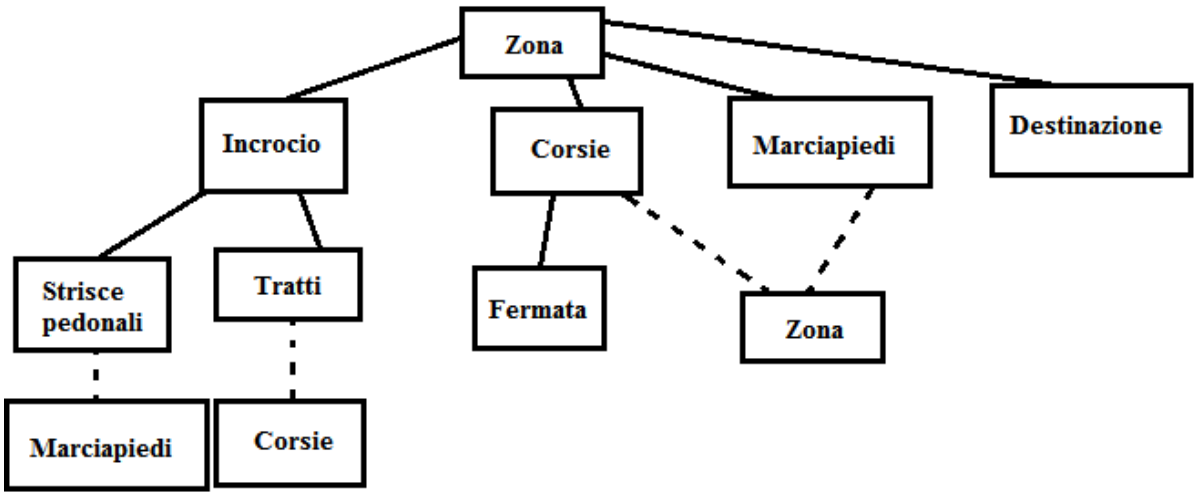
## Linguaggio

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 presente 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.

## Schema




## Descrizione classi

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.
Campi:

- String ID
- ArrayBuffer [String] mezziRicevuti: lista degli ultimi 30 mezzi ricevuti. Necessaria per evitare di creare mezzi doppi in seguito ad ACK non ricevuti
- ArrayBuffer [String] personeRicevute: lista delle ultime 30 persone ricevute. Necessaria per evitare di creare persone doppie in seguito ad ACK non ricevuti
- ArrayBuffer [ActorRef] vicini: lista contenente le zone vicine
- ArrayBuffer [ActorRef] corsiaIn: lista di corsie che ricevono mezzi da altre zone
- ArrayBuffer [ActorRef] corsiaInCont: lista contenente le "continuazioni" di strade divise
- ArrayBuffer [ActorRef] corsiaOut: lista di corsie che passano mezzi ad altre zone
- ArrayBuffer [ActorRef] corsiaOutCont: lista contenente le "continuazioni" di strade divise
- ArrayBuffer [ActorRef] marciapiedeIn: lista di marciapiedi che ricevono pedoni da altre zone
- ArrayBuffer [ActorRef] marciapiedeOut: lista di marciapiedi che passano pedoni ad altre zone

- ArrayBuffer [ActorRef] fermate:  lista di fermate per gli autobus

Funzioni:

- Unit start(…): crea la zona e tutte le sue componenti
- Unit ricevi(Mezzo): riceve un mezzo da una strada
- Unit ricevi(Persona): riceve una persona da un marciapiede
- Unit instrada(Mezzo): passa un mezzo alla corsia di partenza
- Unit instrada(pedone): passa un pedone alla corsia di partenza

### 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

Campi:

- String ID
- ActorRef nextActor: attore che serve per poter trasferire un mezzo ad una nuova zona, ad un pezzo di strada o ad un incrocio
- ActorRef fermata: attore che serve per poter trasferire un autobus ad una eventuale fermata
- ActorRef destinazione

Funzioni:

