Railway Simulation

Moreno Ambrosin
mat. 1035635

Progetto Sistemi Concorrenti e Distribuiti

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

21 maggio 2013

Indice

Elenco delle figure

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.
- 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, ne sull'ambiente di esecuzione;
- è opportuno prevedere un certo livello di distribuzione delle componenti del sistema;
- si possono presentare difficoltà nella trasmissione di dati tra le componenti dovute all'inaffidabilità della rete.

Di seguito andrò ad analizzare gli aspetti più problematici, in termini di Concorrenza e Distribuzione.

2.1 Distribuzione

Nel progetto di un simulatore di un sistema ferroviario, 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.
- É 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.

É 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. La natura del problema infatti, prevede che ci sia un certo grado 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. É 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 dovrà rimanere in esecuzione o in attesa).

2.1.4 Avvio del Sistema

L'*Avvio del sistema* dove 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 di un Treno in un Segmento

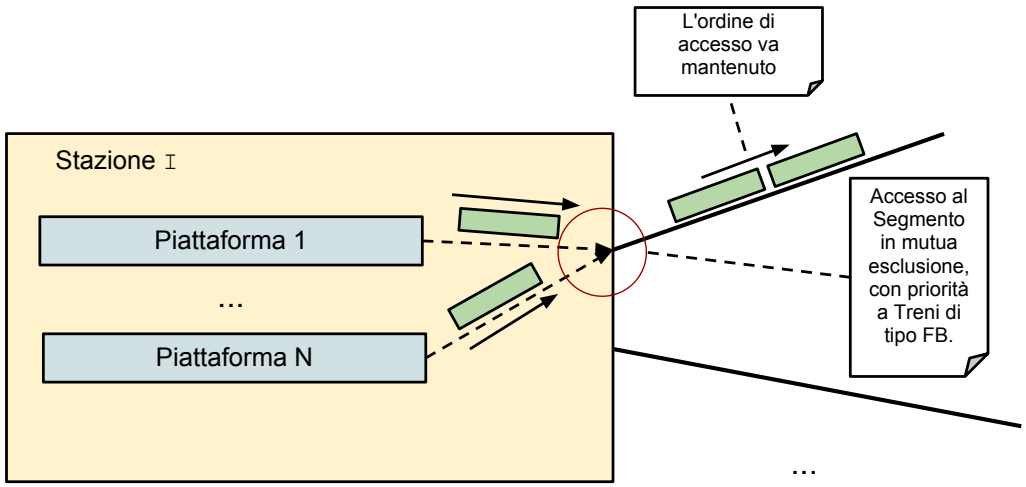

Figura 2.1: Accesso ad un Segmento da Parte di uno o più Treni.

L'accesso ad un Segmento di collegamento tra due Stazioni da parte di un Treno, avviene in maniera concorrente. Tale azione presenta infatti i seguenti requisiti:

- L'ingresso presso un Segmento deve avvenire in *mutua esclusione* tra i Treni; 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;
  - la necessità di adottare un meccanismo che garantisca a tutti i Treni di poter accedere al Segmento (nessun Treno in attesa infinita).
- Dev'essere data precedenza d'accesso al Segmento, ai Treni di tipo `FB`.

2.2.2 Uscita da un Segmento e accesso alla Stazione

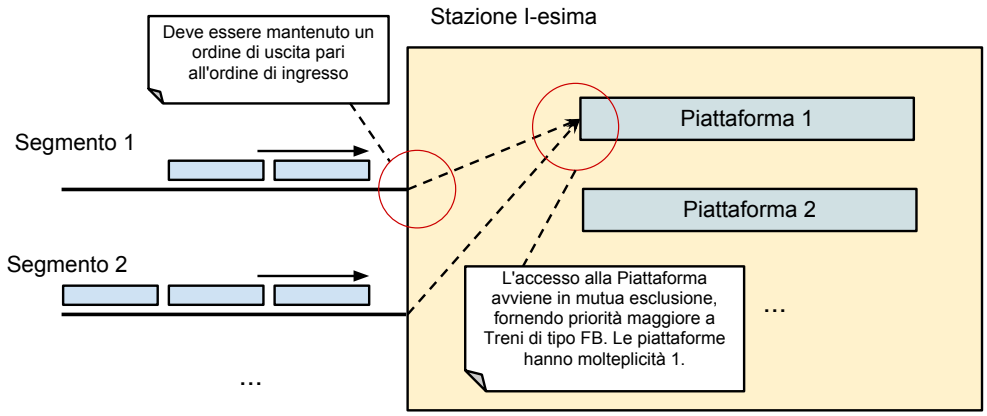

Figura 2.2: Uscita da un Segmento e accesso ad una Piattaforma di uno o più Treni.

