Progetto d’esame

Sistemi Operativi

PROGRAMMA MULTITHREAD PER LA SIMULAZIONE DEL TRAFFICO AUTOMOBILISTICO PASSANTE SU UN PONTE AD UNA SOLA VIA

Anno Accademico 2011/2012

Proposto da: **Alberto Arvizzigno**

Matricola n. 237953
E-mail: albertoa.arvizzigno@gmail.com

Indice

Specifiche del problema **3**

Analisi del problema **4**

Inquadramento 4

Scelte progettuali 6

Diagramma delle classi **9**

Schema a blocchi **11**

Implementazione **16**

Classe PonteAUnaVia 16

Classe Auto 17

Classe Ponte 18

Classe Coda 20

Testing **22**

Specifiche del problema

Allegate nel file PDF Specifiche_progetto_OS_estiva_2011-12.pdf

Analisi del problema

Inquadramento

Dall'analisi delle specifiche proposte viene richiesta l'implementazione di un'applicazione multithread scritta in Java che sia in grado di simulare il traffico di automobili passanti su un ponte.
Il numero **m** di automobili totali generate che dovranno effettuare il passaggio è corrispondente al numero di thread che dovranno essere attivati, inoltre tale numero deve essere definito da una variabile scelta dall'utente, come pure la quantità di automobili che possono salire contemporaneamente sul ponte seguendo la stessa direzione, viene infatti simulato che il ponte possa reggere un peso pari a **n** automobili.

Si dovranno dunque acquisire i seguenti elementi:

INPUT:

Il numero di auto da gestire, equivalenti al numero di thread in esecuzione non possono essere determinate a priori così come il numero di auto che possono passare contemporaneamente sul ponte questi dati devono essere pertanto inseriti dall'utente tramite input specifico.

- Numero di macchine totali che devono essere generate, ogni macchina avrà una direzione casuale ed un tempo di attesa iniziale casuale variabile tra 10 ms e 20 ms.

- Numero massimo di macchine che possono passare contemporaneamente sul ponte se percorrono la stessa direzione.

Si dovranno visualizzare a schermo o su file i seguenti elementi:

OUTPUT:

- Tempo di attesa di ogni automobile (ossia il tempo che intercorre tra l'arrivo al bordo del ponte, subito dopo l'attesa casuale generata alla creazione dell'auto ed il tempo di ingresso dell'automobile sul ponte)

- Tempo di attesa MEDIO che deve essere differenziato per le auto che viaggiano in direzione NORD_SUD e SUD_NORD.

Il divieto nell'uso del costrutto *syncronized* impone il dover fare uso esplicito ed accurato di tutti gli strumenti della classe *java.util.concurrent*, di cui fanno parte alcuni meccanismi descritti in seguito per una gestione esplicita delle concorrenze.
Tali meccanismi permettono di poter creare un flessibile costrutto *monitor*, efficace strumento per la risoluzione del problema proposto.

In particolare nelle recenti versioni di Java viene offerto un meccanismo che consente l'utilizzo di oggetti di tipo *Lock*, che permettono di controllare esplicitamente la mutua esclusione dei differenti thread al momento dell'accesso a sezioni di codice condivise in cui possono verificarsi situazioni di *race condition*.

Tramite gli oggetti di tipo *lock* è possibile dichiarare variabili di tipo *condition* su cui eseguire metodi quali *await(), signal()e signalAll()*. Tali metodi sono usati rispettivamente per bloccare un thread, per svegliare un thread sospeso sul metodo *await()* o per svegliare tutti i thread sospesi.

Il progetto richiede inoltre una gestione di tipo FIFO per le auto che dopo essere state differite di una tempistica casuale all'arrivo sul bordo del ponte, devono mantenere l'ordine costituito venendo gestite in ordine all'ingresso e quindi necessariamente anche all'uscita dal ponte.
Per il mantenimento della FIFO, nessuno dei metodi di gestione delle concorrenze sopraesposti è in grado di gestire esplicitamente la coda di arrivo, ogni elemento della coda deve quindi essere gestito singolarmente.

Scelte progettuali

Per la risoluzione del problema è stato utilizzato il linguaggio Java (version "1.6.0_31") e si è fatto uso dell'ambiente di sviluppo NetBeans (versione 7.1.1) funzionante su SO linux Ubuntu. L'uso di Java, come peraltro previsto da specifiche iniziali, è particolarmente adatto per la presenza di librerie per la gestione dei *thread*. Il codice creato si esegue su una macchina virtuale, una JVM (*Java Virtual Machine*) a cui viene passato il *bytecode* generato da un apposito compilatore che lavora sul codice sorgente.
Il linguaggio Java è di tipo *Object Oriented*, indipendente dall'architettura e facilmente portabile su differenti sistemi operativi, il linguaggio è inoltre corredato da un'estesa libreria di classi che sono organizzate in package tra cui le sopracitate classi di gestione multithreading. Per poter sfruttare al meglio le potenzialità offerte del multithreading è necessario disporre di procedure che permettono la sincronizzazione e la comunicazione tra thread affinché questi possano collaborare.

Per simulare l'arrivo coordinato delle auto prodotte con un ritardo casuale iniziale, si è scelto di inserirle in una coda subito dopo la creazione, simulata da un *array* che assume una dimensione uguale al numero di auto totali. Ad ogni auto viene assegnato un valore indicante l'ordine di arrivo, tale valore dovrà poi essere rispettato per la gestione della FIFO all'interno del ponte. L'*array* è inoltre definito staticamente, la dimensione è infatti conosciuta a priori in quanto inserita dall'utente.
Il ritardo iniziale è simulato alla creazione di ogni auto tramite l'uso del metodo *Thread.sleep()* , tuttavia è rilevante il fatto non c'è garanzia nella determinazione di quanto a lungo il thread dorma realmente prima di essere inserito nell'*array*, in quanto il suo risveglio verrà determinato solo per tempi multipli del timer interrupt e potrebbe non combaciare con il tempo di sleep realmente richiesto (prerogativa ad esempio di un sistema operativo *hard real-time* che è costretto a terminare la determinata operazione nel tempo prestabilito pena l'accettazione del processo).
Ne risulta che per due *thread* creati A e B, anche se al *thread* A viene assegnato un tempo di ritardo iniziale di poco inferiore al thread B, per i motivi sopra discussi potrebbe capitare che sia il thread B in realtà ad entrare per primo nella coda. Tale argomentazione, al fine della simulazione non comporta un fattore determinante in quanto l'intenzione è incentrata nel verificare l'entrata casuale dei thread sul ponte ed il successivo mantenimento della FIFO per l'esecuzione del programma.

