

# Railway Simulation

Moreno Ambrosin  
mat. 1035635

Progetto Sistemi Concorrenti e Distribuiti

Corso di laurea magistrale in Informatica  
Università degli Studi di Padova  
Padova

17 aprile 2013

# Indice

<b>1</b>	<b>Il problema</b>	<b>3</b>
1.1	Specifica arricchita . . . . .	4
<b>2</b>	<b>Analisi del problema</b>	<b>5</b>
2.1	Distribuzione . . . . .	5
2.1.1	Gestione del Tempo . . . . .	6
2.1.2	Acquisto di un Biglietto . . . . .	7
2.1.3	Terminazione del Sistema . . . . .	7
2.1.4	Avvio del Sistema . . . . .	7
2.2	Concorrenza . . . . .	8
2.2.1	Ingresso in un Segmento da parte di un Treno . . . . .	8
2.2.2	Uscita da un Segmento e accesso alla Stazione . . . . .	9
2.2.3	Acquisto di un Biglietto da parte di un Viaggiatore . . . . .	10
<b>3</b>	<b>Soluzione</b>	<b>11</b>
3.1	Logica di Distribuzione . . . . .	11
3.1.1	Regioni . . . . .	11
3.1.2	Biglietterie . . . . .	14
3.1.3	Controller Centrale . . . . .	15
3.2	Logica di Concorrenza . . . . .	16
3.2.1	Segmento . . . . .	16
3.2.2	Viaggiatore . . . . .	16
3.2.3	Treno . . . . .	17
3.2.4	Stazione . . . . .	18
3.3	Interazione tra le Entità . . . . .	19
3.3.1	Percorrenza di un Viaggiatore . . . . .	19
3.3.2	Percorrenza di un Treno . . . . .	23
3.3.3	Creazione di un Biglietto . . . . .	35
3.3.4	Terminazione distribuita . . . . .	39

---

<b>4</b>	<b>Tecnologie Adottate</b>	<b>40</b>
4.1	Specifica di alto livello . . . . .	43

# Capitolo 1

## Il problema

Il progetto didattico prevede la progettazione e la realizzazione di un simulatore software di un sistema ferroviario. Tale sistema prevede i seguenti requisiti di base (la specifica originale è consultabile all'indirizzo <http://www.math.unipd.it/~tullio/SCD/2005/Progetto.html>):

- La presenza di Treni appartenenti a categorie a priorità e modalità di fruizione diverse.
- La presenza di Viaggiatori, che eseguono operazioni elementari come acquisto di un biglietto, salita/discesa su/da un Treno, ecc.
- La definizione di un ambiente costituito da un insieme di Stazioni, ciascuna composta da:
  - Piattaforme di attesa per Treni e Viaggiatori.
  - Un Pannello Informativo che visualizza informazioni sui Treni in arrivo, in transito e che hanno appena superato la Stazione corrente.
  - Una Biglietteria, presso la quale un Viaggiatore può acquistare un biglietto di viaggio.
- La presenza di Segmenti di collegamento tra Stazioni, a percorrenza bidirezionale.
- La presenza di un entità di controllo globale che mantiene lo stato di ciascun Viaggiatore e ciascun Treno in transito.
- L'obbligo da parte di un Viaggiatore di acquistare un Biglietto prima di poter usufruire del servizio ferroviario.

- La presenza di collegamenti multipli tra Stazioni, ovvero ciascuna Stazione può essere raggiunta da più Segmenti e da ciascuna stazione possono partire più segmenti.
- Il percorso portato a termine da un Viaggiatore può comprendere cambi di treno.

## 1.1 Specifica arricchita

La soluzione presentata con il seguente documento, è relativa alla seguente raffinamento della specifica originaria:

- Ciascun Treno appartiene ad una delle seguenti due categorie:

### **Regionale**

Treno a bassa priorità, senza posto garantito.

### **FB**

Treno a priorità più alta, che necessita prenotazione.

- Ciascun Treno possiede una capienza massima di Viaggiatori.

# Capitolo 2

## Analisi del problema

La progettazione di un simulatore di un sistema ferroviario presenta diverse problematiche relativamente alla concorrenza e alla distribuzione, in quanto:

- il problema prevede l'interazione tra una popolazione di entità in maniera concorrente;
- vi sono dei punti di sincronizzazione tra le entità, che devono essere identificati e modellati opportunamente;
- non è ragionevole fare assunzioni a priori sulle tecnologie che risolveranno il problema, né sull'ambiente di esecuzione;
- è ragionevole prevedere un certo livello di distribuzione delle componenti del sistema;
- si possono presentare difficoltà dovute ai ritardi nella trasmissione di rete tra le componenti.

Di seguito andrò ad analizzare quelle che sono le principali.

### 2.1 Distribuzione

Le problematiche legate alla distribuzione sono molteplici. Nel progetto di un simulatore di un sistema ferroviario infatti, la presenza di entità distribuite è auspicabile, sia per suddividere logicamente le entità, sia per distribuire l'onere di calcolo su nodi differenti. Le caratteristiche desiderabili da un sistema distribuito che simula una struttura ferroviaria sono:

- Il complesso deve apparire all'utente come un sistema unitario, la natura distribuita del sistema deve essere nascosta all'utilizzatore finale.

- L'architettura distribuita non deve limitare le funzionalità desiderate.
- È desiderabile che vi sia un buon grado di disaccoppiamento tra le componenti, e dalle tecnologie adottate per la comunicazione tra nodi della rete.
- Il sistema dovrà essere il più possibile robusto agli errori.
- La progettazione architetturale deve prevedere un meccanismo che permetta Avvio del sistema e Terminazione ordinata.
- L'architettura distribuita deve sottostare a vincoli temporali propri di un sistema ferroviario.
- La struttura distribuita deve essere tale da permettere estendibilità e scalabilità.

### 2.1.1 Gestione del Tempo

La simulazione è scandita da orari di partenza e di arrivo dei Treni che circolano tra le stazioni. Per questo è importante dotare il sistema di un *riferimento temporale* adeguato, che permetta di gestire i ritardi introdotti dalla comunicazione di rete, o dalla diversità di sincronizzazione degli orologi dei diversi nodi della rete.

Tale problema assume forme diverse in base al grado di distribuzione scelto per le componenti che generano gli eventi caratterizzanti la simulazione. In particolare, un livello di distribuzione alto, che prevede ad esempio la collocazione di una entità Stazione per nodo della rete, richiederà un meccanismo di regolazione del tempo più complesso e delicato rispetto ad un sistema con un livello di distribuzione più contenuto, che preveda ad esempio una distribuzione di singole Regioni di simulazione.

La scelta del riferimento temporale diviene quindi cruciale per lo svolgersi della simulazione. Abbiamo due tipi possibili di orologio:

- Assoluto: Prevede l'esistenza di un'entità dalla quale le varie componenti attingono per ottenere l'informazione temporale.
- Relativo: Ciascun nodo di calcolo possiede un proprio riferimento temporale interno.

È chiaro che la prima soluzione non si presta ad essere utilizzata per il problema presentato. Infatti esso, possedendo un flusso continuo interno del tempo, non permetterebbe a entità indipendenti su nodi diversi di eseguire logicamente allo stesso istante (ad esempio due treni che in nodi diversi partono contemporaneamente da una stazione)

### 2.1.2 Acquisto di un Biglietto

In base al grado di distribuzione della modellazione realizzata, è necessario prevedere una struttura distribuita di biglietterie, in quanto la natura del problema prevede un livello di conoscenza globale soprattutto per l'acquisto di biglietti di treni a prenotazione.

### 2.1.3 Terminazione del Sistema

La durata di una simulazione di un sistema ferroviario è per sua natura indefinita. È quindi necessario un intervento esterno che ne decreti la terminazione. La *Terminazione del sistema* globale deve essere coordinata tra tutte le componenti distribuite, ed effettuata in modo tale da non far terminare l'esecuzione in uno stato inconsistente. Dovrà inoltre garantire che nessun nodo di calcolo rimarrà attivo (ad esempio, nessun thread in esecuzione o in attesa).

### 2.1.4 Avvio del Sistema

L'*Avvio del sistema* deve essere progettato in modo tale da permettere a tutte le componenti distribuite di interagire, ed evitare errori. In particolare:

- dev'essere previsto un meccanismo che permetta una rapida individuazione dei nodi con i quali ciascuna entità coopera;
- devono essere evitati (o gestiti) errori causati dal tentativo di comunicazione di thread concorrenti con entità non ancora pronte o allocate.



### 2.2 Concorrenza

Nell'analizzare le problematiche di concorrenza, è necessario individuare quali saranno le entità che svolgeranno un ruolo attivo all'interno della simulazione, e quali un ruolo reattivo, in base alle azioni compiute dalle entità attive. Nella simulazione di un sistema ferroviario, ho individuato le seguenti entità attive:

- Treno
- Viaggiatore

mentre le entità reattive principali saranno:

- Segmento
- Piattaforma

#### 2.2.1 Ingresso in un Segmento da parte di un Treno

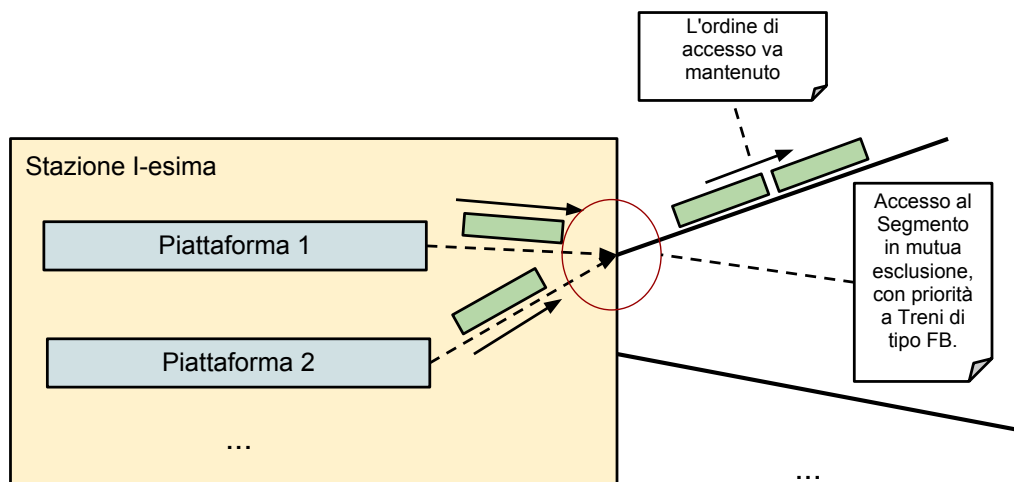


Figura 2.1: Accesso ad un Segmento da Parte di uno o più Treni.