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 della 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 Regionali, 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. É 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

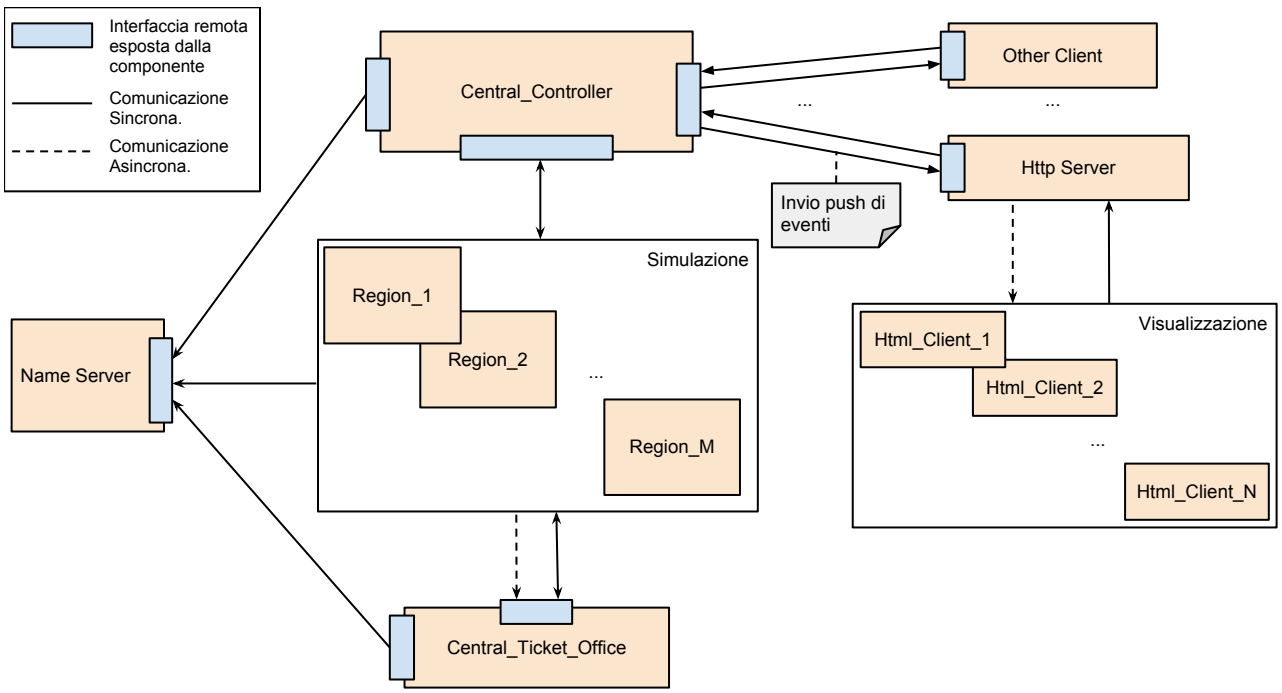

Figura 3.1: Diagramma informale che presenta una progettazione logica delle componenti distribuite.

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 potranno risiedere 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.

Questa scelta comporta 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.

3.1.1.1 Stazioni di Gateway

Nella progettazione di un simulatore distribuito su più Regioni, è necessario prevedere uno o più punti di collegamento tre 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 logicamente come un'unica Stazione, e ciascuna delle N Stazioni di Gateway 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.*