La coda iniziale viene creata utilizzando un buffer con lo scopo di ottenere una simulazione più realistica e poter rappresentare il bordo del ponte sul quale si accodano le automobili prima dell'ingresso.
Si sarebbe potuto utilizzare un *ArrayBlockingQueue*, che offre le stesse funzionalità di mutua esclusione per la bufferizzazione degli elementi nella coda senza passare alla definizione di una classe Coda non prettamente necessaria, tuttavia al fine di una programmazione il più possibile incentrata su meccanismi di funzionamento interni e sulle tecniche specifiche di sincronizzazione multithread in un ambiente di *race condition* si è preferito definire una classe a parte per l'acquisizione di una maggior chiarezza sui meccanismi e le strutture di funzionamento interne di un ambiente concorrente.
Grazie alla classe Coda, in cui sono stati implementati i metodi di inserimento e rimozione degli oggetti istanziati dalla classe Auto, i dati vengono passati sequenzialmente (in un ordine FIFO) al ponte tramite il metodo bloccante di entrata: *entra()*.

Il ponte ha la funzione di *monitor,* ossia un costrutto di sincronizzazione in grado di rendere mutualmente esclusivo l'accesso a determinate risorse.
Il ponte opera quindi gestendo e filtrando le auto che possono attraversarlo e le auto che invece devono essere messe in uno stato di *await*() in quanto momentaneamente non idonee all'attraversamento. Per un maggiori dettagli riguardanti l'implementazione dei metodi si rimanda alla sezione di Implementazione.

La coda generata tramite l'uso della classe Coda, non ha un effettivo utilizzo specifico all'interno del monitor, dunque per la gestione delle operazioni sui thread in ordine FIFO si è scelto d'impiegare un array di variabili Condition tramite l'uso del metodo *signal().* Ad ogni condition è associato un'unica auto.

Una delle caratteristiche della classe java.util.concurrent.* del linguaggio Java è che permette di dichiarare variabili Condition utilizzando oggetti di tipo *Lock*, questo implica che all'interno del monitor un thread messo in *await()* rilasci anche il lock acquisito all'ingresso. Il *lock* verrà poi riacquisito al risveglio del *thread*.

La posizione dell'auto nella coda di attesa all'interno del monitor, viene determinata dalla variabile indice con cui viene gestito l'array.
Grazie a questo sistema è possibile risvegliare i thread con una chiamata del tipo :

- attesaCondizionale [auto.ordineDiArrivo].signal()

in cui il *thread* da risvegliare viene trattato esplicitamente tramite il valore della sua posizione contenuto in auto.ordineDiArrivo, generato al momento dell'inserimento dell'auto nella Coda.

In questo modo il vettore di *condition* assume una funzione di *waiting queue* in cui vengono inserite le auto che devono momentaneamente attendere per poter attraversare il ponte. La gestione dell'attesa non sarà di tipo attivo (*spinlock*).

Ricapitolando le uniche primitive di sincronizzazione necessarie/impiegate per la costruzione del monitor del ponte e per l'inserimento/rimozione delle auto nella Coda sono:

- *Lock* rientranti
- *Condition* applicate al lock

Queste sono non solo risultate adeguate ma anche sufficienti per la corretta gestione delle concorrenze e per la sincronizzazione dei thread del programma.
In particolare sono stati impiegati ai seguenti scopi:

- *Reentrantlock* per accedere in modo *thread-safe* sia alla coda di auto sia al ponte che poi effettuerà la gestione effettiva delle auto;

- Vettore di variabili *Condition* posto su ogni singola auto, con un ciclo *while()* per l'accesso al ponte, mette in attesa in ordine FIFO le auto all'interno del ponte. L'array viene utilizzato anche per il risveglio esplicito delle auto qualora richiesto dalle auto uscenti dal ponte.

Per quanto riguarda l'INPUT, i dati vengono acquisti, come richiesto da specifiche, tramite l'inserimento di dati effettuato dall'utente, in particolare vengono richiesti:

- Il numero di auto totali che dovranno passare dal ponte;

- Il numero di auto che il ponte può contenere nello stesso momento;

Viene effettuato un controllo sulla validità dei dati inseriti per assicurarsi che siano di tipo numerico e che il valore sia accettabile (>0). Nel caso di un errore di battitura e di un inserimento di un carattere al posto di un numero, il programma setta in automatico alcuni valori di default che manderanno comunque avanti la simulazione, è possibile verificare la soluzione implementata nella sezione testing.
Nulla vieta che il numero di auto che possono passare sul ponte sia maggiore del numero di auto totali che vi dovranno passare, in quanto i due dati non sono logicamente correlati.

Per quanto riguarda l'OUTPUT si è scelto di visualizzare i risultati su shell anziché salvarli su file, per avere un riscontro più diretto dei risultati.
Come anche è possibile che il ponte sia simulato guasto e quindi possa in quel momento ospitare 0 automobili, oppure che semplicemente non vi siano automobili in attesa. I casi limite e specifici sono stati trattati separatamente nella stesura del codice.

Il risultato è espresso nella forma:

Il tempo di attesa dell'auto Automobile_X con direzione DIR_AUTO è di Y

Inoltre al termine della simulazione e del passaggio di tutte le auto dal ponte, viene visualizzata la media di attesa suddivisa per le direzioni di provenienza, il risultato è espresso nella seguente forma:

Tempo di attesa medio per le auto con direzione NORD_SUD XXX

Tempo di attesa medio per le auto con direzione SUD_NORD YYY

Diagramma delle classi

La scelta delle classi implementate è stata veicolata dalle specifiche assegnate per il progetto e dalla considerazioni effettuate nelle scelte progettuali.
Il programma è stato ideato implementando quattro differenti classi:

- *PonteAUnaVia:* contiene il metodo main(args: String[]) dal quale parte l'esecuzione di tutto il programma;

- *Auto*: ogni auto corrisponde ad un thread e sarà acquisita dal ponte tramite la coda;

- *Ponte*: oggetto condiviso dalle auto, in grado di gestire le auto in arrivo facendole passare o bloccandole sospendendo temporaneamente il thread fino al verificarsi della condizione richiesta;

- *Coda*: in cui vengono inserite le auto in arrivo subito dopo che sono state prodotte per la successiva gestione.