L'accesso ad un segmento di collegamento tra due stazioni da parte di un Treno, è inerentemente concorrente. Tale azione presenta infatti i seguenti requisiti:

- L'ingresso presso un Segmento deve avvenire in *mutua esclusione*; è infatti impossibile che due o più entità Treno accedano ad uno stesso Segmento contemporaneamente.

- Più Treni possono circolare su un Segmento contemporaneamente. Questo comporta:
  - il mantenimento di un ordine di ingresso al Segmento;
  - la regolazione della velocità di transito di ciascun Treno in base alla velocità di quelli che lo precedono;
  - l'impossibilità di un Treno di accedere ad un Segmento qualora vi siano altri Treni che lo percorrono in senso opposto.
- Dev'essere data precedenza d'accesso al Segmento, ai Treni di tipo FB.

### 2.2.2 Uscita da un Segmento e accesso alla Stazione

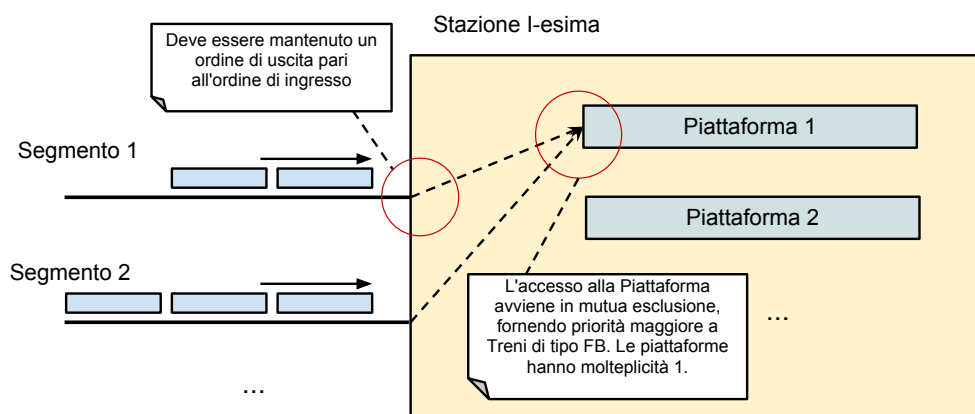


Figura 2.2: Uscita da un Segmento e accesso ad una Piattaforma di uno o più Treni.

Il problema introdotto in sezione 2.2.1, vincola i requisiti che dovranno essere soddisfatti relativamente all'uscita da un Segmento e conseguente accesso alla Stazione successiva. Per quanto riguarda l'uscita dal Segmento avremo quindi i seguenti requisiti:

- L'ordine di ingresso al Segmento dev'essere mantenuto all'uscita.
- L'ordine di uscita da un Segmento dev'essere mantenuto nell'accesso alla stazione successiva da parte dei Treni in transito.

L'ingresso in una Stazione, permette ad un Treno di occupare una delle Piattaforme disponibili e, dato che da specifica una stazione può essere raggiunta da più Segmenti, tale azione sarà svolta in modo concorrente tra i treni in uscita dai vari Segmenti, secondo i seguenti vincoli:

- I Treni di tipo FB avranno priorità maggiore nell'occupare una Piattaforma.
- Ciascuna Piattaforma è acceduta in mutua esclusione, e può essere occupata da un solo Treno alla volta.

Questo significa che l'accesso ad una Piattaforma sarà ordinato, per tutti i Treni provenienti dallo stesso Segmento, in base all'ordine di uscita da quest'ultimo, e concorrente tra Treni provenienti da Segmenti diversi.

### 2.2.3 Acquisto di un Biglietto da parte di un Viaggiatore

Ciascun Viaggiatore deve acquistare un biglietto prima di poter usufruire dei servizi ferroviari. Per fare ciò, all'interno ogni stazione vi è una Biglietteria presso la quale l'acquisto può essere effettuato. Un biglietto sarà composto da una serie di Tappe, ciascuna relativa ad un tratto del percorso da portare a termine con uno specifico Treno, sia Regionale che FB. L'assegnazione di un Biglietto ad un Viaggiatore è semplice se il suo percorso prevede solo l'utilizzo di Treni Regionli, mentre è più complesso se vi sono tappe da raggiungere con Treni FB, i cui biglietti vengono erogati se e soltanto se vi è ancora posto all'interno del treno. È quindi necessario prevedere l'esistenza di una *biglietteria centrale* che mantenga le prenotazioni dei Treni di tipo FB. Si ricavano quindi i seguenti requisiti:

- La biglietteria centrale va acceduta in mutua esclusione, in modo da evitare prenotazioni inconsistenti di Biglietti.
- L'erogazione di biglietti può fallire in caso non vi siano posti per percorrere alcune tappe; il sistema deve reagire di conseguenza.

# Capitolo 3

## Soluzione

Di seguito, verrà presentata la soluzione realizzata, in termini di progettazione come sistema distribuito e concorrente.

### 3.1 Logica di Distribuzione

Un diagramma informale delle componenti distribuite che compongono il sistema è presentato in figura 3.1. Di seguito verranno descritte le principali.

#### 3.1.1 Regioni

La simulazione è stata suddivisa in *Regioni*, le quali risiederanno su nodi di calcolo diversi. Questa scelta aggiunge i seguenti requisiti minimi alla specifica iniziale:

- I Treni, se previsto dal percorso, possono viaggiare da una Regione all'altra.
- I Passeggeri possono raggiungere destinazioni in Regioni diverse.
- Esisteranno punti di collegamento che permettono a Treni e Passeggeri di raggiungere Regioni diverse.
- Deve essere garantita consistenza temporale nel passaggio da una Regione ad un'altra.

Da questa scelta consegue inoltre l'introduzione di un semplice *Server dei Nomi* che mantiene traccia di ciascuna Regione (mantiene cioè le coppie `<Regione, indirizzo>`), in modo tale da rendere agevole la risoluzione della locazione alla quale l'entità si trova.

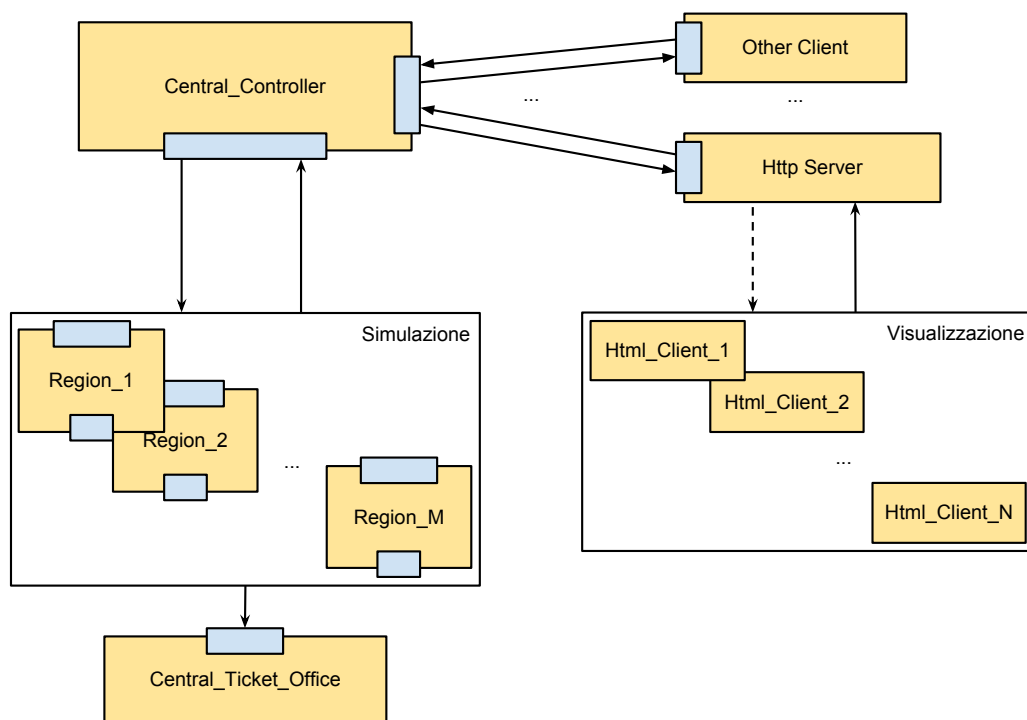


Figura 3.1: Diagramma informale che presenta una progettazione logica delle componenti distribuite.

