Sincronizzazione

Di seguito verranno elencate le componenti con i relativi stati

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

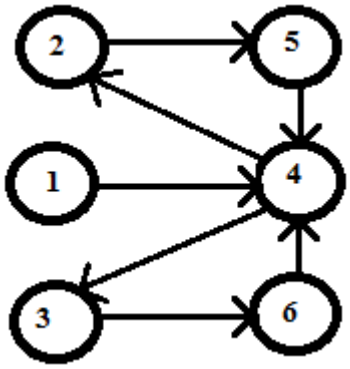
Tratto:

28. Invio mezzi
29. Ricezione mezzi
30. Attesa

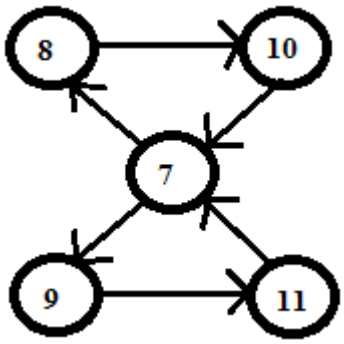
Striscia pedonale:

31. Invio pedone
32. Ricezione pedone
33. Attesa

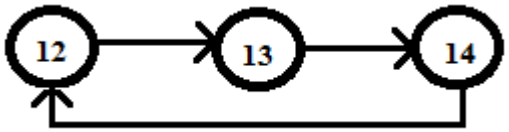
Zona



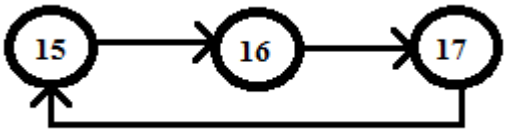
Incrocio



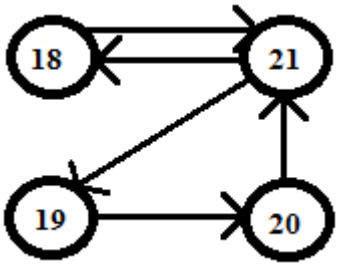
Corsia



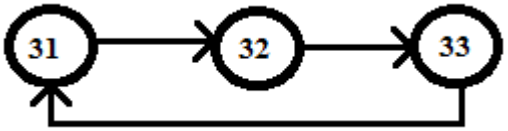
Marciapiede



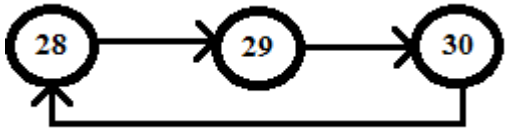
Fermata



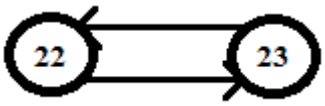
Striscia pedonale



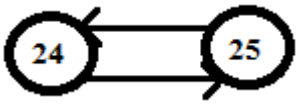
Tratto



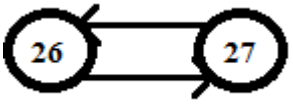
Autobus



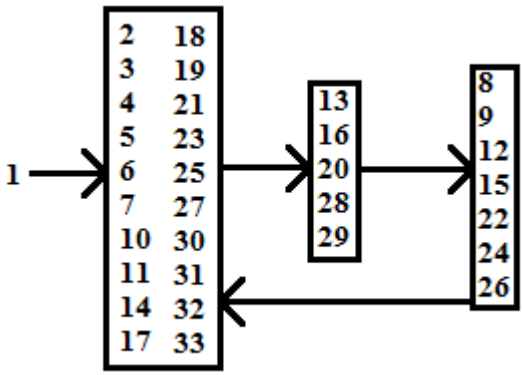
Automobile



Pedone



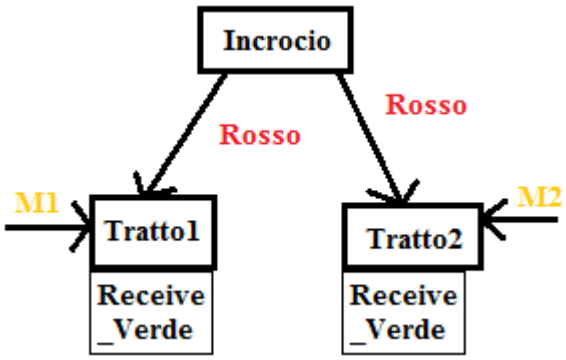
Aggregando ora le macchine a stati finiti trovate si ottiene:



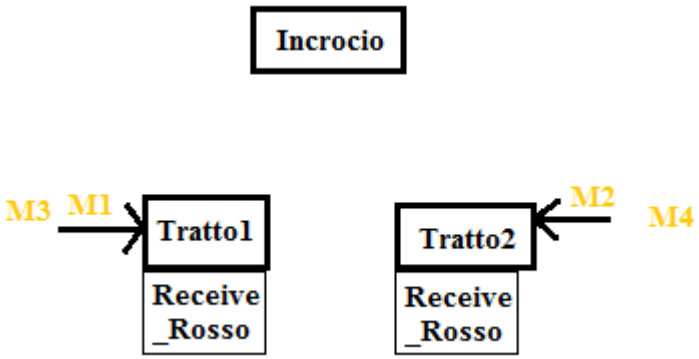
Concorrenza

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"*.