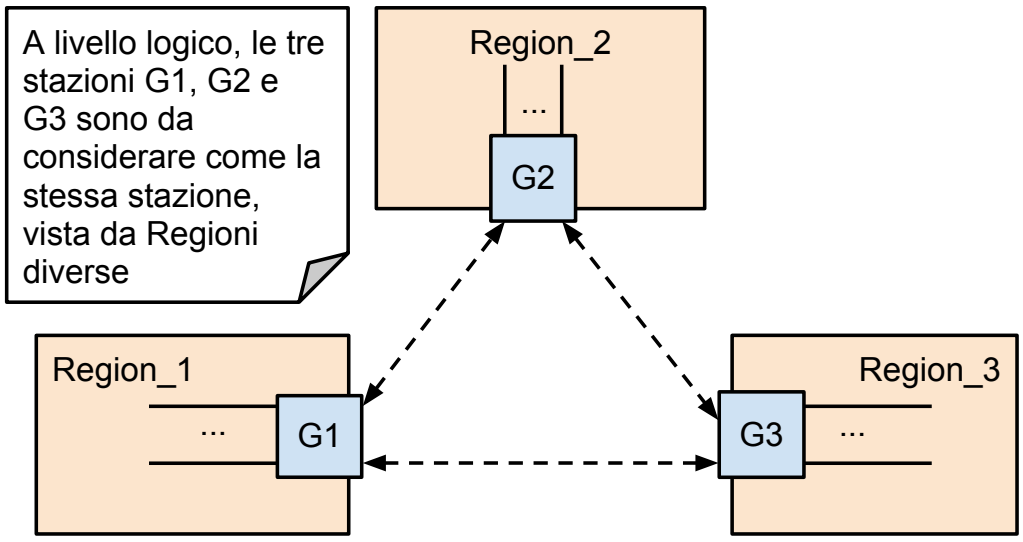

Figura 3.2: Utilizzo di stazioni di Gateway per collegare tre Regioni.

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 $\{P_1, ..., P_M\}$, le quali permettono un *unico senso di percorrenza*, ovvero:

- Le prime $\{P_1, ..., P_K\}$ saranno disponibili all'accesso dalla Regione in cui G risiede.
- Le Piattaforme $\{P_K + 1, .., P_M\}$ 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, e presentata in sezione 3.3.2.4. 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 Centrale
: 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 di Gateway, 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 *Controllo Centrale* è una entità distribuita, alla quale tutti i nodi inviano Eventi per notificare lo stato di avanzamento globale della simulazione, che viene memorizzato in strutture dati opportune. 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 Centrale sarà anche responsabile dell'invio di un *segnale di terminazione*, che verrà recapitato a tutti i nodi che compongono la simulazione, mediante la procedura descritta 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 con agente di controllo, a molteplicità* $N \geq 1$, 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 di occupazione.

3.2.2 Viaggiatore

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

Nel caso in cui il Viaggiatore fosse messo in relazione 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 e quindi non desiderabile.
- La replicazione del thread che rappresenta il Viaggiatore su tutti i Nodi di Simulazione, attivato (reso competitivo nell'ottenimento della CPU) o meno a seconda della presenza del Viaggiatore sulla Regione. 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 le informazioni che lo contraddisinguono e le operazioni che esso eseguirà (`Traveler_Descriptor`). Il cambio di Regione di un Viaggiatore potrà quindi essere ottenuto semplicemente aggiornando le informazioni relative al Viaggiatore da trasferire, sul Nodo di destinazione, e inserendo la prossima operazione da eseguire per esso nella coda di `Traveler_Pool` 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 Treno 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 una Regione (Nodo) ad un'altra, 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 thread sarà tale per cui una volta ottenuto un descrittore dalla coda, eseguirà per lui una fissata sequenza di azioni. Le interazioni tra le entità del

sistema, prevedono la possibilità che i thread del pool vengano posti in attesa su variabili di condizione, in attesa di eventi: è quindi necessario che la dimensione del pool di thread sia pari al massimo numero di Treni circolanti nella Regione alla quale esso appartiene, in modo tale da evitare situazioni di stallo dei thread in attesa. Infine, per fornire una garanzia di esecuzione più elevata a Treni di tipo `FB`, è possibile introdurre una coda per tipologia di Treno, ciascuna servita da un pool dedicato a dimensione fissata.

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, permettendo così un agevole trasferimento tra nodi diversi.

3.2.4 Stazione

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

- un certo numero $n \geq 2$ 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 con agente di controllo, 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. Per i dettagli sulla creazione di un biglietto, si rimanda alla sezione 3.3.3.

3.2.4.3 Pannello Informativo

Un Pannello Informativo è modellato 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

Viene ora presentata una descrizione delle interazioni tra le varie entità introdotte nella sezione precedente. Le entità reattive vengono descritte supponendo l'adozione del costrutto *monitor* a protezione di esse, avendo quindi a disposizione una struttura dati che incapsula la risorsa esponendo procedure accessibili in mutua esclusione dai thread; inoltre si suppone l'esistenza dei costrutti `wait(var_condizione)` e `signal(var_condizione)`, i quali rispettivamente forzano l'attesa del chiamante su coda FIFO, e risvegliano il primo processo in attesa su tale coda. Si suppone inoltre la presenza di un costrutto `signal_all(val_condizione)` il quale risveglia ordinatamente tutti i thread in attesa su una data variabile di condizione.