### 3.1.1.1 Stazioni di Gateway

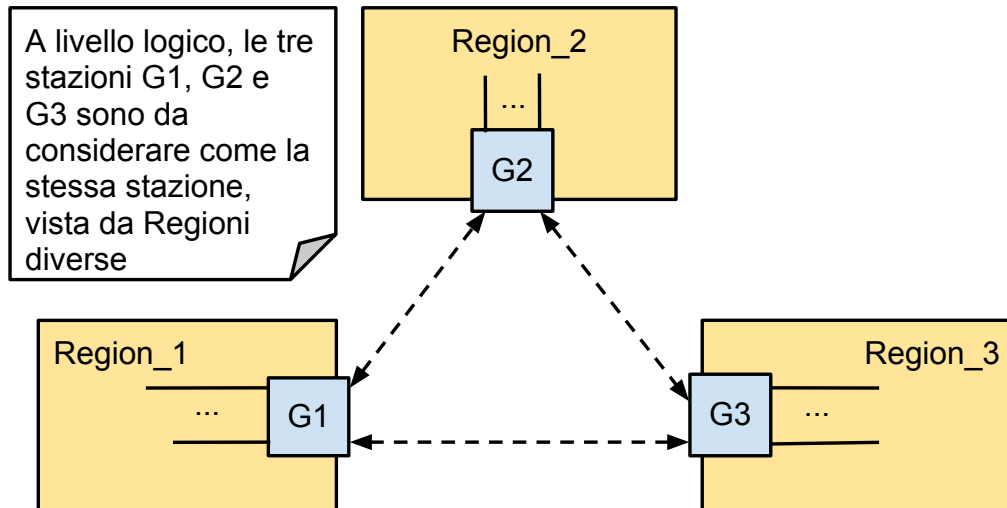


Figura 3.2: Utilizzo di stazioni di Gateway per collegare tre Regioni.

Nella progettazione di un simulatore distribuito su più Regioni, è necessario prevedere uno o più punti di collegamento tra di esse. A tal proposito, il progetto prevede *Stazioni di Gateway* (*Gateway\_Station*), che permettono di raggiungere determinate Regioni remote. Le Stazioni di Gateway che collegano  $N$  Regioni in modo diretto, sono da considerare come un'unica Stazione logica, e ciascuna delle  $N$  Stazioni può essere vista come una porzione della Stazione globale accessibile dalla Regione corrente. Una rappresentazione dell'idea generale è visibile in figura 3.2.

Da tale assunzione, ricaviamo la seguente conseguenza:

**Conseguenza 1** *Per ogni coppia  $G$  e  $G'$  di Stazioni di Gateway connesse (ovvero mutuamente raggiungibili in modo diretto) appartenenti a Regioni diverse rispettivamente  $R_G$  ed  $R_{G'}$ ,  $\nexists$  altra Stazione di Gateway  $G'' \in R_{G'}$  tale per cui  $G''$  è connesso a  $G$ .*

Per evitare sincronizzazioni tra thread distribuiti, sono posti dei vincoli nella struttura interna di questo tipo di stazioni. Ciascuna Stazione di Gateway  $G$  sarà dotata di un numero  $M$  di Piattaforme  $\{P_1, \dots, P_M\}$ , le quali permettono un *unico senso di percorrenza*, ovvero:

- Le prime  $\{P_1, \dots, P_K\}$  saranno disponibili all'accesso dalla Regione in cui  $G$  risiede.

- Le Piattaforme  $\{P_K + 1, \dots, P_M\}$  saranno invece dedicate all'accoglimento di Treni provenienti da nodi diversi.

Questa scelta, vincola così il numero di Piattaforme per Stazione di Gateway:

**Conseguenza 2** *Date  $N$  Stazioni di Gateway connesse, sia  $K_i$  il numero di Piattaforme che permettono ad un Treno di lasciare la Regione  $i$ -esima; allora ciascuna Stazione dovrà complessivamente mantenere un numero di Piattaforme pari a  $\sum_{i=1}^N K_i$ .*

Tali vincoli sono sufficienti a mantenere locale ai nodi la sincronizzazione tra i thread che permettono l'esecuzione dei Treni. Una descrizione dettagliata del protocollo di accesso alle Stazioni di Gateway e di migrazione dei Treni, è presentata in sezione 3.3.2.3. Per una descrizione dell'entità e del ruolo delle Piattaforme, si rimanda alla sezione 3.2.4

#### 3.1.2 Biglietterie

Per poter gestire meglio la definizione di un percorso e l'erogazione di un Biglietto per un Viaggiatore, ho pensato di introdurre una gerarchia su due livelli, di Biglietterie. Ci saranno dunque tre categorie di Biglietterie:

##### **Biglietterie di Stazione**

Forniscono un'interfaccia adeguata ai Viaggiatori per poter acquistare un biglietto.

##### **Biglietterie Regionali**

Hanno conoscenza regionale della topologia del grafo composto da Stazioni e Segmenti.

##### **Biglietteria Cantrale**

Ha conoscenza di più alto livello; in particolare, essa mantiene traccia delle connessioni tra le varie regioni ( ovvero i collegamenti tra Stazioni di Gateway di regioni diverse). Tale conoscenza si può immaginare sia memorizzata in una mappa chiave-valore **Links** dove per ciascuna coppia di Regioni, è mantenuta una lista di tappe successive Regione-Stazione di Gateway da percorrere per andare dalla prima alla seconda.

### 3.1.3 Controller Centrale

Il Conterollo Centrale è una entità distribuita, alla quale tutti i nodi inviano Eventi per notificare lo stato di avanzamento globale della simulazione. Esso fornisce una interfaccia alle varie Regioni per ricevere gli Eventi di simulazione, ed un'interfaccia per permettere a client remoti di poter visualizzare gli effetti di tali Eventi. Quest'ultima possibilità è ottenuta mediante un meccanismo di tipo Publish/Subscribe, attraverso il quale client remoti possono registrarsi presso il Controller per ricevere, in modalità Push, gli Eventi. In questo modo è possibile per un qualsiasi client interfacciarsi al Controller e fornire, ad esempio, una rappresentazione grafica della simulazione.

Il Controller sarà anche responsabile dell'invio di un *segnale di terminazione*, che verrà recapitato a tutti i nodi che compongono la simulazione, mediante un algoritmo di Snapshot Distribuito, descritto in sezione 3.3.4.



## 3.2 Logica di Concorrenza

Internamente a ciascuna Regione di simulazione, sono definite entità che modellano le interazioni previste dalla specifica del problema. Di seguito sono descritte le principali.

### 3.2.1 Segmento

Un Segmento (**Segment**) è modellato come una *entità reattiva ad accesso mutuamente esclusivo, a molteplicità  $N$* , dove  $N$  è il numero massimo di utilizzatori, la quale fornisce un'interfaccia utilizzabile da entità attive di tipo Treno per accedere ed uscire in maniera ordinata. Esso è caratterizzato da:

- le due stazioni che esso collega;
- una lunghezza;
- una velocità massima di percorrenza.
- un flag booleano **Free** che indica lo stato della risorsa, occupata o no.

### 3.2.2 Viaggiatore

In prima analisi, un Viaggiatore (**Traveler**) può essere modellato come una *entità Attiva* che esegue una sequenza di operazioni semplici. In presenza di un modello di distribuzione come quello presentato in sezione 3.1.1 però, si presenta il problema di modellare il passaggio da una Regione (nodo) ad un'altra del Viaggiatore, nel caso in cui il viaggio da esso percorso lo preveda.

Nel caso in cui il Viaggiatore fosse messo in relazione univoca con un thread, si presenterebbero solo due possibili soluzioni:

- La migrazione del processo che rappresenta il Viaggiatore sul nodo (Regioni) di destinazione, come distruzione del processo sul nodo di partenza e creazione dinamica dello stesso sul nodo destinazione. Questa operazione è in generale computazionalmente molto costosa.
- La replicazione del thread che rappresenta il Viaggiatore su tutti i nodi, e l'attivazione, intesa come cambio di stato del processo in modo tale che possa competere per la CPU; tale soluzione è molto costosa in termini di memoria utilizzata, e non scalabile all'aumentare del numero di passeggeri.

La soluzione che ho adottato, consiste nel disaccoppiare le operazioni svolte da ciascun Viaggiatore dal processo che le esegue, prevedendo una struttura dati (**Traveler.Pool**) costituita da:

- un *pool di M thread* dimensionato in maniera opportuna;
- una *coda di operazioni* che man mano vengono estratte ed eseguite dai thread nel pool.

In questo modo è sufficiente replicare per ciascun Viaggiatore, su tutti i nodi che compongono il sistema, una struttura dati che contiene i suoi dati e le operazioni che esso eseguirà, e cioè un Descrittore (**Traveler.Descriptor**). Il cambio di Regione di un Viaggiatore potrà quindi essere ottenuto semplicemente inserendo la prossima operazione da eseguire per il Viaggiatore nella coda del pool di thread del nodo destinazione.

#### 3.2.3 Treno

Un Treno (**Train**) è una *entità Attiva*, la quale esegue ciclicamente un numero finito di operazioni. Ciascuna entità Treno può appartenere a due categorie, FB a priorità più alta, e Regionale a priorità minore, ed è caratterizzata da:

- un identificativo univoco;
- una capienza massima;
- una velocità massima raggiungibile;
- una velocità corrente;

Anche in questo caso come per le entità di tipo Viaggiatore, ciascuna entità non viene mappata su un singolo processo. Infatti anche per le entità di tipo Treno ciò comporterebbe una complessa e costosa gestione del passaggio da un nodo (Regione) ad un altro, in termini computazionali e di memoria. Ho optato invece nel progettare una struttura dati **Train.Pool** che mantiene:

- una *coda di Descrittori di Treno*, che contengono tutte le informazioni che distinguono una entità Treno;
- un *pool di thread* dimensionato in maniera opportuna; ciascun processo sarà tale per cui una volta ottenuto un descrittore dalla coda, eseguirà per lui una fissata sequenza di azioni.

In questo modo un entità Treno eseguirà all'interno di un nodo se e soltanto se il suo descrittore sarà inserito nella coda di descrittori della struttura dati descritta.

### 3.2.4 Stazione

Una Stazione (**Station**) è modellata come una struttura dati contenete:

- un certo numero  $n > 2$  di Piattaforme (**Platform**);
- una Biglietteria (**Ticket\_Office**);
- un Pannello Informativo (**Notice\_Panel**).

Essa offre una interfaccia alle entità Treno e Viaggiatore per l'accesso a Piattaforme e Biglietteria.