- Unit accoda(Mezzo): riceve un mezzo da una zona, da un incrocio o da una strada. Chiamerà subito *"passa"*
- Unit passa(Mezzo): passa un mezzo ad una fermata, ad un incrocio, ad una strada o ad un'altra zona
- Unit arrivo(Mezzo): passa un mezzo a destinazione

### 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.

Campi:

- String ID
- ActorRef marciapiede: serve per poter mandare le persone scese dall'autobus al marciapiede
- ArrayBuffer[Persona] in: lista di persone in coda per salire in una corriera
- ArrayBuffer[Persona] out: lista di persone appena scede da una corriera

Metodi:

- Unit fermataBus(Autobus): funzione per permettere la salita e la discesa di persone da un autobus
- Unit uscitaPedoni(ArrayBuffer[Persona]): manda le persone al marciapiede
- Unit fermataPedone(Persona): funzione per permettere ad una persona di aspettare un autobus

### 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.

Campi:

- String ID

- ActorRef nextActor: attore che serve per poter trasferire una persona ad una nuova zona o ad un incrocio
- ActorRef fermata: attore che serve per poter trasferire un autobus ad una eventuale fermata
- ActorRef destinazione

Metodi:

- Unit accoda(Persona): riceve una persona da una zona, da un incrocio o da una strada. Chiamerà subito *"passa"*
- Unit passa(Persona): passa una persona ad una fermata, ad un incrocio, ad una strada o ad un'altra zona
- Unit arrivo (Persona): passa un pedone a destinazione

### 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.
Campi:

- ActorRef zona

Metodi:

- Unit ricevi(Mezzo): riceve un mezzo e lo rimanda alla zona dopo un po' di tempo
- Unit ricevi(Persona): riceve un pedone e lo rimanda alla zona dopo un po' di tempo

### 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.
Campi:

- ArrayBuffer[ActorRef] tratti
- ArrayBuffer[ActorRef] strisce
- ArrayBuffer[Boolean] semaforiMezzi
- ArrayBuffer[Boolean] semaforiPedoni

Metodi:

- Unit start(…): crea l'incrocio e tutte le sue componenti
- Unit ricevi(Persona): riceve a persona da un marciapiede
- Unit passa(Persona): passa una persona  alle strisce pedonali
- Unit ricevi(Mezzo): riceve un mezzo da una strada
- Unit passa(Mezzo): passa un mezzo ad un tratto

### 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 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.
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

Campi:

- ArrayBuffer[ActorRef] tratti
- ActorRef corsia: corsia a cui mandare i mezzi
- ArrayBuffer[Mezzi] codaMezzi

Metodi:

- Unit ricevi(Mezzo): riceve un mezzo da una strada o da un altro tratto e lo inserisce nella coda
- Unit passa(Mezzo): passa un mezzo ad una strada o ad un altro tratto

### 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.
*“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 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 finché 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”.*
Campi:

- ActorRef strisciaSuccessiva
- ActorRef marciapiede: marciapiede a cui mandare le persone
- ArrayBuffer[Mezzi] codaPedoni

Metodi:

- Unit ricevi(Persona): riceve una persona dal marciapiede o dalla striscia pedonale precedente e la inserisce nella coda
- Unit passa(Persona): passa una persona alla striscia perdonale successiva

### Persona

Trait che rappresenta una persona generica.

### Pedone

Classe che estende il trait *“Persona”.*
Campi:

- String ID
- ArrayBuffer [String] percorso: insieme di stringhe che indica il persorso che il pedone deve seguire
- Int next: indice che indica la prossima destinazione del pedone

Metodi:

- String to(): ritorna la prossima destinazione del pedone

### Mezzo

Trait che rappresenta un mezzo generico.
Campi

- String ID
- ArrayBuffer [String] percorso: insieme di stringhe che indica il persorso che il mezzo deve seguire
- Int next: indice che indica la prossima destinazione del mezzo

Metodi:

- String to(): metodo che ritorna la prossima destinazione del mezzo

### Auto

Classe che estende il trait *"Mezzo"*.
Campi:

- Persona  guidatore: guidatore dell'automobile

### Autobus

Classe che estende il trait *"Mezzo"*.
Campi:

- ArrayBuffer[Persona] passeggeri: insieme di passeggeri all'interno dell'autobus
- Int Limit: numero massimo di passeggeri dell'autobus

Metodi:

- Unit faSalire(Persona): permette ad una persona di salire nell'autobus
- ArrayBuffer[Persona] faScendere(String): ritorna la lista delle persone che devono scendere ad una certa fermata. Riceve in ingresso l'identificativo della fermata per poter far scendere le persone corrette

## Correttezza ordinamento messaggi

Di seguito verrà 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.
Invio mezzi:

- 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 proprietà 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 proprietà private ad esempio)
- tratto->se stesso: succede quando un semaforo diventa verde. Il tratto deve inviare prima i mezzi già in coda e non gli ultimi arrivati. Per fare ciò viene usata un'apposita mailbox che gestisce le priorità e al tempo stesso garantisce l'ordine per i messaggi con la stessa priorità. In questo modo, appena il semaforo diventa verde, il tratto si invierà i mezzi in coda con una priorità un po' più alta, così 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

Invio pedoni:

- 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 già in coda e non gli ultimi arrivati. Per fare ciò viene usata un'apposita mailbox che gestisce le priorità e al tempo stesso garantisce l'ordine per i messaggi con la stessa priorità. In questo modo, appena il semaforo diventa verde, il tratto si invierà i pedoni in coda con una priorità un po' più alta, così 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

## Gestione 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 sono presenti variabili condivise tra gli attori: ogni attore ha 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 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 hanno priorità diverse. L'unica situazione in cui vengono gestite priorità diverse è quella della ricezione di messaggi relativi ai semafori. Se un tratto o una striscia pedonale riceve 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.
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. **AGGIUNGO ALTRO?**
Come già detto inoltre, non è necessario l'utilizzo di liste per poter salvare mezzi e pedoni. Tali code sono 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).
L'unica eccezione è stata fatta per gli incroci, ma questo è stato già spiegato in maniera esaustiva in precedenza.

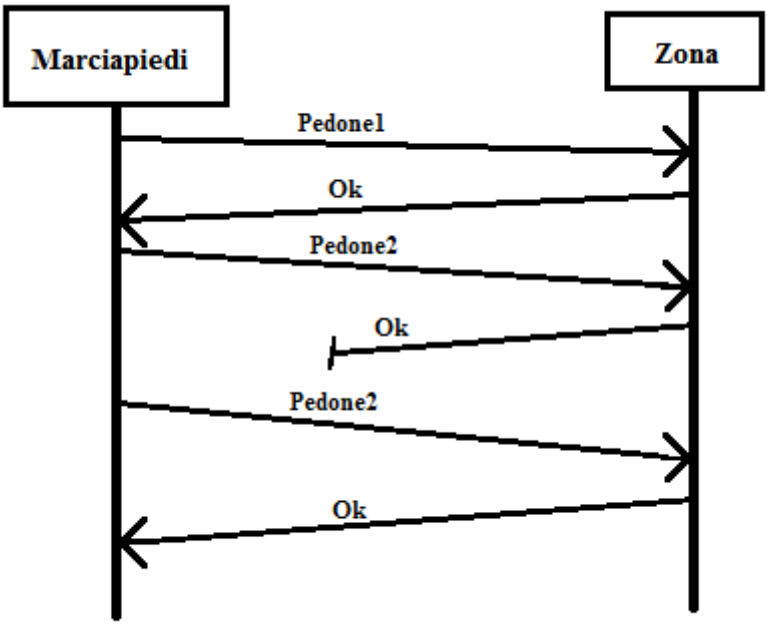
## Gestione sincronizzazione (SALVATAGGIO DI STATO?)

-

## Gestione distribuzione

Per la realizzazione del sistema è stato utilizzato 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, sarei 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 comunicazione tra le varie zone, visto che avviene tramite messaggi, utilizza una semantica "at-most-once". Questo non va bene in quanto non è garantito l'arrivo e la lettura dei messaggi da parte delle altre zone. Prima di poter dunque eliminare un mezzo da una zona, devo essere certo 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 l'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"*.




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.