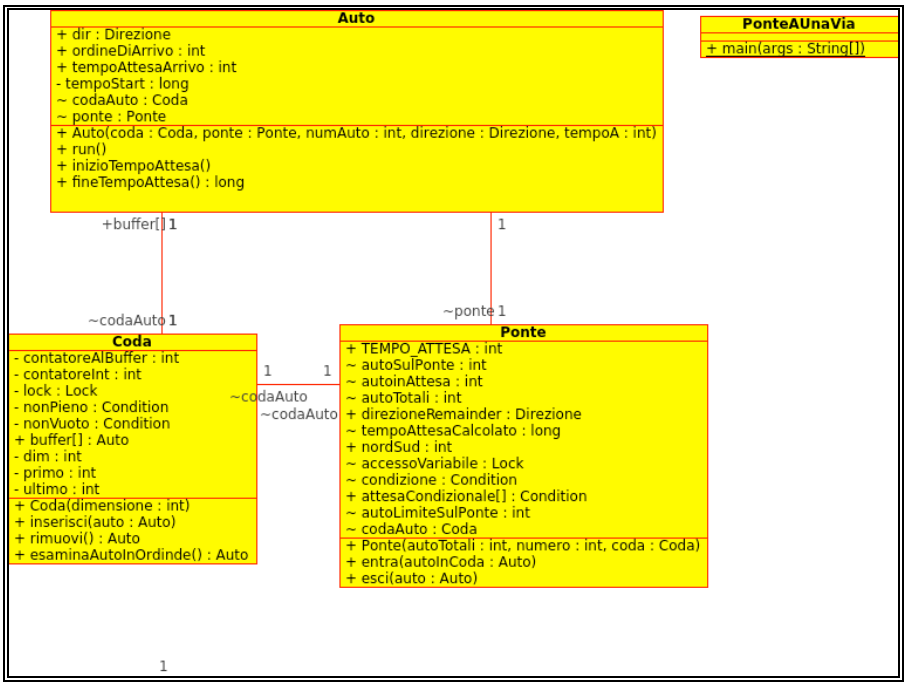

Figura 1

Dal diagramma proposto è possibile osservare le dipendenze che sussistono tra le varie classi che sono state sviluppate. In particolare la classe Ponte istanzia un oggetto condiviso, infatti il costruttore per la classe Auto acquisisce come parametro il ponte in uso.
Ad ogni auto viene passata inoltre la coda in cui viene salvato l'ordine di arrivo, che costituisce un secondo oggetto condiviso, non prettamente indispensabile ma che viene creato per simulare l'ordine di arrivo sul bordo del ponte e che viene utilizzato come buffer per il salvataggio iniziale delle auto nella FIFO.
La stessa classe Ponte è quindi necessariamente in comunicazione con la coda in quanto è da questa che riceve l'auto che poi dovrà essere processata.

I metodi e le variabili usate dalle classi verranno descritte in maggior dettaglio nei paragrafi successivi. Si nota comunque la presenza dei due metodi principali per la gestione del passaggio delle auto sul ponte richiesti da specifiche quali *entra() ed esci(),* i metodi utilizzati per il calcolo dei tempi d'attesa della classe Auto ed il metodo *run()* sempre della classe Auto che permette la generazione del *thread*.

Schema a blocchi

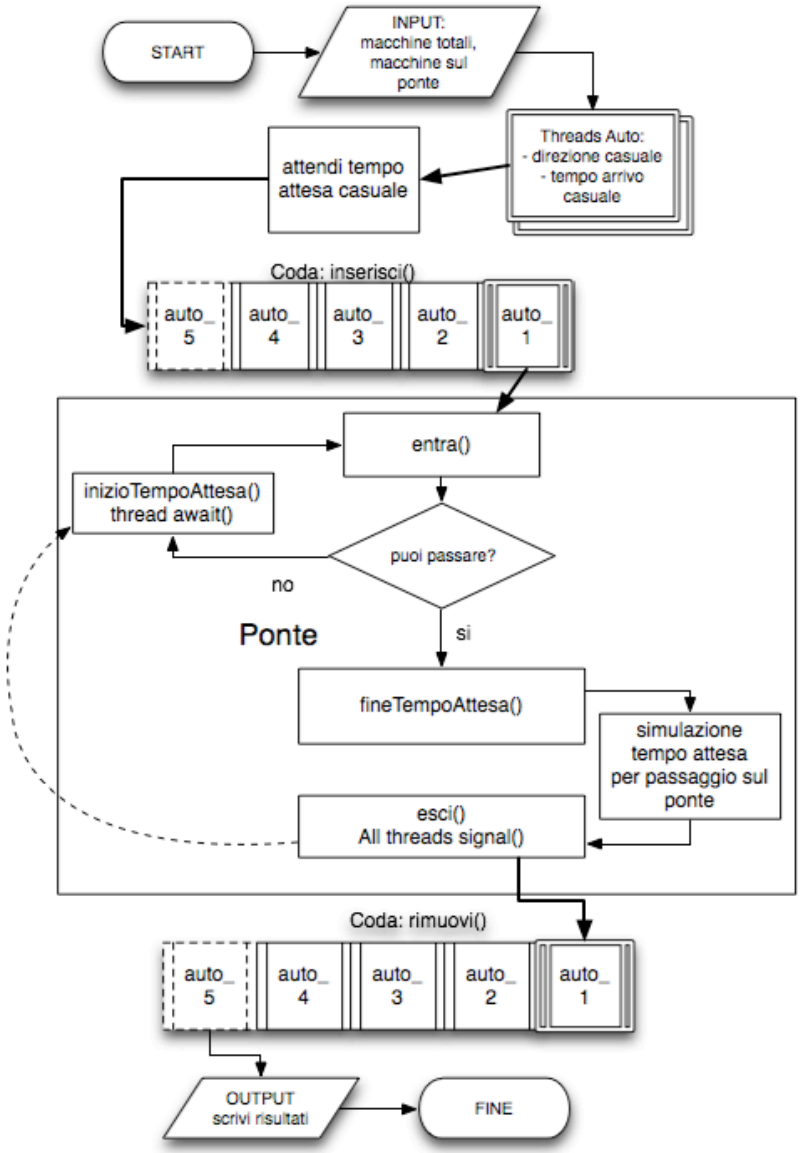

Figura 2

Le auto, dopo essere stare istanziate acquisiscono l'oggetto condiviso ponte, che ha la funzione di un costrutto di tipo *monitor*. Tutti i thread che vogliano utilizzare il monitor-ponte devono quindi acquisire il *lock* di accesso. I metodi e le operazioni dichiarate all'interno del ponte sono contraddistinte dalla mutua esclusione e solo un thread alla volta potrà operarvi attivamente in uno stato *running*.
Le auto all'interno del *monitor* ponte potranno assumere solamente *tre stati* in quanto: o il thread è attivo (*running*) in quel momento all'interno del monitor, oppure si trova nella FIFO di attesa (*waiting*) , oppure si troverà nello stato di *ready* indicante il processo pronto per l'esecuzione ma non ancora attivo all'interno del *monitor*.
Il passaggio sul ponte, inteso come simulazione del tempo di attesa, può comunque essere effettuato da più macchine contemporaneamente. Se ad esempio due macchine provengono

da nord, sono sequenzialmente in ordine di arrivo ed il ponte è in grado di ospitarle, entrambe possono passare il ponte senza attendere.
Per poter rispettare questa policy, la simulazione del tempo di attesa di 100 ms, che di fatto non permetterebbe a più thread di essere in attesa sul ponte in quanto questi devono rilasciare il *lock*, pur essendo mantenuta all'interno del *monitor,* viene implementata utilizzando il metodo *await(time)* della variabile *condition* impostando il time a 100, in questo modo il *thread* si metterà in uno stato *waiting* rilasciando il *lock* ed uscendo di fatto dal *monitor*, risvegliandosi da solo dopo 100 ms e permettendo nel frattempo al prossimo *thread*, che diventerà attivo all'interno del *monitor*, di poter continuare la propria esecuzione e di essere eventualmente messo in attesa per 100 ms qualora si tratti di un'auto che rispetti le condizioni di passaggio.