#### 3.2.4.1 Piattaforma

Una Piattaforma è modellata come una *entità reattiva ad accesso mutuamente esclusivo, a molteplicità 1*. Essa espone un'interfaccia che permette alle entità Treno di

- poter sostare ed effettuare discesa e salita dei Viaggiatori;
- poter superare la stazione.

mentre alle entità Viaggiatore di accodarsi in attesa di uno specifico Treno. Internamente essa mantiene due code di **Traveler\_Descriptor**:

- **Leaving\_Queue**, che conterrà i descrittori dei Viaggiatori in attesa di Treni per poter effettuare la partenza.
- **Arrival\_Queue**, che conterrà i descrittori dei Viaggiatori in attesa di Treni per poter effettuare l'arrivo alla stazione.

Inoltre contiene un flag booleano **Free** che ne indica lo stato di occupazione. Ciascuna Piattaforma, in base al tipo di Stazione avrà percorrenza bidirezionale o unidirezionale.

#### 3.2.4.2 Biglietteria

Una Piattaforma è modellata come una interfaccia che permette al Viaggiatore di acquisire un Biglietto di viaggio.

#### 3.2.4.3 Pannello Informativo

Una Piattaforma è modellata come una *entità reattiva con agente di controllo, a molteplicità 1*. Esso espone una interfaccia tale da permettere alla stazione di notificare lo stato delle entità Treno che stanno arrivando, quelle in sosta e quelle in partenza.

## 3.3 Interazione tra le Entità

### 3.3.1 Percorrenza di un Viaggiatore

Alla luce della definizione dell'entità *Viaggiatore* in sezione 3.2.2, è possibile definire le azioni che compongono il viaggio:

- Acquisto di un Biglietto (*Ticket*) presso la Biglietteria della Stazione (*Ticket\_Office*) di partenza. Ogni Biglietto è composto da una sequenza ordinata di Tappe (*Ticket\_Stages*), ciascuna contenente:

- *start\_station*
- *next\_station*
- *train\_id*
- *start\_platform*
- *destination\_platform*
- *next\_region*

e da un indice della prossima tappa del Percorso (*Next\_Stage*).

- Una volta ottenuto un *Ticket*, vengono eseguite le seguenti operazioni per ciascuna Tappa del Biglietto:
  - Accodamento presso la Piattaforma *start\_platform* della Stazione *start\_station* in attesa del Treno *train\_id*.
  - All'arrivo del treno *train\_id*, Accodamento presso la Piattaforma *destination\_platform* della Stazione *next\_station* in attesa dell'arrivo di *train\_id*.

Le azioni sopraelencate possono essere incapsulate in strutture dati che rappresentano una specifica operazione. In questo modo viene generata una *gerarchia di operazioni*, tutte derivanti da un'unica *operazione generica*. Di seguito identificheremo tali operazioni rispettivamente con *BUY\_TICKET*, *LEAVE* e *ARRIVE*.

Il protocollo di operazioni che vengono eseguite da un *Viaggiatore*, è stato mantenuto il più semplice possibile, e l'intero percorso viene regolato dagli eventi generati dalle entità *Treno* alla partenza e all'arrivo dalle/nelle Stazioni.

#### 3.3.1.1 Acquisto di un Biglietto

L'acquisto di un Biglietto da parte di un *Viaggiatore* è effettuato tramite l'operazione *BUY\_TICKET*. Essa si limita a richiedere alla Biglietteria della Stazione di partenza (che viene definita da configurazione iniziale) un Biglietto per

una specifica destinazione. Al termine della richiesta, l'esecuzione dell'operazione termina, e il thread corrente ritorna nella coda di thread `Traveler.Pool` (descritta in sezione 3.2.2) per poter eseguire una nuova operazione. Questo comporta la definizione di una nuova operazione, `TICKET_READY`, la quale verrà inserita nella coda di operazioni di `Traveler.Pool`, all'avvenuta ricezione del Biglietto richiesto, sia per assegnare il `Ticket` creato e caricare nella coda della struttura dati `Traveler.Pool` l'operazione `LEAVE` del Viaggiatore corrente, sia per eseguire operazioni in caso la richiesta non fosse andata a buon fine.

#### 3.3.1.2 Partenza da una Stazione

Denominiamo  $t$  la tappa di indice `Next.Stage`, ovvero la tappa corrente. Perché l'operazione di Partenza dalla stazione corrente venga effettuata, è necessaria la preconditione per la quale:

- l'operazione `LEAVE` è stata inserita nella apposita coda di `Traveler.Pool`;
- in qualche momento essa venga prelevata (rimossa) da tale luogo ed eseguita da uno dei thread del pool di `Traveler.Pool`.

Le azioni compiute dall'operazione `LEAVE` sono:

- Estrazione di `start_station` da  $t$ ;
- Tramite l'interfaccia esposta dalla Stazione `start_station` (una procedura che identifichiamo con `Wait.For.Train.To.Go`), viene richiesto di attendere presso la Piattaforma `start_platform`.

Internamente alla Stazione, tale richiesta viene tradotta nell'inserimento del Viaggiatore, o meglio del suo Descrittore `Traveler.Descriptor`, nella coda `Leaving.Queue` della Piattaforma indicata da `start_platform`. Tale operazione non utilizza una sezione critica presso la risorsa protetta a protezione della Piattaforma, ma possiamo pensare che la coda (FIFO) adottata sia ad accesso sincronizzato.

Nel caso in cui la Stazione di partenza del Viaggiatore sia una Stazione di Gateway, allora la procedura di accodamento presso la Piattaforma designata diviene:

- Se la Regione di Destinazione `next_region` è diversa dalla Regione corrente, allora
  - I dati relativi al Viaggiatore (Descrittore e Biglietto) vengono serializzati (*marshalling*).

- Viene inviato un messaggio alla prossima Regione (ovvero `next_region`, il cui indirizzo viene recuperato dal Server dei Nomi) contenente i dati serializzati, destinato alla Stazione di Gateway corrispondente in tale Regione (nodo).
- A destinazione, vengono aggiornati i dati relativi al Viaggiatore locale alla Regione e eseguito un accodamento presso la coda `Leaving_Queue` della Piattaforma indicata da `start_platform`.

Il comportamento descritto non è proprio della operazione `LEAVE`, ma è contenuto nell'operazione della Piattaforma per l'accodamento del Viaggiatore presso la coda `Leaving_Queue`.

#### 3.3.1.3 Arrivo alla Destinazione successiva

Similmente a quanto descritto per la partenza, l'arrivo a destinazione prevede che l'operazione `ARRIVE` sia già stata inserita all'interno della coda di operazioni in `Traveler_Pool`, e che un thread del pool la esegua (rimuovendola dalla coda). A differenza della partenza, questa operazione può comportare l'accodamento del Viaggiatore presso una Stazione che non appartiene alla Regione (al nodo) corrente, anche se non di Gateway.

Vengono eseguite le seguenti azioni:

- Estrazione della prossima stazione (`next_station`) e della Regione di destinazione (`next_region`) dalla tappa corrente  $t$ . Si presentano ora due casi:
  - Se la prossima Regione è la stessa della Regione corrente, allora mediante l'interfaccia esposta dalla Stazione locale `next_station`, viene effettuata una richiesta di attesa presso la Piattaforma `destination_platform`
  - Se la destinazione è una Stazione appartenente ad un'altra Regione, allora:
    - \* Viene individuato l'indirizzo della Regione remota (richiesto al `Name_Server` che gestisce le Regioni).
    - \* Vengono serializzati (*marshalling*) i dati relativi al Viaggiatore (Descrittore del Viaggiatore) e al suo Biglietto.
    - \* Viene inviato un Messaggio Remoto alla Regione specificata che provvederà ad effettuare l'aggiornamento del corrispondente Viaggiatore, e al suo accodamento presso la Stazione corretta.

Internamente alla Stazione, la richiesta si traduce nell'inserimento del Descrittore del Viaggiatore corrente all'interno della coda **Arrival.Queue** interna alla Piattaforma **destination\_platform** selezionata.

Si noti che l'eventuale azione di trasferimento del Viaggiatore al nuovo nodo viene effettuata dall'operazione **ARRIVE** e non dalla stazione all'arrivo del Treno (descritto nella sottosezione 3.3.2.2). Questa scelta è stata fatta per limitare il più possibile l'occupazione da parte di un Treno della Piattaforma (risorsa protetta), ed evitare quindi l'esecuzione di operazioni potenzialmente lunghe come l'invio di un messaggio remoto.

#### 3.3.2 Percorrenza di un Treno

A ciascuna entità Treno, è assegnato un Percorso (**Route**) di andata e un Percorso di ritorno, ovvero sequenze di Tappe (**Stage**) successive, ciascuna composta dai seguenti campi:

- `start_station`,
- `start_platform`,
- `next_segment`,
- `next_station`,
- `next_platform`,
- `leave_action`,
- `enter_action`,
- `node_name`

dove `leave_action` e `enter_action` indicano rispettivamente quello che un Treno dovrà compiere alla partenza dalla stazione `start_station` e all'arrivo presso la prossima stazione `next_station`, a scelta tra **ENTER**, per entrare ed effettuare discesa e salita passeggeri o **PASS** per non fermarsi e oltrepassare la Stazione; il campo `node_name` invece, indica la regione di destinazione, e viene utilizzato nel caso di transito su stazione di Gateway.

Una volta che un thread del pool della struttura **Train.Pool** ottiene un Descrittore, effettua le seguenti operazioni, per ciascuna Tappa del Percorso corrente (di andata o di ritorno):

- Partenza dalla Stazione `start_station`, Piattaforma `start_platform`.
- Accesso al prossimo Segmento `next_segment`.
- Percorrenza all'interno del Segmento come attesa finita di durata proporzionale alla lunghezza del Segmento e alla velocità massima alla quale il Treno può percorrerlo.
- Uscita dal Segmento e richiesta di Accesso alla Stazione successiva (`next_station`) presso la Piattaforma indicata da `next_platform`, per eseguire l'azione `action`.
- Se `action = ENTER` allora effettua discesa e salita dei Viaggiatori in attesa dell'arrivo del Treno.
- Se ci sono ancora Tappe da percorrere nel percorso (**Route**), allora il l'indice della prossima tappa del Descrittore del Treno corrente viene incrementato, e lo stesso Descrittore viene inserito in una delle code di **Train.Pool** (in base alla priorità).