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
  - run_number
  - next_region
  ```

  e da un indice della prossima tappa del Percorso (`Next_Stage`). Nel caso in cui il Treno `train_id` sia un Treno a prenotazione, il valore di `run_number` sarà > 0.

- 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 precondizione 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 FIFO `Leaving_Queue` della Piattaforma indicata da `start_platform`, che sarà 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) contente 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`.

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). 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 3.3.2.3). 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`) che comprende andata e ritorno, composto da una sequenze di Tappe (`Stage`) successive, ciascuna contenente i seguenti campi:

```
- start_station,
- start_platform,
- next_segment,
- next_station,
- next_platfom,
- 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. Il percorso di ciascun Treno è scandito da una *Tabella degli Orari* (`Time_Table`), la quale definisce per $N > 1$ *Corse* (`Runs`) successive del Percorso, gli *orari di partenza* dalle Stazioni da rispettare per ciascuna Tappa.

Una volta che un thread del pool della struttura `Train_Pool` ottiene un Descrittore, effettua le seguenti operazioni, per ciascuna Tappa del Percorso:

- Partenza dalla Stazione `start_station`, Piattaforma `start_platfom` all'orario indicato da `Time_Table` per la Tappa corrente; se previsto (`action = ENTER`) allora effettua la salita dei Viaggiatori.
- 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_platfom`, per eseguire l'azione `action`.
- Se `action = ENTER` allora effettua discesa 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 Gestione della Tabella degli Orari

Per limitare la dimensione della Tabella degli Orari per ciascun percorso, viene mantenuto un numero N di corse. La Tabella viene fornita dalla Biglietteria Centrale, la quale dovrà essere sempre aggiornata relativamente alla corsa correntemente percorsa da un Treno per poter permettere la prenotazione dei Biglietti per Treni di tipo `FB` (si veda il paragrafo 3.3.3.4). Per poter consultare la `Time_Table`, vengono quindi utilizzati due indici:

- `Current_Run` che mantiene l'indice della Corsa corrente.
- `Current_Stage` che serve ad accedere l'orario per la tappa successiva.

Tali indici vengono aggiornati dal Treno al momento della partenza da una Stazione per raggiungere la Tappa successiva. Nel caso in cui il Treno ultimi una corsa, esso incrementa l'indice `Current_Run`, ed invia un messaggio remoto *sincrono* alla Biglietteria Centrale comunicando il passaggio alla Corsa successiva. Ci sono due possibili risultati alla richiesta di aggiornamento:

- Si è raggiunta l'ultima Corsa ($N-esima$), e quindi viene restituita una nuova `Time_Table` contenente gli orari per N corse, composta dalla corsa $N - esima$ della Tabella precedente in prima posizione, e da $N - 1$ corse calcolate. L'indice `Current_Run` viene posto ad 1, e viene memorizzato l'identificativo della Corsa corrente `Current_Run_Id`, il quale può essere ad esempio un numero progressivo, che la identifica.
- Non si è raggiunta l'ultima corsa, e quindi viene aggiornata la corsa corrente presso la Biglietteria Centrale e restituito l'identificativo della stessa.

La scelta di riportare solo l'orario di partenza nella definizione della Tabella degli orari, permette di risolvere i problemi causati dal passaggio di alcuni Treni attraverso Regioni su nodi distribuiti, dovuti, ad esempio, dai clock non sincronizzati dei nodi di calcolo che compongono il sistema, o dai ritardi che possono essere introdotti dalla trasmissione sulla rete. Si supponga una differenza in valore assoluto Δt tra i clock t_1 e t_2 di due Nodi N_1 ed N_2, e supponiamo di avere un treno T che viaggia da N_1 ad N_2. Si presentano i seguenti due casi:

- $t_1 = t_2 + \Delta t$: il Treno T localmente ad N_2 potrà avere un ritardo aggiuntivo compreso tra 0 e Δt rispetto all'eventuale ritardo accumulato in N_1;
- $t_2 = t_1 + \Delta t$: il Treno T si ritroverà in anticipo rispetto al clock di N_1, e quindi vi è la possibilità che prolunghi la sua attesa.