Si sarebbe potuto anche spezzare il *monitor* in due parti, differenziando esplicitamente la parte in ingresso e in uscita ed inserendo tra i due blocchi il metodo di attesa dell'auto di 100ms. in modo che non fosse soggetto a mutua esclusione, ma si è preferito mantenere un unico blocco su cui operare.

E' da notare inoltre che le auto vengono messe in attesa per 100ms utilizzando l'indice di posizione *nell'array di condition*, in questo modo viene rispettata la policy di passaggio sequenziale del ponte in ingresso.

Altro caratteristica importante del monitor è che l'istruzione di *signal(),* eseguita all'interno del metodo di uscita *esci()* per il risveglio degli altri thread è l'ultima istruzione che viene eseguita prima *dell'unlock,* quindi il thread in uscita riattiva i *thread* in coda, se questa è non vuota, oppure lascia il *monitor*.
Il *thread* che effettua il *signal* dei processi in attesa non perde comunque il *lock*, che verrà mantenuto fino alla sua uscita dal monitor nell'istruzione successiva.

Per quanto riguarda il calcolo del valori dei tempi d'attesa di ogni auto, questi sono determinati dal tempo che intercorre dall'arrivo al bordo del ponte fino all'effettiva entrata sul ponte. Per determinare tale valore si è dunque effettuata la lettura di un tempo d'inizio e di fine attesa. Le letture sono state posizionate nei seguenti modi:

- *inizio tempo attesa*: arrivo dell'auto al bordo del ponte, subito prima dell'inserimento nella coda di attesa, qualora l'auto non fosse in grado di attraversare il ponte e venisse messa in uno stato waiting.

- *fine tempo attesa*: l'istruzione prima del passaggio del ponte implementata con la simulazione di attesa di 100ms.

Ci si potrebbe aspettare, che visto che il tempo di attesa sul ponte è di 100ms le auto restino in attesa per un multiplo di tale valore. La questione non sarebbe del tutto erronea ma devono essere studiati più in dettaglio i tempi di attesa delle auto.
Di fatto, gruppi di auto che possono passare insieme il ponte potrebbero avere un tempo di attesa inferiore ad un multiplo di 100ms, in quanto le auto che si accodano più tardi nel tempo, aspettano anche di meno ma sono comunque risvegliate tutte insieme alla prima auto in uscita.

In **figura nr.3** si mostra un esempio per la determinazione delle tempistiche di arrivo.

Supponiamo di avere 4 auto che arrivano con tempi differenti:

-L' auto A arriva al t=10 ha direzione NORD_SUD
-L'auto B arriva al t=12 ha direzione NORD_SUD
-L'auto C arriva al t=13 ha direzione SUD_NORD
-L'auto D arriva al t=14 ha direzione SUD_NORD

Numero massimo di auto accettate dal ponte=2

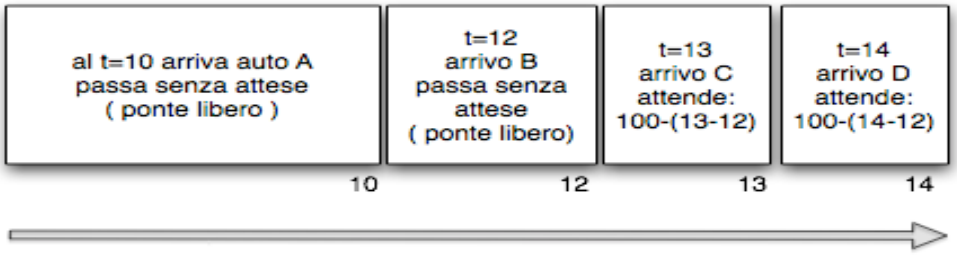

Figura 3

Le auto A e B che arrivano per prime con tempi t=10 e t=12, trovano subito il ponte libero per l'attraversamento, o meglio l'auto B troverà il ponte occupato dall'auto A ma essendo entrambe nella stessa direzione di percorrenza il ponte può accettarle entrambe, quindi A e B appena arrivano posso passare subito, ne consegue che il tempo di attesa sarà pari a zero.
L'auto C, quanto arriva al tempo t=13 trova il ponte occupato, tuttavia l'auto B che ha occupato il ponte per ultima sarà entrata 1 millisecondo prima dell'arrivo di C, quindi il tempo di attesa per C che deve aspettare che si liberi il ponte sarà di 99 ms. ossia il tempo di attraversamento che resta all'auto B.

L'auto D arriva al tempo t=14 e anch'essa trova il ponte ancora occupato dall'auto B, che sarà entrata sul ponte 2 ms prima (t=12), ne consegue che il tempo di attesa per D sarà di 98 ms, in quanto C e D entreranno sul ponte insieme e saranno risvegliate con lo stesso tempo di fine attesa, ma non lo stesso tempo d'inizio attesa. Ne consegue che i gruppi di auto che devono attraversare il ponte nella stessa direzione vedranno una diminuzione del tempo di attesa derivante dalla tempistica dell'ordine di arrivo, la questione è apprezzabile anche da verifica sperimentale.

Riassumendo quanto detto il Ponte costituisce di fatto un'implementazione di un meccanismo di sincronizzazione **tipico dei monitor** essendo costituito da lock per la mutua esclusione e da variabili condition. Grazie all'acquisizione del *lock* viene garantito che non vi è possibilità che due processi possano eseguire un *await() o un signal()* contemporaneamente o in modo non voluto.

Le auto all'interno del ponte, come accennato poco fa e come mostrato in figura nr.4, possono assumere tre tipologie di stato:

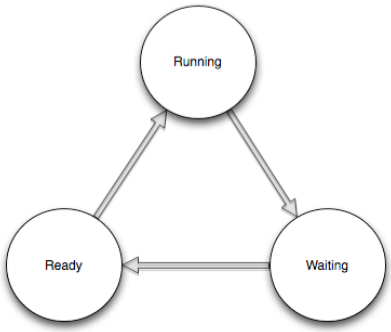

Figura 4

- **stato attivo** (*running*) in cui un solo *thread*, possessore del *lock*, procede nella sua esecuzione mentre tutti gli altri *thread* sono dormienti;

- **stato di attesa** (*waiting*)in cui il *thread*, inserito in una *waiting queue,* attende su una variabile *condition* tramite un metodo di *await().* In particolare tale metodo è posto all'interno di un ciclo *while()* all'inizio del quale vengono testate le condizioni di accesso al ponte.