#### 3.3.2.1 Accesso ad un Segmento

L'accesso alla risorsa protetta Segmento è regolato da una interfaccia ben definita, che permette:

- Ingresso.
- Uscita.

Per mantenere l'ordine di accesso, ciascun Segmento è dotato di una Coda FIFO (**Train.Queue**) che conterrà i Descrittori dei Treni correntemente in transito. Tale coda avrà una capienza massima per limitare il numero di accessi consecutivi da una singola direzione.

##### *Ingresso*

La richiesta di accesso (**Enter**) al segmento avviene in mutua esclusione tra tutti le entità Treno. In ogni momento quindi solo una entità eseguirà all'interno della risorsa protetta. Una volta ottenuta la risorsa ciascun Treno compie le seguenti operazioni

- Inserimento del Descrittore nella coda **Train.Queue**;
- Aggiornamento della velocità di percorrenza del Treno entrato in base a quella dei treni che lo precedono.
- Nel caso in cui la risorsa risulti vuota (viene consultato il flag booleano che mantiene questa informazione), allora viene modificato il valore del flag in modo tale da indicare lo stato occupato della risorsa, e memorizzata la stazione di provenienza del Treno.

Una volta terminate queste due operazioni, il Treno rilascia la risorsa e, basandosi sulla lunghezza e sulla velocità da mantenere, simula la percorrenza rendendosi inattivo (non competitivo per l'ottenimento della CPU) per un tempo dato dalla semplice equazione:  $Time = Segment\_Length / Actual\_Speed$ .

Nel permettere percorrenza multipla del Segmento, si presentano diverse problematiche: nel caso in cui una volta ottenuta la risorsa protetta Segmento, un Treno richieda l'accesso nella direzione opposta a quella dei Treni che percorrono il Segmento (ovvero abbiamo la situazione in cui il flag booleano **Free = False**, e la stazione di provenienza memorizzata presso il Segmento è diversa da quella del Treno richiedente) allora questo dovrà attendere fino a che il Segmento non sarà si sarà

liberato dai treni in transito. Questa attesa dev'essere tale da evitare starvation del Treno in attesa.

Una primo approccio possibile è quello di prevedere una coda di attesa interna alla risorsa Segmento (**Waiting\_Queue**), per i Treni provenienti dalla direzione opposta rispetto al senso di marcia corrente: tali Treni dovranno avere priorità maggiore nell'accedere al Segmento appena esso diventa vuoto, rispetto ad altri Treni che sopraggiungono successivamente. Questo però, non risulta sufficiente a garantire che i Treni nella coda avranno accesso al Segmento in qualche momento: si consideri la presenza di un percorso circolare composto da  $N$  Segmenti, e di  $M$  Treni che viaggiano lungo tale percorso in senso orario, in modo tale che in ogni istante ci sia almeno un treno all'interno di ciascun Segmento. Se ora aggiungiamo al sistema un Treno che viaggia in direzione opposta, allora esso, adottando la semantica di accesso descritta non riuscirà mai a percorrere uno dei Segmenti.

Una possibile soluzione al problema, è mantenere nella coda **Waiting\_Queue** *tutte* le entità Treno che non possono accedere correntemente, sia perché provenienti dalla direzione opposta, sia perché il numero massimo di accessi da una direzione è stato raggiunto. In questo modo la semantica di accesso viene modificata come segue:

- Se **Free** = **True** allora il Treno corrente
  - Modifica il valore di **Free** a **False**.
  - Imposta la Stazione provenienza con la propria.
  - Incrementa di 1 il contatore di accessi.
- Se invece **Free** = **False** allora
  - Se la stazione di provenienza del Treno corrente è diversa da quella memorizzata nel Segmento, allora inserisci il Treno nella coda **Waiting\_Queue** e STOP.
  - Altrimenti
    - \* Se il contatore di accessi è  $<$  del limite massimo di Treni provenienti da una singola direzione, allora incrementa di 1 il contatore di accessi.
    - \* Altrimenti accoda il Treno su **Waiting\_Queue** e STOP.
- Se il Treno ha avuto accesso al Segmento, inserisci il Descrittore del Treno nella coda **Train\_Queue** e modifica velocità di transito del Treno a seconda della massima velocità possibile.

#### *Uscita*

L'uscita (**Leave**) da una Segmento da parte di una entità Treno, ha come prerequisito l'aver avuto accesso al Segmento. È quindi possibile assumere che il descrittore del Treno che intende uscire dal Segmento sia presente all'interno della coda **Train\_Queue**, e che inoltre al momento di tale richiesta abbia già terminato il tempo previsto di attesa che simula la percorrenza.

Il requisito principale richiesto dall'azione di uscita è che essa avvenga in maniera ordinata, in base all'ordine con cui i Treni hanno avuto accesso al Segmento. La soluzione apportata mira a garantire tale ordine di uscita, senza fare alcuna assunzione sull'ordine con il quale i Treni verranno scelti per l'esecuzione dallo scheduler. La semantica adottata è quindi la seguente:

- Viene controllato per prima cosa se il Treno corrente è effettivamente il prossimo che deve uscire secondo l'ordine di ingresso.
- Se è il prossimo (vengono confrontati i Descrittori) allora il Treno può abbandonare la risorsa protetta.
- Altrimenti il Descrittore del Treno corrente viene posto in attesa su una coda; tale coda verrà riesaminata ogni volta che un Treno abbandonerà la risorsa protetta (disponendo ad esempio di istruzioni wait e signal, ciascun Treno uscente invocherà una signal sulla coda), e se il Treno in esecuzione non sarà nuovamente il Treno destinato ad uscire, esso verrà riaccodato.

La semantica descritta, garantisce ingresso sequenziale a molteplicità limitata, e uscita ordinata secondo l'ordine di ingresso. Il passo successivo consiste nel garantire l'accesso alla Stazione con lo stesso ordine di uscita per tutti i Treni provenienti dallo stesso Segmento.

#### **3.3.2.2 Accesso ad una Stazione Regionale**

I prerequisiti alla richiesta da parte di un Treno  $T$  di accesso ad una Stazione sono:

- Il Treno  $T$  ha avuto accesso ad un Segmento  $S$  che collega la stazione corrente a quella dalla quale proviene. Ciò significa che il suo Descrittore sarà stato inserito nella coda **Train\_Queue** di  $S$ .
- Il Treno  $T$  ha simulato la percorrenza su  $S$

### 3.3. INTERAZIONE TRA LE ENTITÀ

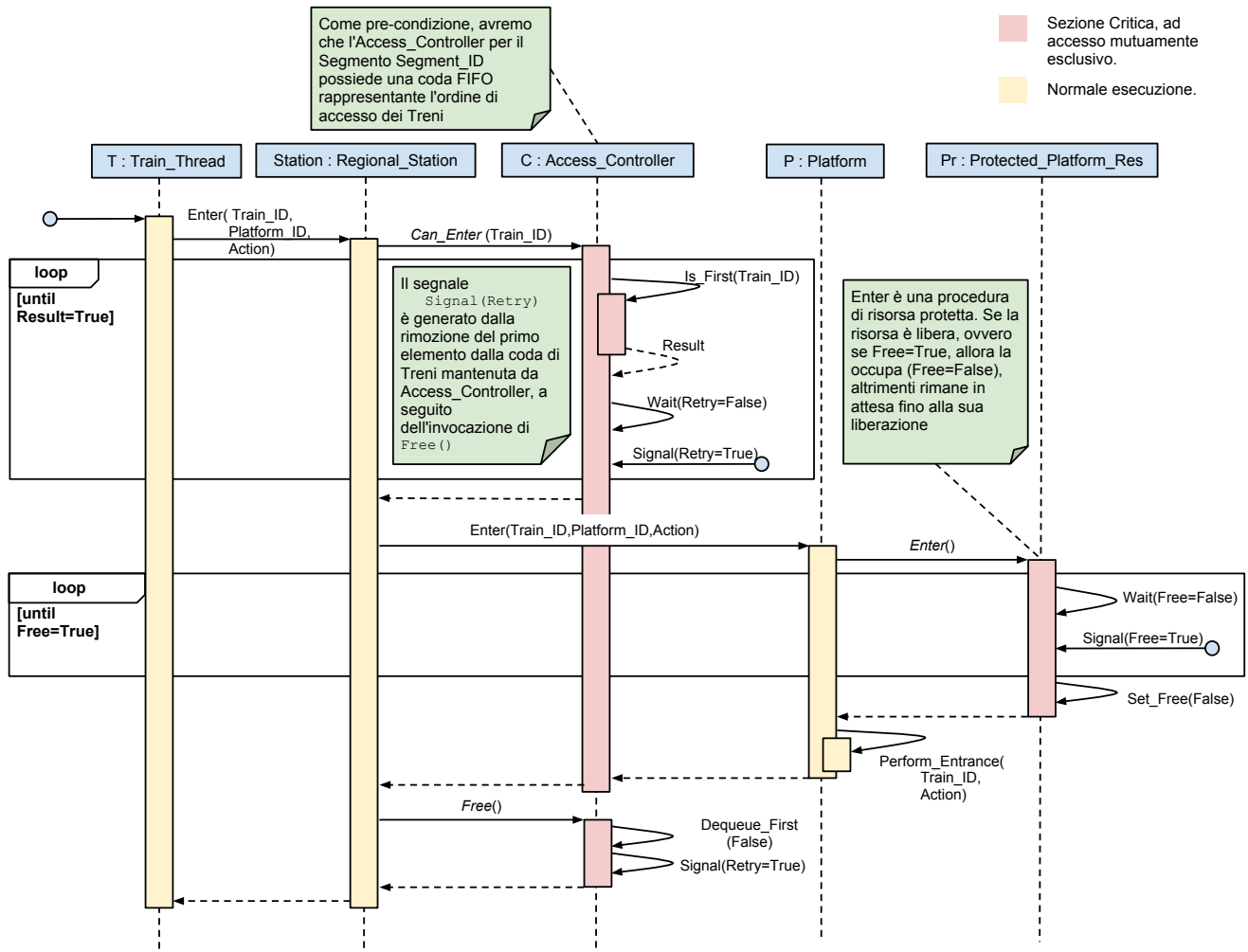


Figura 3.3: Diagramma di Sequenza, operazioni necessarie per l'ingresso ad una Piattaforma.

