BoundedStablePriorityMailbox' in quanto essa gestisce messaggi con priorità, garantendo al tempo stesso l'ordinamento corretto di messaggi con la stessa priorità. L'idea alla base dei semafori è la seguente: Messaggi 'semaforo rosso' e 'semaforo verde' avranno priorità massima Messaggi per l'invio dei mezzi avranno priorità minima All'arrivo di un messaggio 'semaforo verde' , l'attore invierà verso se stesso i mezzi accodati (in modo da essere letti con il secondo 'receive' ), questa volta però con priorità media. In questo modo i mezzi già accodati saranno guardati ed inviati prima rispetto agli ultimi arrivati, ma saranno comunque analizzati dopo eventuali messaggi relativi al semaforo All'arrivo di un messaggio 'semaforo rosso' , questo verrà letto subito, ed i mezzi che non sono ancora stati trasferiti saranno nuovamente accodati (prima quelli con priorità media, non cambiando quindi l'ordine della coda) subito dopo la fine dell'invio in corso

- StrisciaPedonale: Classe che estende 'Actor' e che rappresenta le strisce pedonali. Queste funzionano in maniera simile ai tratti ed alle corsie. Ricevono un pedone e lo mandano alle strisce pedonali successive (se questo deve effettivamente continuare). L'unica diffrerenza rispetto alle corsie sta nel fatto che le strisce pedonali funzionano o meno in base ai semafori. L'incrocio infatti avviserà le strisce pedonali quando dovranno cominciare a trasferire pedoni oppure quando dovranno smettere. 'Trat-

to' e 'StrisciaPedonale' avranno due stati (due metodi 'receive' ) per poter gestire correttamente gli invii, uno da usare quando il semaforo è rosso e l'altro da usare quando il semaforo è verde. Il primo riceverà i pedoni e li salva nella coda. Appena riceverà un messaggio (semaforo verde), cambierà metodo 'receive' . Il secondo dunque invierà i pedoni in coda ed eventuali nuovi mezzi ricevuti finch non riceverà un certo messaggio (semaforo rosso). In questo caso si ritornerà al primo metodo 'receive' . Invio e accodamento funzioneranno esattamente come quelli nella classe 'Tratto' .

- Persona: Trait che rappresenta una persona generica.
- PersonaDeviata: Classe che estende il trait 'Persona' e che rappresenta per una persona che deve compiere una deviazione a causa di una zona non raggiungibile.
- PersonaPiùPriorità: Classe che contiene una persona. Questa classe viene utilizzata nelle strisce pedonali per poter gestire correttamente l'ordine dei pedoni e rappresenta semplicemente un pedone con una priorità maggiore (come è già stato spiegato nella sezione 4.1.2).
- Pedone: Classe che estende il trait 'Persona' e che rappresenta per l'appunto un pedone.
- Mezzo: Trait che rappresenta un mezzo generico.
- MezzoPiùPriorità: Classe che contiene un mezzo. Questa classe viene utilizzata nei tratti per poter gestire correttamente l'ordine dei mezzi e rappresenta semplicemente un mezzo con una priorità maggiore (come è già stato spiegato nella sezione 4.1.2).
- MezzoDeviato: Classe che estende il trait 'Mezzo' e che rappresenta per un mezzo che deve compiere una deviazione a causa di una zona non raggiungibile.
- Auto: Classe che estende il trait 'Mezzo' e che rappresenta per l'appunto una automobile.
- Autobus: Classe che estende il trait 'Mezzo' e che rappresenta per l'appunto un autobus.
- Classi container: Sono classi utilizzate per poter distinguere correttamente dello stesso tipo che devono essere utilizzati in maniera diversa.
- MapActor: classe che facilita l'invio dei messaggi ad altre zone. Dato che ogni zona non sa in che cluster sono le altre,

### 4.1.5 Correttezza ordinamento messaggi

- Scrivo che ogni messaggio arriva prima di altri (aggiorno con la sincronizzazione iniziale)

Di seguito verr à analizzata la correttezza dell'ordinamento dei messaggi ricevuti dai vari attori. è importante infatti che i messaggi arrivino nel corretto ordine e che l'ordine dei mezzi e dei pedoni non cambi all'improvviso.