- **stato pronto** (*ready*) è il caso in cui il *thread* viene risvegliato da una *signal(),* tuttavia il *thread* che effettua il risveglio possiede ancora il *lock* nella sezione critica. Quindi il thread risvegliato sarà spostato dalla *waiting queue* alla *ready queue* in quanto formalmente non aspetterà più sulla variabile condition ma aspetterà solo che si liberi il *lock* per poterlo acquisire. In questo caso il *thread* che segnala il risveglio agli altri resterà attivo fino all'uscita dal monitor.

Il comportamento dell'auto viene descritto dai seguenti passi proposti dalle specifiche:

- attesa di un tempo casuale tra 10ms e 20 ms effettuata al momento della creazione;
- richiesta di entrata sul ponte tramite il metodo *entra();*
- simulazione del tempo di attesa necessario al passaggio del ponte;
- notifica alle altre auto alla fine dell'attraversamento.

E' da notare che l'ultimo punto sopracitato è in realtà rispettata solo parzialmente, in quanto non risulta completamente necessario, infatti solo l'ultima auto, appartenente al gruppo di auto che attualmente stanno attraversando il ponte nel numero massimo previsto e nella

direzione di percorrenza coerente, lasciando il ponte in uno stato libero, riattiverà le auto bloccate, tramite l'uso di un *signal().* Sarebbe stato inutile far risvegliare le auto dormienti per opera di ogni auto in uscita in quanto queste, trovando il ponte occupato si sarebbero rimesse immediatamente a dormire.
Le auto bloccate verranno in questo modo risvegliate in ordine FIFO, poiché ogni auto ha una propria posizione *nell'array di condition* e la sua posizione in attesa è determinabile tramite l'indice, si ricorda che l'indice indicante l'ordine di arrivo viene determinato per ogni auto al momento della creazione e dell'inserimento nella coda.
Le auto risvegliate, solo qualora rispettino la policy di attraversamento del ponte inserita come condizione nel ciclo *while()*, potranno attraversarlo, in caso contrario saranno rimesse a dormire, nello stesso ordine, fino alla prossima *signal()* generata dell'ultima macchina uscente dal ponte, come descritto dallo schema a blocchi in figura nr.2.
Il ciclo si ripeterà fino all'esaurimento di tutte le auto in attesa generate all'inizio dell'esecuzione e presenti nella coda.

Implementazione

Segue una descrizione dettagliata delle classi, dei metodi e delle variabili di utilizzo implementate per la risoluzione del problema.

Classe PonteAUnaVia

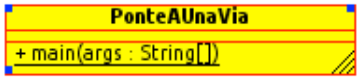

Figura 5

La classe PonteAUnaVia è la classe che contiene il main(), da cui parte l'esecuzione del programma. All'interno del main() è stato inserito il codice necessario all'esecuzione di alcune funzioni iniziali quali:

- l'inserimento ed al controllo della validità dei dati in input;
- la generazione della direzione casuale NORD o SUD che verrà passa all'oggetto auto;
- la generazione del valore di tempistica casuale di arrivo (tra 10 ms e 20 ms) che verrà passato come parametro all'oggetto auto;
- la scrittura dei dati di OUTPUT per quanto riguarda le medie dei tempi di attesa.

In particolare, come ulteriore scelta progettuale si è ricorso al main() e non direttamente alla classe Auto per:

- il calcolo dei dati casuali (direzione e tempo di arrivo) da passare ad ogni auto;
- il calcolo delle medie richieste da specifiche, in quanto vi era necessità di dover aspettare l'ultimo valore di attesa dell'ultima automobile.

Per il calcolo dei numeri casuali viene utilizzata la classe Random() importata dal package java.util.Random. La classe genera numeri pseudocasuali, utilizzando il timestamp di sistema, tali numeri sono distribuiti uniformemente all'interno dell'intervallo richiesto e poiché il timer è differente ad ogni chiamata, questo assicura una sequenza di valori differenti ad ogni avvio del programma.
Lo stesso generatore viene utilizzato sia per la produzione della direzione casuale utilizzando il metodo *nextBoolean(),* in quanto le direzioni di marcia possono essere solo due, nonché per la produzione dell'intervallo di 10ms tra compreso tra 10ms e 20ms nel qual caso viene utilizzato il metodo *nextInt().*

Classe Auto

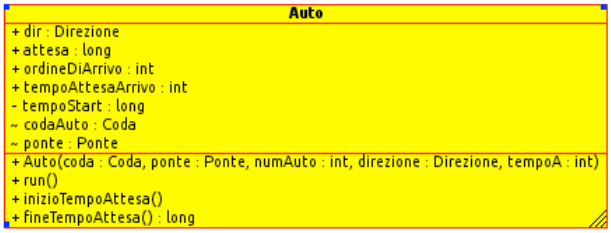

Figura 6

La classe *Auto* crea le istanze delle automobili, ogni auto è in realtà *un'estensione della classe Thread* ed implementa il metodo *run()* grazie al quale partirà l'esecuzione di ogni thread generato da un'istanza della classe.
Sebbene l'ereditarietà di Java non consente ad una classe di avere più di una classe padre è comunque possibile implementare un numero arbitrario d'interfacce, in tutte le situazioni in cui non sia possibile derivare una classe per estensione della classe Thread, può essere utilizzata l'interfaccia *runnable* (). Nel caso della classe Auto non si è posta tale necessità quindi si è preferito estendere semplicemente la classe auto con l'uso della classe *Thread* riscrivendo il metodo *run()*.

Ogni auto deve essere inoltre dotata di una direzione di marcia che può essere *NORD_SUD* oppure *SUD_NORD,* acquisita in modo casuale, e da un *tempoAttesaArrivo* anch'esso generato casualmente, che determina l'ordine di arrivo sul bordo del ponte.
I valori casuali vengono passati all'auto al memento della creazione dalla classe PonteAUnaVia.
L'ordine di arrivo viene poi salvato dalla variabile *ordineDiArrivo* all'inserimento dell'auto all'interno della Coda e verrà utilizzato per posizionare l'automobile in attesa con il metodo di *await()* o per poter risvegliare l'automobile con il metodo *signal()* secondo la FIFO.

Il metodo costruttore della classe acquisisce l'oggetto condiviso Ponte, e l'oggetto condiviso Coda che funziona da buffer di contenimento delle auto in arrivo, nonché i dati di direzione casuale e tempo di attesa casuale che gli vengono passati dal *main().*
La classe contiene inoltre i due metodi per il calcolo dei tempi di attesa al bordo del ponte di ogni automobile. Il tempo di attesa dell'auto, calcolato tramite il metodo *inizioTempoAttesa()* parte dall'istruzione antecedente all'esecuzione del ciclo *while(),* all'interno del quale sarebbe messa in pausa l'auto non passa all'interno del ciclo while() in quanto possiede già le caratteristiche per poter attraversare il ponte allora il tempo viene immediatamente fermato risultando 0, qualora invece l'auto entri in attesa il dato salvato poco prima verrà poi utilizzato all'uscita dell'attesa per il calcolo del tempo di attesa.