- Il Treno  $T$  uscito dal Segmento  $S$ , quindi è stato rimosso dalla coda `Train_Queue` di  $S$ .

Il diagramma di sequenza in figura 3.3 mostra le fasi che portano all'accesso alla Piattaforma.

#### *Ingresso Ordinato*

Per tutti i Treni provenienti dallo stesso Segmento, è necessario mantenere un ordine di richiesta di accesso alla Stazione successiva uguale a quello utilizzato per l'uscita dal Segmento. Poiché il sistema non può fare assunzioni sulle scelte dello scheduler che verrà adottato, è necessario introdurre un meccanismo che riesca a mantenere l'ordine di richiesta di accesso anche nel caso in cui, il thread che rappresenta il Treno appena uscito dal Segmento venga prerilasciato e venga eseguito il thread rappresentante il Treno successivo.

Ho vagliato le seguenti soluzioni possibili:

1. Rappresentazione della Stazione come entità passiva ad accesso mutuamente esclusivo. Essa mantiene al suo interno una coda per Segmento, del tutto simile alla coda `Train_Queue` interna ai Segmenti, e popolata parallelamente a `Train_Queue`. In questo modo è semplice garantire (con un meccanismo simile a quello utilizzato per l'uscita da un Segmento) un accesso ordinato. Questa soluzione ha però uno svantaggio evidente: la richiesta di accesso avviene in mutua esclusione anche tra i Treni provenienti da Segmenti diversi, e che quindi si ritroveranno a dover superare un punto di sincronizzazione inutile, se diretti a Piattaforme diverse della Stazione.
2. Una seconda possibilità prevede la presenza, all'interno di ciascuna Stazione, di una risorsa ad accesso mutuamente esclusivo per ciascun Segmento entrante, la quale si occuperà di garantire l'ordine di accesso (tale ordine sarà dato da una coda interna aggiornata in modo analogo alla soluzione precedentemente illustrata). Una volta che il Treno ha guadagnato il permesso di accedere in mutua esclusione alla risorsa protetta di controllo, esso avrà il permesso di accedere alla Piattaforma corretta, in modo concorrente con i Treni provenienti da altri Segmenti entranti. La coda con politica di ordinamento FIFO che indicherà l'ordine di accesso, sarà aggiornata dal Segmento mediante un'invocazione ad una procedura di interfaccia della Stazione (ad esempio `Add_Train`). In questo

modo la Stazione diventerà solo un'interfaccia utilizzabile per l'accesso alle piattaforme, e inoltre i vincoli di ordinamento e accesso concorrente alle Piattaforme da Treni provenienti da Segmenti diversi saranno garantiti.

Lo svantaggio di utilizzare questa soluzione risiede nella creazione delle entità protette, una per ciascun Segmento entrante, che regolano l'accesso ordinato, e quindi spostano parte della conoscenza della topologia della rete ferroviaria anche sulle Stazioni, rendendone più complessa la configurazione iniziale.

3. La terza soluzione esaminata, che è quella effettivamente utilizzata nel progetto, è simile alla soluzione precedente, ma a differenza di essa opera una popolazione dinamica della struttura dati che andrà a contenere le risorse protette (**Access\_Controller**) che garantiscono l'ordinamento. Ciascun Treno nell'accedere alla Stazione, includerà anche un'identificativo univoco del Segmento di provenienza, e se per esso non esisterà un controllore degli accessi, allora esso verrà creato. Questa variante ha il vantaggio per il quale le stazioni non devono necessariamente essere a conoscenza della topologia del sistema ferroviario; inoltre l'allocazione di controllori sarà limitata al massimo al numero di Segmenti in ingresso.

Il controllore di accessi (**Access\_Controller**) ha quindi il compito di fornire da semaforo internamente alla Stazione, in modo tale ogni volta da lasciar passare solamente il Treno (il thread che lo esegue) che effettivamente è primo nella coda. Siano **Enter** e **Leave** rispettivamente le due procedure protette del semaforo, allora l'accesso alla Piattaforma verrà regolato nel seguente modo:

- Il Treno *T* usa **Enter** per poter accedere alla prossima Piattaforma.
- Se *T* è effettivamente il primo della Coda allora
  - prosegue all'*accesso alla Piattaforma*;
  - libera l'**Access\_Controller** con **Free**.
- Altrimenti viene messo in attesa del proprio turno su una apposita coda (anch'essa FIFO). I Treni in attesa su questa coda verranno risvegliati ogni volta che un Treno effettuerà un accesso e una successiva chiamata a **Free**.

*Ingresso in una Piattaforma*

Dopo aver superato la risorsa di controllo **Access\_Controller**, ciascun Treno è libero di richiedere l'accesso alla Piattaforma successiva prevista dal percorso, interna a ciascuna Stazione, poiché sicuramente sarà l'unico ad eseguire a questo punto. La semantica di accodamento su di una generica Piattaforma  $P$  (supponiamo disponibile mediante procedura eseguita in mutua esclusione) è la seguente:

- Se  $P$  è libera, ovvero il flag booleano **Free** sarà **True**, allora
  - Il Treno occupa la Piattaforma (**Free** = **False**).
  - Se l'azione (**action**) prevista dal Percorso è **ENTER** allora:
    - \* esegue l'operazione di **Discesa dei Viaggiatori**;
    - \* rilascia la risorsa, in modo tale che altri Treni vi si possano accodare, e rimane in attesa fino all'orario prestabilito di partenza;
- Se invece la Piattaforma risulta occupata (**Free** = **False**), il Treno viene accodato presso una coda di Treni interna alla Piattaforma. Rappresentando la Piattaforma come una risorsa protetta da monitor, l'accodamento si traduce in una attesa su condizione, **wait(Free = True)**.

L'Operazione di **Discesa dei Viaggiatori**, si basa sulla possibilità che presso la Piattaforma corrente  $P$  vi siano Viaggiatori all'interno della coda **Arrivals\_Queue** in attesa di un evento generato da uno specifico Treno, ovvero il suo arrivo presso  $P$ . L'operazione di Discesa dei passeggeri ha come preconditione l'accesso alla Piattaforma da parte di un Treno  $T$  descritto in precedenza, ed è realizzata dalle seguenti azioni (eseguite dal thread associato a  $T$  in mutua esclusione all'interno della risorsa protetta  $P$ ):

- Viene estratto ciascun Viaggiatore  $V$  dalla coda **Arrival\_Queue**, e per ciascuno di essi:
  - Se il Treno atteso da  $V$  (informazione recuperabile dalla Tappa corrente del suo Biglietto) è proprio  $T$  allora
    - \* il numero di posti occupati di  $T$  viene decrementato;
    - \* se la Stazione corrente  $S$  **non** è la destinazione che  $V$  deve raggiungere allora:
      - l'Operazione **LEAVE** del Viaggiatore viene inserita nella coda di operazioni di **Traveler\_Pool**;
      - viene incrementato l'indice **Next\_Stage** nel Biglietto del viaggiatore  $V$ .

- Se  $T$  invece non è il Treno atteso da  $V$ , quest'ultimo viene reinserito nella coda `Arrivals_Queue`.

Le operazioni necessarie ad attuare la Discesa dei Viaggiatori, possono essere eseguite sia internamente che esternamente alla risorsa a protezione della Piattaforma. Le due soluzioni sono logicamente equivalenti, tuttavia ho preferito mantenere l'esecuzione del codice per la discesa esterno, per limitare il compito svolto dalla Risorsa protetta a Semplice semaforo per garantire l'accesso mutuamente esclusivo.

#### *Uscita da una Piattaforma*

L'uscita da una Piattaforma  $P$  necessita di due prerequisiti principali:

- Il thread che gestisce il Treno ha prima effettuato l'ingresso, e quindi `Free = False` presso la risorsa a protezione della Piattaforma, ed è già stata effettuata la Discesa dei Passeggeri (se prevista dal Percorso del Treno).
- È stato generato l'evento che notifica l'arrivo dell'orario previsto per la partenza.

A questo punto, le operazioni effettuate (eseguite in mutua esclusione dal thread corrente) sono:

- Viene effettuata la Salita dei Viaggiatori in attesa presso  $P$ .
- La risorsa viene rilasciata dal Treno in esecuzione, cambiando il valore del flag `Free` a `True`.

La **Salita dei Viaggiatori** è simile alla discesa dei Passeggeri presentata al punto precedente. Essa viene eseguita in mutua esclusione, ed è composta dalle seguenti azioni:

- Viene estratto ciascun Viaggiatore  $V$  dalla coda `Leaving_Queue` di  $P$ , e per ciascuno di essi:
  - Se il Treno atteso da  $V$  (informazione recuperabile dalla Tappa corrente del suo Biglietto) è proprio il Treno correntemente in esecuzione  $T$  e *se la capienza massima di  $T$  non è stata raggiunta* allora:
    - \* il numero di posti occupati di  $T$  viene incrementato;
    - \* l'Operazione `ENTER` del Viaggiatore viene inserita:



- Se  $T$  invece non è il Treno atteso da  $V$ , quest'ultimo viene reinserito nella coda `Leaving Queue`.