- corsiaOut -> zona: non importa l'ordine in cui una zona riceve i mezzi (e l'ordine da una certa corsiaOut è garantito)
- corsiaIn -> Incrocio: non importa l'ordine in cui l'incrocio riceve i mezzi (e l'ordine da una certa corsiaIn è garantito)
- strada -> continuazione strada: continuazione strada riceve mezzi solo da una corsia (sempre la stessa) e da una zona. Messaggi provenienti dalla zona possono mescolarsi a quelli provenienti dalla corsia (le macchine possono immettersi in strada da propriet à private ad esempio)
- tratto -> corsiaOut: corsiaOut riceve mezzi solo da un tratto (sempre lo stesso) e da una zona. Messaggi provenienti dalla zona possono mescolarsi a quelli provenienti dal tratto (le macchine possono immettersi in strada da propriet à private ad esempio)
- tratto -> se stesso: succede quando un semaforo diventa verde. Il tratto deve inviare prima i mezzi gi à in coda e non gli ultimi arrivati. Per fare ci ò viene usata un'apposita mailbox che gestisce le priorit à e al tempo stesso garantisce l'ordine per i messaggi con la stessa priorit à. In questo modo, appena il semaforo diventa verde, il tratto si invier à i mezzi in coda con una priorit à un po' pi è alta, cos ì da poter spedire i primi mezzi in coda e non gli ultimi arrivati
- corsia -> destinazione: non importa l'ordine in cui destinazione riceve i mezzi
- destinazione -> zona: non importa l'ordine in cui la zona riceve i mezzi
- marciapiedeOut -> zona: non importa l'ordine in cui una zona riceve i pedoni (e l'ordine da un certo marciapiedeOut è garantito)
- marciapiedeIn -> Incrocio: non importa l'ordine in cui l'incrocio riceve i pedoni (e l'ordine da un certo marciapiedeIn è garantito)
- striscie pedonali -> striscie pedonali: le striscie pedonali ricevono pedoni solo dalle strisce pedonali precedenti (sempre le stesse) e da un incrocio. Messaggi provenienti dall'incrocio possono mescolarsi a quelli provenienti dalle strisce pedonali senza problemi

- striscie pedonali -> marciapiedeOut: marciapiedeOut ricevono pedoni solo dalle strisce pedonali (sempre le stesse) e da una zona. Messaggi provenienti dalla zona possono mescolarsi a quelli provenienti dalle strisce pedonali senza problemi
- striscie pedonali -> se stesse: succede quando un semaforo diventa verde. Il tratto deve inviare prima i pedoni gi à in coda e non gli ultimi arrivati. Per fare ci ò viene usata un'apposita mailbox che gestisce le priorit à e al tempo stesso garantisce l'ordine per i messaggi con la stessa priorit à. In questo modo, appena il semaforo diventa verde, il tratto si invier à i pedoni in coda con una priorit à un po' pi è alta, cos ì da poter spedire i primi pedoni in coda e non gli ultimi arrivati
- marciapiede -> destinazione: non importa l'ordine in cui destinazione riceve i pedoni
- destinazione -> zona: non importa l'ordine in cui la zona riceve i pedoni

## 4.2 Funzionalità non implementate

- Motivazioni
- Salvataggio di stato ogni tanto (quando una zona torna su, dovrebbe avere un metodo di recupero dei mezzi che aveva)
- Grafica (creazione, ogni corsia, marciapiede, tratto e striscia pedonale ha un riferimento ad una label che modificherebbe man mano)

## 4.3 Screenshots

- Fermata autobus con pedoni che salgono e scendono
- Code ai semafori
- Caduta zone e deviazioni

# 5 Bibliografia


- [1] Scala: *http://www.scala-lang.org/documentation/*
- [2] Akka: *http://doc.akka.io/docs/akka/snapshot/scala.html*
- [3] SBT: *http://www.scala-sbt.org/*
- [4] Sistemi Concorrenti e Distribuiti: *http://www.math.unipd.it/∼tullio/SCD/2014/*
- [5] Routing: *http://en.wikipedia.org/wiki/List_of_ad_hoc_routing_protocols*