Il secondo metodo utilizzato per il calcolo del tempo di attesa è *fineTempoAttesa()* che calcola la differenza di tempo tra lo start e lo stop scrivendo il risultato nella variabile *attesa* propria della classe Auto.
Per poter determinare il tempo effettivo d'inizio e fine attesa viene utilizzato il metodo statico *currentTimeMillis()* appartenente alla classe *System* (come suggerito da specifiche).
E' da notare tuttavia, che la lettura del tempo in millisecondi non è esatta ma ricade in un intorno del tempo di lettura la cui granularità è determinata dal quanto di tempo assegnato del sistema operativo, inoltre dato che tale metodo per il calcolo del tempo di attesa è richiamato due volte, tale errore si propagherà nel calcolo del risultato finale e quindi

potrebbe risultare un valore non molto preciso, ma comunque abbastanza significativo al fine della simulazione.
Per quanto riguarda maggiori dettagli sulle specifiche del calcolo del tempo di attesa delle auto si rimanda alla sezione dello Schema a Blocchi.

Classe Ponte

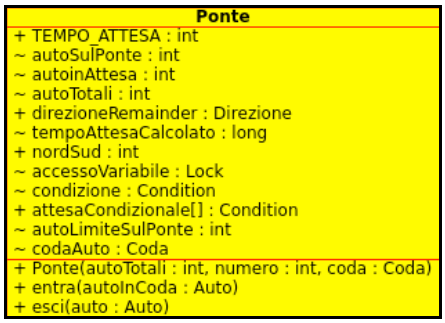

Figura 7

La classe Ponte serve per istanziare un oggetto condiviso tra le varie auto. E' composta principalmente dalle variabili che entreranno in gioco per la gestione dei *thread* quali *lock e condition* e dai due metodi previsti da specifiche quali *entra()* ed *esci()* che verranno richiamati dalle automobili per entrare ed uscire dal ponte.
Il metodo di entrata, come peraltro richiesto da specifiche, è stato implementato in modo da bloccare le auto su una variabile *condition* tramite il metodo *await().* La variabile *condition* viene dichiarata grazie ad un *lock* denominato *accessoVariabile*, il *lock* è di tipo rientrante ossia è possibile per lo stesso *thread* acquisire più volte la stessa primitiva di sincronizzazione.
Il *lock* accessoVariabile è gestito inoltre con l'opzione di *fairness* settata a TRUE, infatti in *Java* non vi è nessuna garanzia riguardo alla sequenza in cui i *thread* attendono sullo stesso *lock*, ci potrebbe essere il rischio che ad un *thread* non sia mai garantita l'acquisizione del *lock* il che potrebbe portare al verificarsi di una *starvation* (ossia una situazione di attesa indefinita tra i processi concorrenti) . La *fairness* impostata a TRUE impedisce invece il verificarsi di questa problematica garantendo l'accesso a tutti i thread nel modo più possibile vicino a quello di arrivo.

La sospensione ed il risveglio delle auto viene gestito proceduralmente all'interno di un ciclo *while()* le cui condizioni agiscono da filtro, lasciando passare solo le auto che rispettano le *policy* quali:

- l'auto che chiede l'attraversamento deve avere la stessa direzione di percorrenza momentanea del ponte oppure il ponte deve avere una direzione neutra (==null). La prima auto che attraversa il ponte, trovandolo vuoto, assegna a questo una direzione di percorrenza in modo che se anche le auto che seguono hanno la stessa direzione possano essere accettate per l'attraversamento. La policy permette quindi il passaggio del blocco di auto nella stessa direzione limitatamente al numero di auto massime accettabili dal ponte;

- non deve essere superato il limite massimo di auto che possono passare contemporaneamente sul ponte, in quanto il ponte può gestire un numero massimo totale inserito dall'utente, quindi sebbene le auto che seguono la prima abbiamo la

stessa direzione di percorrenza del ponte, potranno essere bloccate qualora il ponte risulti essere pieno;

- l'auto che richiede l'accesso non può avere altre auto in attesa prima di lei, ma deve essere trattata seguendo un ordine di arrivo. Se tre auto arrivano al bordo in modo alternato da direzioni opposte, le due auto di direzione uguale non possono passare insieme, ma la terza auto deve comunque aspettare il passaggio della seconda per poter accedere al ponte.

Come esempio, supponiamo che abbiamo tre auto richiedenti l'accesso e provenienti dalle seguenti direzioni:

- auto_1: NORD;
- auto_2 : SUD;
- auto_2 : SUD;

in questo caso l'auto_1 passerà immediatamente perché troverà subito il ponte libero mentre le auto auto_2 e auto_3, ammesso che il ponte le possa contenere entrambe, passeranno in ordine subito dopo che l'auto_1 è uscita dal ponte.
In questo caso quindi l'auto_1 avrà tempo di attesa 0 e le auto: auto_2 e auto_3 avranno un tempo di attesa di circa 100 ms (ossia il tempo necessario all'attraversamento della prima auto).
Per il tempo di attesa di 100 ms viene inizializzata la costante TEMPO_ATTESA della classe, tale costante è utilizzata come parametro per il metodo di await(TEMPO_ATTESA, TimeUnit.MILLISECONDS) che sta ad indicare che il valore è espresso in millisecondi.

Il numero di auto, in attesa che il ponte si liberi, viene salvato nella variabile *autoInAttesa*, incrementata ogni volta che un'auto entra nel ciclo *while()* e che pertanto rispetti le condizioni di attesa sopraesposte.
Il valore *autoInAttesa* sarà utile per sapere a quante altre auto dovrò effettuare una chiamata *signal* per consentirne il risveglio dallo stato di *waiting*.