Anche in questo caso, le operazioni che attuano la discesa dei passeggeri possono essere eseguite internamente alla sezione critica definita dalla risorsa a protezione della Piattaforma, o esternamente, per poi permettere l'acquisizione da parte di un altro Treno liberandola (`Free = True`). Anche in questo caso ho preferito la seconda soluzione.

#### 3.3.2.3 Accesso ad una Stazione di Gateway

Alcuni Treni seguono percorsi che attraversano più Regioni. Tra le tappe che compongono tali percorsi, alcune indicheranno l'attraversamento di Regioni di Gateway, introdotte nella sezione 3.1.1.1. Le azioni di Ingresso, Salita dei Viaggiatori, e Ripartenza presso questo tipo di stazioni seguono le stesse regole descritte per le stazioni Regionali nella sezione 3.3.2.2. Esse hanno quindi un effetto locale alle Regioni (nodi) di provenienza e di destinazione. Per quanto riguarda invece la Discesa dei Viaggiatori, essa può prevedere un *trasferimento remoto* dei Viaggiatori in attesa di arrivo.

Il Passaggio di un Treno da una regione alla successiva può essere schematizzato come segue: siano  $G1$  e  $G2$  Stazioni di Gateway connesse, che colleghino le regioni  $R1$  ed  $R2$ , con  $G1 \in R1$ , e sia  $G2 \in R2$ . Un Percorso che attraversa i due Gateway conterrà almeno una Tappa  $T1$  tale per cui il campo `next_station` avrà il valore  $G1$ , `destination_platform` una delle Piattaforme possibili per effettuare l'accesso  $P$ , e `region` la regione  $R1$ , e una tappa  $T2$  tale per cui il campo `next_station` avrà il valore  $G2$ , `destination_platform` la stessa piattaforma specificata in  $T$ , e `region` la regione  $R2$ . Le operazioni eseguite saranno le seguenti:

- Il Treno corrente  $T$  effettua l'accesso alla Stazione  $G1$ , Piattaforma  $P$ .
- $T$ , se previsto dal Percorso, effettua Discesa dei Viaggiatori. Tale azione è simile alla Discesa descritta in sezione 3.3.2.2, ma se un Viaggiatore una volta sceso proseguirà il proprio percorso su un nodo diverso da quello corrente, esso verrà trasferito mediante messaggio remoto al nodo successivo, e solo a questo punto verrà inserita l'Operazione `ENTER` del Viaggiatore nella coda di operazioni di `Traveler Pool` locale al nodo destinazione.
- Il descrittore del Treno corrente  $D_T$  viene serializzato (*marshalling*), e inviato tramite invocazione remota alla Stazione di Gateway della Regione  $G2$  ad essa connessa. L'individuazione dell'indirizzo della regione

specificata nel parametro **region**, avviene interrogando il **Name\_Server**, se esso non è già presente in una cache locale.

- Il flusso ciclico di istruzioni eseguite dal thread che gestisce  $T$  viene interrotto (esso potrà così ottenere un nuovo Descrittore di Treno ed eseguire per esso le proprie operazioni).
- Presso la regione  $R2$ , stazione di Gateway  $G2$ , vengono eseguite le seguenti operazioni:
  - Il Descrittore viene de-serializzato (*unmarshalling*).
  - I dati del Descrittore  $D_T$  presenti nella regione  $R2$  vengono aggiornati con quelli ricevuti.
  - Viene ottenuta la Piattaforma in mutua esclusione e ne viene impostato il campo **Free** a **False**.
  - Vengono operate attesa fino all'orario di partenza previsto (secondo il clock del nodo corrente), Salita dei Viaggiatori e Partenza dalla Stazione come per le Stazioni Regionali.
  - Viene restituito un messaggio di *acknowledgement* al nodo  $R1$ , che comunica l'avvenuta esecuzione delle operazioni.
- Una volta ricevuto il messaggio di *acknowledgement*, presso il nodo  $R1$ , viene liberata la piattaforma corrente (**Free** viene impostato a **True**) in modo tale da permettere ad altri Treni di occuparla.

Ciò che garantisce la correttezza della soluzione presentata, relativamente a discesa e salita dei passeggeri locale, è la modalità con la quale il Viaggiatore viene accodato presso la Stazione di Gateway come descritto nella sezione 3.3.1, ovvero in modo tale per cui:

- Presso le coda **Arrival\_Queue** delle Piattaforme nel nodo di partenza vi saranno tutti e soli i passeggeri in attesa di scendere presso la Stazione di Gateway, se previsto dal loro Biglietto.
- Presso le code **Leaving\_Queue** delle Piattaforme nel nodo di destinazione vi saranno invece solamente i Viaggiatori che vogliono raggiungere una destinazione interna al nodo corrente.

Si noti che la soluzione riportata prevede lo scambio di due messaggi remoti, il primo per la transizione dei Treni tra le Regioni, e il secondo di *acknowledgement* per poter liberare la risorsa Piattaforma presso il nodo di partenza. Di fatto, data l'inaffidabilità della rete, è possibile che uno dei due messaggi scambiati non venga ricevuto dal nodo di destinazione. Abbiamo quindi due casi:

- *L'invio del primo messaggio fallisce.* In questo caso, il flusso di esecuzione del thread che gestisce il Treno viene interrotto. Per semplicità si può pensare che il Treno sia cancellato a causa di un guasto, o si può ridurre il Percorso fino alla Tappa che ha causato l'errore, dopo essersi accertati dell'effettiva irraggiungibilità del nodo destinazione.
- *Il messaggio di acknowledgement non viene consegnato.* In questo caso il problema è più grave. Il Treno presso il nodo di destinazione continua la propria corsa, mentre la Piattaforma abbandonata rimane occupata, e quindi inutilizzabile dai Treni che sopraggiungono. Per risolvere questo tipo di problema è possibile prevedere un tempo massimo di occupazione della Piattaforma, oltre il quale richiedere l'effettivo stato di occupazione della Piattaforma presso la Stazione di Gateway della Regione di destinazione e, nel caso in cui risultasse libera, procedere alla liberazione della stessa, impostando **Free** a **True**.

### 3.3.3 Creazione di un Biglietto

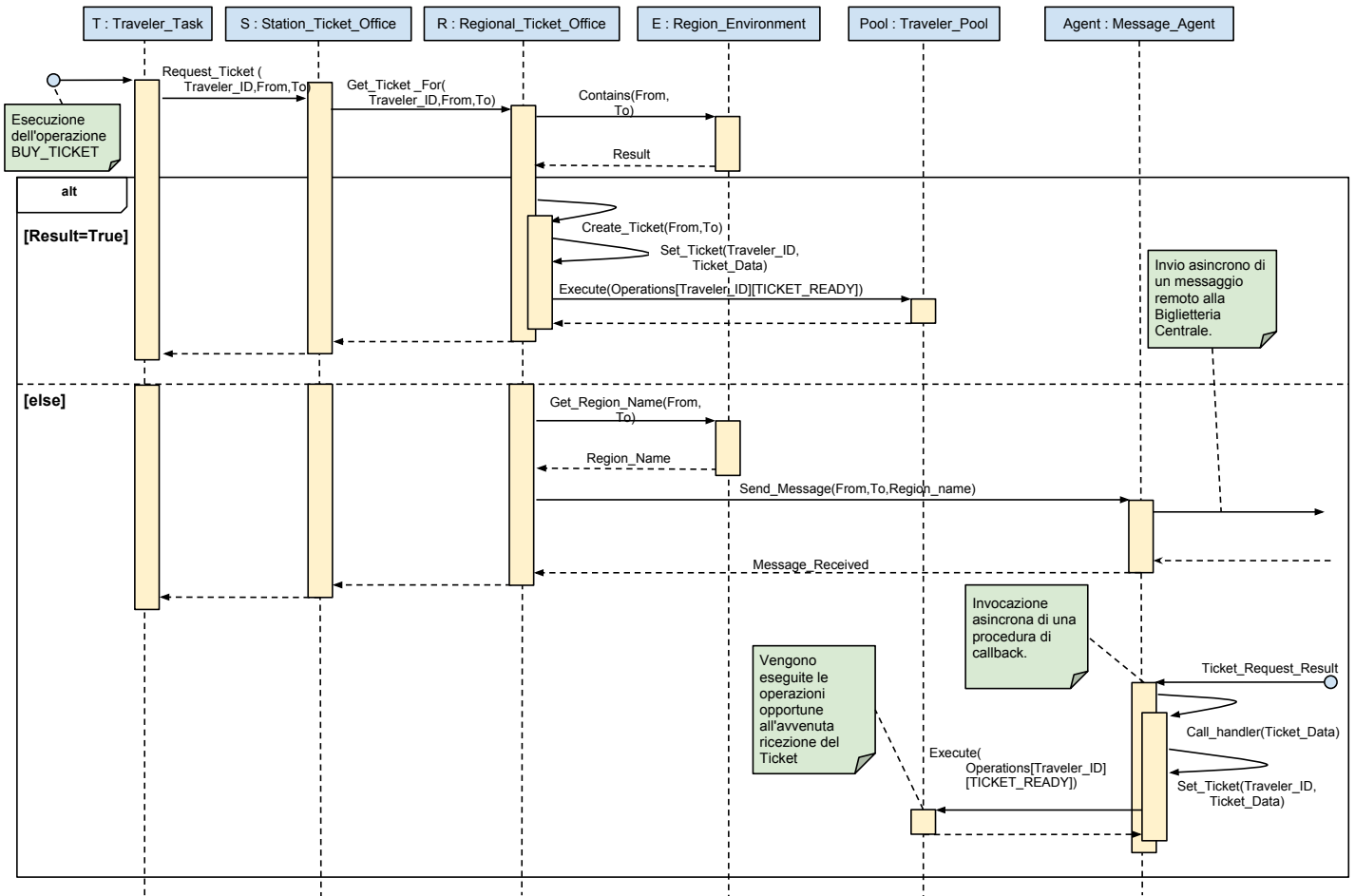


Figura 3.4: Diagramma di Sequenza, operazioni necessarie alla creazione di un Ticket.