3.3.2.2 Accesso ad un Segmento

Un Segmento può essere rappresentato da una struttura dati `Segment` che espone una interfaccia semplice, che permette di effettuare:

- Ingresso (`Enter`).
- Uscita (`Leave`).

Ingresso

La richiesta di accesso (`Enter`) al segmento avviene in mutua esclusione tra tutti le entità Treno. A protezione dell'accesso ad un Segmento, viene utilizzata una struttura dati di tipo *monitor*, che chiameremo `Segment_Monitor`, che sarà contenuta nella struttura dati `Segment`. Essa possiede un flag booleano `Free` che permette di indicare lo stato occupato della risorsa monitor; una prima definizione della procedura protetta `Enter_Monitor` a gestione dell'ingresso prevede l'esecuzione delle seguenti operazioni:

- Se `Free=False` e la direzione di percorrenza corrente è opposta a quella del Treno che vuole accedere, allora il thread corrente si pone in attesa su condizione, `wait(Not_Free)`. I thread in attesa verranno risvegliati dalla procedura `Leave_Monitor`.
- Altrimenti,
  - il Descrittore corrente viene inserito nella coda `Train_Queue`;
  - Viene aggiornata la velocità di percorrenza del Treno entrato, in base a quella dei Treni che lo precedono.
  - Nel caso in cui la risorsa risulti vuota (`Free=True`), allora viene modificato il valore di `Free` a `False` , e memorizzata la stazione di provenienza del Treno.

Una volta terminate queste operazioni, il thread rappresentante il Treno rilascia la risorsa e, basandosi sulla lunghezza del Segmento e sulla velocità da mantenere, simula la percorrenza rendendosi inattivo (non

competitivo per l'ottenimento della CPU) per un tempo dato dalla semplice equazione: $Time = Segment\_Length/Actual\_Speed$.

La semantica di base descritta per `Enter_Monitor` non è completa: essa infatti non garantisce che tutti i thread che rimarranno in attesa sulla condizione `Not_Free` potranno eseguire, e può quindi portare a *starvation*. Una prima estensione possibile per evitare questo problema è garantire ai thread in attesa su condizione una *priorità maggiore nell'accedere* al Segmento appena esso diviene libero rispetto a quelli che sopraggiungono successivamente. Sono inoltre introdotte due variabili di condizione, `Can_Enter_First_End` e `Can_Enter_Second_End`, a sostituzione di `Not_Free`, per poter porre i thread in attesa sull'estremo di Segmento dal quale hanno tentato l'ingresso.

Questa accorgimenti non risultano però sufficiente a garantire che i Treni in attesa avranno accesso al Segmento in qualche momento: si consideri, ad esempio, 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 a questo problema, è prevedere un *numero massimo di accessi consecutivi per direzione*, raggiunto il quale, un thread viene posto in attesa sulla variabile di condizione relativa all'estremo dal quale tenta di effettuare l'accesso. Sia `Access_Number` il numero di accessi nella direzione corrente, e sia `Current_Direction` la direzione di percorrenza corrente, che indica l'accesso da uno dei due estremi del Segmento.

La semantica completa adottata per la regolazione dell'accesso (`Enter_Monitor`) è la seguente:

- Sia T il thread Treno che esegue correntemente. Esso potrà accedere al segmento se e soltanto se una delle seguenti condizioni è verificata:
  - `Free = True`
  - `Free = False`, `Current_Direction` coincide con l'estremo dal quale accede T ma il numero massimo di accessi per direzione non è stato raggiunto.
  - `Free = False`, `Current_Direction` coincide con l'estremo dal quale accede T, e il numero massimo di accessi per direzione

è stato raggiunto ma il numero di Treni in attesa di accedere in direzione opposta è 0.

- Se T non può accedere, allora a seconda della direzione con la quale intende percorrere il Segmento, viene posto in attesa su condizione `Can_Enter_First_End` oppure `Can_Enter_Second_End` mediante il costrutto `wait`. Una possibile realizzazione in pseudo-codice della procedura `Enter_Monitor` della risorsa *monitor* a protezione del Segmento è presentata nel listato 3.1.