Ogni *thread* che richiama la funzione di entrata *entra()* acquisisce un *lock* che permette la *mutua esclusione* all'interno del blocco *monitor* (delimitato da un *unlock*) per evitare situazioni di *race condition,* quindi tutte le variabili all'interno del blocco definito da un *lock* e da un *unlock* possono essere modificate da un solo *thread* alla volta, ossia il *thread* attivo all'interno del *monitor*, il *thread* inoltre rilascia il *lock* anche quando viene sospeso dal metodo *await()* della *condition*, questa caratteristica permette agli altri *thread* in attesa di poter acquisire il *lock* subito dopo, permettendo il progresso generale del processo.
Ogni *thread* che soddisfa le condizioni di attesa del ciclo *while(),* viene messo in uno stato dormiente grazie ad un *array di condition*, avente come indice la posizione salvata nella variabile *ordineDiArrivo.*
Questa metodologia permette di poter trattare esplicitamente la sospensione di ogni *thread* secondo l'ordine di arrivo, verrà poi utilizzato nuovamente per il risveglio, sempre in ordine, grazie al metodo *signal()*.
I *thread* in attesa vengono infatti risvegliati grazie al metodo *esci(),* ogni auto che abbia terminato l'attraversamento del ponte dopo la simulazione di 100ms, richiama il metodo di uscita dal ponte.
Ogni *thread* in uscita decrementa il contatore del numero di auto presenti sul ponte,

quando l'ultima auto del blocco di auto che hanno il permesso di attraversamento, termina il suo passaggio , questa risveglia tutte le altre auto in attesa tramite il suddetto metodo *signal()* richiamato *sull'array di condition*, in cui viene sfruttata la posizione nota della prima auto in attesa sulla FIFO e dell'ultima auto in attesa, ovviamente l'ultima auto in attesa corrisponderà sempre anche all'ultima auto presente nella FIFO, quindi il limite superiore del blocco da risvegliare sarà sempre lo stesso.
Il *thread* auto che richiama il metodo *esci(),* risvegliando tutte le auto in attesa, possiede anche il *lock* per poter essere nello stato *running* all'interno del *monitor*. Poiché all'interno del *monitor* è possibile che sia attivo un solo *thread* alla volta, come già precedentemente detto nella sezione dedicata allo Schema a Blocchi, l'ultima istruzione effettuata dal *thread running* prima dell'uscita dal *monitor* è proprio quella di risvegliare gli altri *thread* con un *signal()*, il *thread* che resterà attivo sarà comunque quello segnalante, quindi i *thread* risvegliati dovranno attendere *l'unlock* di tale *thread* per poter operare attivamente.
La classe è inoltre utilizzata per la stampa su schermo del risultato del tempo di attesa di ogni auto, in particolare il dato viene salvato nella variabile *attesa* della classe Auto prima d'iniziare la simulazione di attesa per l'entrata sul ponte.
L'output richiesto è espresso per ogni auto nella forma:

Il tempo di attesa dell'auto Automobile_X con direzione DIR_AUTO è di Y

Classe Coda

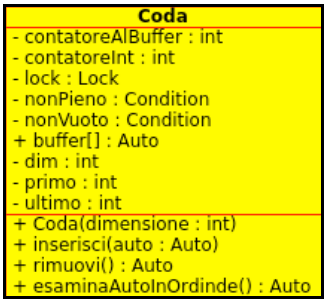

Figura 8

La classe Coda è stata implementata per simulare il bordo del ponte al quale si radunano in linea di arrivo (FIFO) le auto dopo aver atteso una tempistica casuale.
Ogni auto all'ingresso della coda riceve un ordine di arrivo, che verrà salvato nella variabile *auto.ordineDiArrivo,* tale variabile verrà poi utilizzata come indice per *l'array* di condition e risulta quindi importante per la determinazione della sequenza di risveglio delle auto.
Le auto poi sono estratte dalla coda singolarmente nell'ordine in cui sono state inserite e vengono passate al ponte per poter essere gestite secondo le *policy* in vigore.

Nella classe Coda sono presenti tre metodi oltre al metodo costruttore quali:

- il metodo d'inserimento *inserisci()* si occupa dell'inserimento dell'auto nella coda e viene richiamato all'interno del metodo *entra()* della classe Ponte subito prima dell'acquisizione del lock che servono per gestire il monitor, infatti la classe Coda viene gestita con dei lock propri.

- il metodo *rimuovi()* si occupa della rimozione dell'auto dalla coda all'uscita della stessa dal ponte, viene richiamato all'interno della metodo *esci()* prima dell'acquisizione del lock proprio della classe Ponte in quanto la classe Coda viene gestita con dei lock propri.

- il metodo *esaminaAutoInOrdine()* serve per la lettura degli elementi contenuti all'interno della coda che in ordine vengono poi passati al metodo di entrata della classe Ponte.

La gestione dei metodi avviene all'interno di un ambiente di mutua esclusione gestito tramite l'acquisizione di *lock* per evitare una condizione di *race condition*.
Valgono inoltre le stesse implicazione già discusse per la classe Ponte, per quel che riguarda la *fairness* del *lock* rientrante settata a TRUE.

Testing

Vengono riportati gli output dei vari test effettuati sul programma:

- Test 1 (test basico)

```
Inserisci il numero di automobili totale:
10
Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:
3

Il tempo di attesa dell'auto Automobile_1 con direzione SUD_NORD è di 0
Il tempo di attesa dell'auto Automobile_2 con direzione SUD_NORD è di 0
Il tempo di attesa dell'auto Automobile_8 con direzione NORD_SUD è di 100
Il tempo di attesa dell'auto Automobile_4 con direzione NORD_SUD è di 100
Il tempo di attesa dell'auto Automobile_5 con direzione SUD_NORD è di 200
Il tempo di attesa dell'auto Automobile_3 con direzione SUD_NORD è di 200
Il tempo di attesa dell'auto Automobile_9 con direzione SUD_NORD è di 199
Il tempo di attesa dell'auto Automobile_6 con direzione SUD_NORD è di 298
Il tempo di attesa dell'auto Automobile_0 con direzione SUD_NORD è di 298
Il tempo di attesa dell'auto Automobile_7 con direzione NORD_SUD è di 399
Tempo di attesa medio per le auto con direzione NORD_SUD 199
Tempo di attesa medio per le auto con direzione SUD_NORD 170
```

Note: Come si può notare dal test le auto: Automobile_5, Automobile_3,Automobile_9 hanno la pressappoco la stessa tempistica di attesa, queste infatti attraverseranno insieme il ponte che è in gradi di accettare 3 automobili contemporaneamente.

- Test 2 (test basico)

```
Inserisci il numero di automobili totale:
7
Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:
3
Il tempo di attesa dell'auto Automobile_5 con direzione SUD_NORD è di 0
Il tempo di attesa dell'auto Automobile_1 con direzione SUD_NORD è di 0
```

Il tempo di attesa dell'auto Automobile_0 con direzione NORD_SUD è di 99

Il tempo di attesa dell'auto Automobile_4 con direzione SUD_NORD è di 200

Il tempo di attesa dell'auto Automobile_2 con direzione NORD_SUD è di 296

Il tempo di attesa dell'auto Automobile_6 con direzione SUD_NORD è di 397

Il tempo di attesa dell'auto Automobile_3 con direzione SUD_NORD è di 397

Tempo di attesa medio per le auto con direzione NORD_SUD 197

Tempo di attesa medio per le auto con direzione SUD_NORD 198

La prima auto a far ingresso sul ponte, trovandolo subito libro avrà sempre tempo di attesa 0.