Per la creazione di un Biglietto, necessario per permettere ai Viaggiatori di poter raggiungere una destinazione, ho utilizzato un algoritmo che coinvolge le componenti (distribuite) introdotte in sezione 3.1.2. Il diagramma di Sequenza in figura 3.4 fornisce una rappresentazione delle fasi successive di creazione.

La descrizione sarà suddivisa in fasi successive:

#### 3.3.3.1 Richiesta di Creazione

La richiesta di creazione di un Ticket da parte di un Viaggiatore, viene operata da un thread appartenente al pool di **Traveler\_Pool** che esegue l'ope-

razione `CREATE_TICKET` (come descritto precedentemente in sezione 3.3.1.1). Tale operazione, effettua una richiesta di creazione alla stazione di partenza *S*, la quale inoltrerà la richiesta alla propria Biglietteria interna. Quest'ultima richiederà la creazione effettiva alla Biglietteria Regionale, se lo stesso `Ticket` non è presente in una cache locale.

#### 3.3.3.2 Creazione del Ticket

Siano `From` e `To` rispettivamente l'identificativo della Stazione di partenza, e l'identificativo della Stazione di destinazione. Le operazioni svolte dalla procedura invocata presso la Stazione Regionale sono le seguenti:

- Se `To` appartiene alla Regione corrente, viene creato il `Ticket`, effettuando le seguenti operazioni:
  - Si considera il percorso più breve da `From` a `To`, che indicheremo con  $Path = \{s_1, \dots, s_N\}$ . Questo percorso è ottenuto applicando un semplice algoritmo di cammino minimo sul grafo avente come vertici le Stazioni, e come archi non direzionati i Segmenti che li collegano. Il cammino può minimizzare la lunghezza totale del percorso.
  - Il percorso *Path* viene intersecato con i Percorsi dei Treni (`Route`), per poter definire le Tappe che compongono il Biglietto. Sia  $i = 1$ , allora finché  $i \leq N$ :
    - \* Ottieni i percorsi dei Treni (`Routes`)  $R = r_1, \dots, r_k$  che contengono una Tappa *t* tale per cui i campi `Start_Station` e `Next_Station` sono rispettivamente  $s_i$  e  $s_{i+1}$ . Per ciascuno di questi percorsi, vengono ottenuti indice e posizione della tappa *t* al suo interno. A questo punto si procede all'individuazione del percorso che meglio si adatta al cammino minimo da seguire. Sia quindi  $k = i$ ; per ciascuna route  $r_j \in R$ , a partire dalla posizione di *t* in  $r_j$ , si procede ad estendere la corrispondenza. Viene quindi mantenuto il percorso con la corrispondenza di lunghezza massima *Max\_Length*, memorizzandone l'indice in *Max\_Match*.
    - \* Viene quindi creata una Tappa del `Ticket`:
      - `start_station` : indice della stazione di inizio dell'estensione,
      - `next_station` : indice della stazione di fine dell'estensione massima,

- `train_id` : indice del Treno che percorre il percorso di indice `Max_Match`,
- `start_platform` : indice della Piattaforma di partenza in `t`,
- `destination_platform` : indice della Piattaforma dell'ultima tappa
- `next_region` : regione corrente,

\* *i* viene incrementato di *Max\_Length*.

- Il `Ticket` creato viene assegnato al Viaggiatore e viene quindi inserita l'operazione `TICKET_READY` nella coda di operazioni di `Traveler_Pool`.
- Se `To` non appartiene alla Regione corrente, allora viene effettuata una richiesta remota alla Biglietteria Centrale. Tale richiesta è asincrona, in modo tale da permettere al thread che sta effettuando la richiesta di creazione del Biglietto di non effettuare attesa attiva, e quindi di poter eseguire altre operazioni eventualmente presenti nella coda di `Traveler_Pool`.

#### 3.3.3.3 Richiesta di Creazione Remota

Una volta che la Biglietteria Centrale riceve, tramite l'interfaccia remota esposta, un messaggio remoto di richiesta di Creazione di un Biglietto, essa effettua 3 operazioni:

- Individua la Regione di appartenenza della destinazione *To*; se tale informazione non è presente nella cache locale mantenuta dalla Biglietteria Centrale, essa viene ricercata nel modo seguente:
  - viene recuperato l'elenco completo delle Regioni (ovvero una lista di coppie (`Node_Name`, `Node_Address`)) dal *Server dei Nomi*;
  - per ciascun elemento dell'elenco, viene inviata una richiesta remota, alla quale ciascuna Regione risponde con `True`, se contiene la Stazione, o `False` in caso contrario.
- Se nessuna risposta positiva viene ricevuta, allora viene comunicato l'errore alla Biglietteria Regionale richiedente, inviando un messaggio di errore.
- Nel caso in cui si abbia una risposta positiva da una Regione *R* allora:

- Viene recuperata la lista di Regioni attraverso le quali costruire il percorso per raggiungere la destinazione (dalla mappa **Links**).
- A ciascuna stazione di questa lista viene inviata una richiesta remota *sincrona* per ottenere dei Ticket che collegati, permettono di raggiungere **To** a partire da **From**.
- I vari **Ticket** ottenuti vengono poi uniti (eliminati i passaggi per le stazioni di gateway se non necessari) e il Biglietto risultante inviato come risposta al Nodo richiedente, presso il quale verrà assegnato al viaggiatore richiedente, e verrà inserita l'operazione **TICKET\_READY** nella coda di **Traveler\_Pool**.

### **3.3.4 Terminazione distribuita**



# Capitolo 4

## Tecnologie Adottate

La scelta delle tecnologie per l'implementazione del prototipo realizzato, è stata operata nell'ottica di individuare strumenti che permettessero una agevole attuazione della soluzione presentata.

Sono stati utilizzati principalmente due linguaggi di programmazione.

### Scala

Il linguaggio Scala è un linguaggio funzionale Object Oriented. Ho utilizzato questo linguaggio per la realizzazione di

- **Controller Centrale**
- **Biglietteria Centrale**
- **Application Server**, al quale i client HTTP possono collegarsi mediante protocollo *Web Socket* per poter fornire una rappresentazione grafica della simulazione, in linguaggio HTML e Javascript. Per la sua realizzazione è stato utilizzato il framework MVC Play 2.0 (disponibile all'indirizzo <http://www.playframework.com/>).
- **Name Server** per la gestione delle Regioni di simulazione.

Le interazione tra thread interne alle componenti distribuite, sono state gestite adottando il modello di concorrenza ad Attori fornito nativamente dal linguaggio Scala.

### Ada

Il linguaggio Ada è stato utilizzato per realizzare il core di simulazione presente in ciascuna Regione. Il modello di concorrenza fornito dal linguaggio è sembrato per me più adatto, rispetto ai modelli offerti da altri linguaggi, per rappresentare le interazioni tra le entità introdotte

---

al capitolo precedente. In prima battuta, ho valutato l'utilizzo del modello di concorrenza ad Attori di Scala; esso avrebbe fornito un meccanismo di linguaggio per l'esecuzione delle entità Viaggiatore e Treno, simile a quello descritto per nelle sezioni 3.2.2 e 3.2.3. Tuttavia, il modello ad Attori avrebbe reso necessaria l'adozione di entità attive Server a protezione delle entità reattive come Segmenti e Piattaforme, e ciò avrebbe comportato:

- la necessità di utilizzare thread per l'esecuzione degli attori a protezione delle entità;
- una maggiore complessità di terminazione;

L'utilizzo di Ada ha permesso un buon livello di controllo di allocazione di thread, risorse protette e oggetti, e il suo sistema di tipi molto restrittivo ha consentito di ridurre i possibili errori in fase di sviluppo.

L'interazione tra componenti remote del sistema, realizzate con tecnologie omogenee, è stata possibile mediante il middleware per lo scambio di messaggi Yami4 (<http://www.inspire1.com/yami4/>), compatibile, tra gli altri linguaggi, con Ada e Java (e quindi di conseguenza anche con Scala). Questo strumento è stato preferito agli altri possibili middleware (Distributed Systems Annex per RPC in Ada e CORBA) in quanto si è rivelato molto semplice da utilizzare e ha permesso di adottare un certo grado di disaccoppiamento tecnologico tra le componenti. Per ciascun Nodo di simulazione del Sistema è stato utilizzato un singolo Attore (definito da Yami4), sia per quanto riguarda l'invio che la ricezione di messaggi remoti, in ascolto ad un singolo indirizzo (adottando il protocollo TCP). Per ciascun Attore, è possibile infatti definire:

- **Object**: un oggetto remoto raggiungibile ad un dato indirizzo, identificato univocamente da una stringa.
- **Services**: insieme di servizi offerti da un oggetto remoto, identificati univocamente da una stringa.

Per quanto riguarda i nodi di simulazione, ho utilizzato quindi un unico **Object**, il quale mette a disposizione vari **Service** a seconda del tipo di azione da prendere. Questa soluzione mi è sembrata la più semplice e estendibile possibile. Il protocollo usato tra le varie componenti del sistema per lo scambio di messaggi, comprende il protocollo definito dal stesso middleware Yami4, e quindi lo scambio di dati in formato stringa chiave-valore, sia l'utilizzo dello standard JSON per il marshalling dei dati delle entità coinvolte. Per la codifica e decodifica di stringhe JSON ho utilizzato:

- 
- La libreria `Gnatcoll` fornita con la distribuzione `Gnat`, per il linguaggio Ada.
  - La libreria `json-smart` (disponibile all'indirizzo <https://code.google.com/p/json-smart/>), scritta in Java e integrata nelle componenti realizzate in linguaggio Scala. L'ho preferita alla libreria standard Scala `scala.util.parsing.json` poiché più semplice da utilizzare.

Ho utilizzato il formato JSON anche per la definizione dei file di configurazione, utilizzati all'avvio del sistema.

## 4.1 Specifica di alto livello