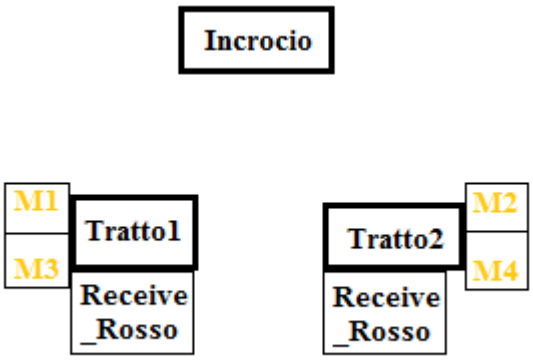
Per gestire correttamente i semafori e quindi fermare subito il trasferimento dei mezzi, l'attore necessita di una mailbox particolare. Sarà infatti necessario l'utilizzo di una "BoundedStablePriorityMailbox" in quanto essa gestisce messaggi con priorità, garantendo al tempo stesso l'ordinamento corretto di messaggi con la stessa priorità.
Di seguito si potrà vedere uno schema che dimostra la correttezza dell'idea appena presentata.



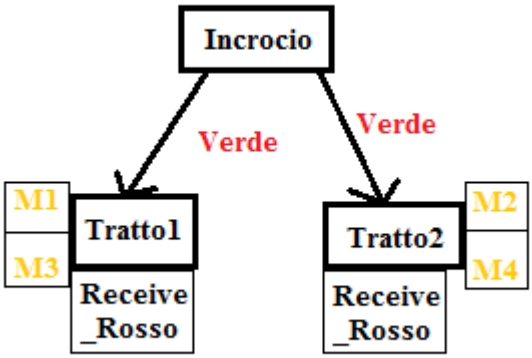
Un attore inizia con il "receive" relativo al semaforo verde. Questo legge un mezzo ricevuto e lo rimanda al tratto o alla strada di destinazione. Supponiamo ora che l'attore stia ricevendo dei mezzi ed un messaggio "Semaforo rosso". All'arrivo di un messaggio "Semaforo rosso", questo verrà letto prima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



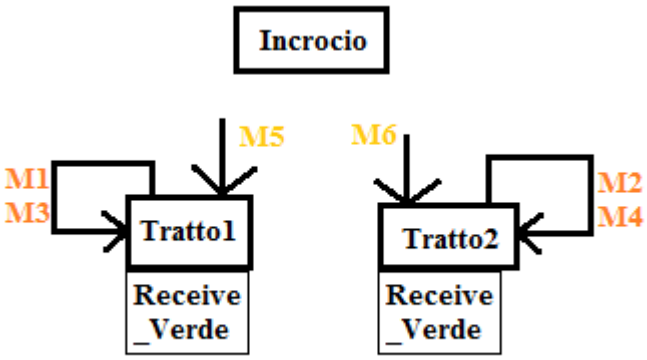
Quel messaggio farà cambiare mailbox all'attore. Gli altri messaggi non ancora gestiti verranno quindi letti con la nuova mailbox.



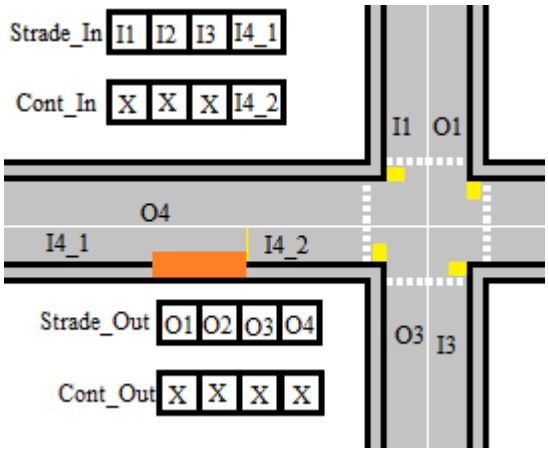
Tale mailbox ora ha il compito di salvare tutti i mezzi in arrivo, simulando quindi la coda che si forma quando c'è un semaforo rosso.



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



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 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.

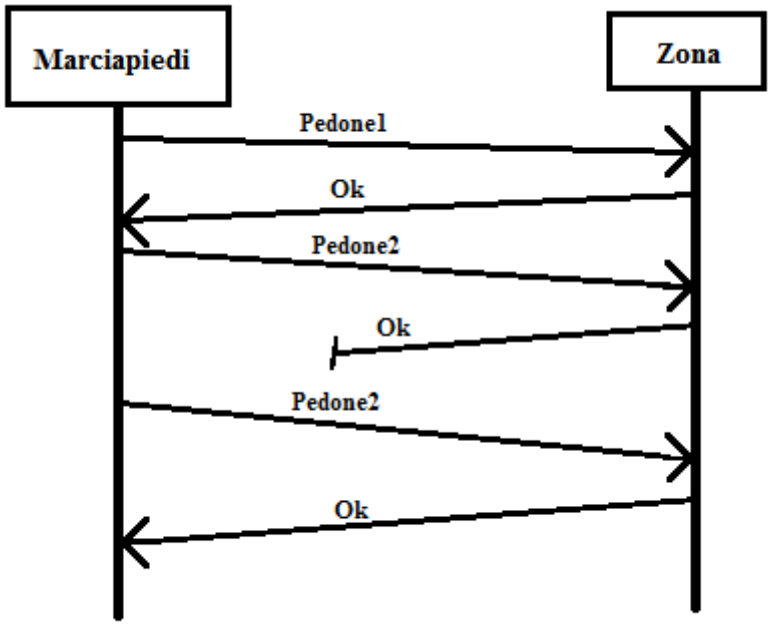


Ogni strada infatti dovrà solo guardare se nella STESSA posizione della lista Cont_In o Cont_Out c’è o meno un riferimento. In tal caso comunicherà con quella strada, altrimenti comunicherà con l’incrocio o con la zona vicina a seconda se la strada è “entrante” oppure “uscente”. Basterà quindi, durante la creazione delle strade, impostare correttamente i destinatari dei messaggi.

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).
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).

Distribuzione

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 comunicazione tra le varie zone, visto che avviene tramite messaggi, utilizzerà 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 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 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.