- Test 3 (test basico: 1 sola automobile)

Inserisci il numero di automobili totale:

1

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

1

Il tempo di attesa dell'auto Automobile_0 con direzione SUD_NORD è di 0

Tempo di attesa medio per le auto con direzione NORD_SUD 0

Tempo di attesa medio per le auto con direzione SUD_NORD 0

- Test 4 (test con inserimento parametri errati)

Inserisci il numero di automobili totale:

a

Proseguo con dati di DEFAULT (numero auto=10)

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

b

Proseguo con dati di DEFAULT (numero massimo auto =3)

Il tempo di attesa dell'auto Automobile_4 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_5 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_9 con direzione NORD_SUD è di 101

Il tempo di attesa dell'auto Automobile_8 con direzione NORD_SUD è di 101

Il tempo di attesa dell'auto Automobile_6 con direzione NORD_SUD è di 100

Il tempo di attesa dell'auto Automobile_2 con direzione SUD_NORD è di 198

Il tempo di attesa dell'auto Automobile_0 con direzione NORD_SUD è di 300

Il tempo di attesa dell'auto Automobile_7 con direzione NORD_SUD è di 300

Il tempo di attesa dell'auto Automobile_1 con direzione SUD_NORD è di 401

Il tempo di attesa dell'auto Automobile_3 con direzione NORD_SUD è di 501

Tempo di attesa medio per le auto con direzione NORD_SUD 233

Tempo di attesa medio per le auto con direzione SUD_NORD 149

L'inserimento dei parametri errati produce la continuazione del programma con parametri di default numero auto= 10 e numero massimo auto = 3.

- Test 5 (test con inserimento parametri errati)

Inserisci il numero di automobili totale:

5

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

a

Proseguo con dati di DEFAULT (numero massimo auto =3)

Il tempo di attesa dell'auto Automobile_0 con direzione NORD_SUD è di 0

Il tempo di attesa dell'auto Automobile_1 con direzione NORD_SUD è di 0

Il tempo di attesa dell'auto Automobile_3 con direzione SUD_NORD è di 101

Il tempo di attesa dell'auto Automobile_4 con direzione SUD_NORD è di 100

Il tempo di attesa dell'auto Automobile_2 con direzione NORD_SUD è di 195

Tempo di attesa medio per le auto con direzione NORD_SUD 65

Tempo di attesa medio per le auto con direzione SUD_NORD 100

L'inserimento di parametri errati produce la continuazione del programma con elementi di default anche nel caso in cui solo uno dei due sia errato.

- Test 6 (nessuna macchina in attesa)

Inserisci il numero di automobili totale:

0

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

1

Non sono presenti macchine in attesa

- Test 7 (simulazione ponte guasto)

Inserisci il numero di automobili totale:

2

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

0

Sul ponte non possono passare macchine in questo momento!

- Test 8 (test ponte pieno)

Inserisci il numero di automobili totale:

3

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

3

Il tempo di attesa dell'auto Automobile_0 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_1 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_2 con direzione SUD_NORD è di 0

Tempo di attesa medio per le auto con direzione NORD_SUD 0

Tempo di attesa medio per le auto con direzione SUD_NORD 0

Dal test si nota che il ponte può contenere al massimo 3 auto e che se queste vengono tutta dalla stessa direzione (in questo caso SUD_NORD) il tempo di attesa, così come le medie generate è sempre uguale a 0.

- Test 9 (test ponte posto singolo)

Inserisci il numero di automobili totale:

5

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

1

Il tempo di attesa dell'auto Automobile_0 con direzione NORD_SUD è di 0

Il tempo di attesa dell'auto Automobile_4 con direzione SUD_NORD è di 100

Il tempo di attesa dell'auto Automobile_2 con direzione SUD_NORD è di 198

Il tempo di attesa dell'auto Automobile_3 con direzione NORD_SUD è di 299

Il tempo di attesa dell'auto Automobile_1 con direzione NORD_SUD è di 398

Tempo di attesa medio per le auto con direzione NORD_SUD 232

Tempo di attesa medio per le auto con direzione SUD_NORD 149

Essendo il ponte percorribile da una sola auto, l'automobile X dovrà attendere il passaggio di tutte le automobili X-1 e poiché solo la prima automobile ha tempo di attesa 0 il tempo di attesa dell'automobile X sarà in questo caso calcolabile a priori e sarà pari a (X-1)*100ms a cui deve ancora essere sottratto il tempo di permanenza dell'auto X-1 sul ponte prima dell'arrivo dell'auto X (vedi la sezione inerente lo Schema a Blocchi).

- *Test 10 (stressing test)*

Effettuato test con 25 auto in attesa e 5 auto al massimo sul ponte

Inserisci il numero di automobili totale:

25

Inserisci il numero di automobili che possono percorrere il ponte contemporaneamente:

5

Il tempo di attesa dell'auto Automobile_1 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_3 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_13 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_16 con direzione SUD_NORD è di 0

Il tempo di attesa dell'auto Automobile_18 con direzione NORD_SUD è di 100

Il tempo di attesa dell'auto Automobile_7 con direzione NORD_SUD è di 100

Il tempo di attesa dell'auto Automobile_10 con direzione SUD_NORD è di 200

Il tempo di attesa dell'auto Automobile_23 con direzione SUD_NORD è di 200

Il tempo di attesa dell'auto Automobile_2 con direzione NORD_SUD è di 300

Il tempo di attesa dell'auto Automobile_12 con direzione NORD_SUD è di 299

Il tempo di attesa dell'auto Automobile_19 con direzione SUD_NORD è di 399

Il tempo di attesa dell'auto Automobile_14 con direzione SUD_NORD è di 398

Il tempo di attesa dell'auto Automobile_17 con direzione NORD_SUD è di 498

Il tempo di attesa dell'auto Automobile_22 con direzione NORD_SUD è di 499

Il tempo di attesa dell'auto Automobile_21 con direzione SUD_NORD è di 599

Il tempo di attesa dell'auto Automobile_11 con direzione SUD_NORD è di 599

Il tempo di attesa dell'auto Automobile_0 con direzione SUD_NORD è di 599

Il tempo di attesa dell'auto Automobile_6 con direzione NORD_SUD è di 699

Il tempo di attesa dell'auto Automobile_4 con direzione NORD_SUD è di 699

Il tempo di attesa dell'auto Automobile_5 con direzione NORD_SUD è di 698

Il tempo di attesa dell'auto Automobile_20 con direzione NORD_SUD è di 698

Il tempo di attesa dell'auto Automobile_9 con direzione NORD_SUD è di 698

Il tempo di attesa dell'auto Automobile_8 con direzione SUD_NORD è di 798

Il tempo di attesa dell'auto Automobile_24 con direzione SUD_NORD è di 799

Il tempo di attesa dell'auto Automobile_15 con direzione NORD_SUD è di 898

Tempo di attesa medio per le auto con direzione NORD_SUD 515

Tempo di attesa medio per le auto con direzione SUD_NORD 353