Listing 3.1: Procedura protetta `Enter_Monitor` per l'accesso al Segmento.

```
monitor Segment_Monitor
  ...
  procedure Enter_Monitor(T:Train,Access_End:integer)
  begin
    ...
    if Access_End = First_End then
      // Accesso dal primo estremo del Segmento
      while
        // Se il Segmento non e' libero perche'
        // occupato da Treni in transito nella
        // direzione opposta...
        ((not Free) and
         (Access_End /= Current_Direction)) or
        // ...oppure se la direzione di accesso
        // del Treno corrente e' la stessa dei
        // Treni in transito ma il numero massimo
        // di accessi e' stato raggiunto e nella
        // coda di attesa presso la direzione
        // opposta vi sono Treni in attesa...
        ((not Free) and
         (Access_End = Current_Direction) and
         (Access_Number = MAX) and
         (First_End_Count > 0)) loop
        // ...il chiamante e' posto in attesa
        // su condizione.
        wait(Can_Enter_First_End);
      end loop;
    else
      while
        ((not Free) and
         (Access_End /= Current_Direction)) or
        ((not Free) and
```

```
            (Access_End = Current_Direction) and
            (Access_Number = MAX) and
            (First_End_Count > 0)) loop
          wait(Can_Enter_Second_End);
        end loop;
      end if;
      // A questo punto il Treno puo' accedere.
      ...
    end;
...
end monitor;
```

- Una volta che T ha ottenuto l'accesso:
  - Se `Free = True` allora occupa il Segmento (imposta `Free` a `False`) e assegna a `Access_Number` il valore 1, se la direzione di T è diversa da quella corrente memorizzata nel Segmento;
  - Altrimenti, incrementa di 1 il valore di `Access_Number` se esso è inferiore al limite massimo consentito.
  - In ciascuno dei due casi infine, inserisce il Descrittore del Treno nella coda `Train_Queue` e aggiorna la velocità di percorrenza.

Una volta che il Segmento si sarà liberato (`Free=True` dopo l'invocazione di `Leave_Monitor` da parte di tutti i thread che rappresentano i Treni in transito) allora ciascun Treno in attesa nella coda relativa all'estremità opposta a quella di accesso dell'ultimo Treno uscito, potrà ritentare l'accesso al Segmento (la coda viene svuotata e ripopolata).

Vi è infine un ultima estensione al protocollo di accesso al Segmento, necessaria al fine di garantisce accesso preferenziale ai treni di tipo `FB`. Per introdurre priorità senza poter fare assunzioni sulle politiche di scheduling che verranno utilizzate è possibile procedere nel seguente modo: viene introdotta una nuova risorsa protetta da *monitor*, `Priority_Access_Handler`, la quale contiene due variabili intere `FB_Count` e `Regional_Count`, un flag booleano `Free` e due varibili di condizione `Can_Access_FB` e `Can_Access_Regional`. Tale risorsa avrà il comportamento descritto nel listato 3.2,

Listing 3.2: monitor utilizzato per garantire accesso preferenziale a Treni di tipo `FB`.

```
monitor Priority_Access_Handler
  ...
```

```
  procedure Gain_Access(T:Train)
  begin
    if not Free then
      if T.Type = FB then
        FB_Count := FB_Count + 1;
        wait(Can_Access_FB);
        FB_Count := FB_Count - 1;
      else
        Regional_Count := Regional_Count + 1;
        wait(Can_Access_Regional);
        Regional_Count := Regional_Count - 1;
      end if;
    end if;
    Free := False;
  end;

  procedure Access_Gained()
  begin
    Free := True;
    if FB_Count > 0 then
      signal(Can_Access_FB);
    else
      signal(Can_Access_Regional);
    end if;
  end;
end monitor;
```

e può essere utilizzata nel modo seguente: supponiamo che la struttura dati `Segment` contenga un oggetto `Priority_Access_Handler`, e che esponga una procedura `Add_Train`, la quale aggiunge un dato Descrittore di Treno ad una coda, `Trains_Order`, interna all'oggetto `Segment_Monitor`. La procedure `Enter` sarà quindi composta dalle seguenti azioni:

- Viene richiesto l'accesso con la procedura protetta `Gain_Access` sull'oggetto di tipo `Priority_Access_Handler` mantenuto.
- Viene aggiunto il Descrittore del Treno corrente alla coda che regola l'ordine di esecuzione, con `Add_Train`.
- Viene liberata la risorsa `Priority_Access_Handler` con la procedura `Access_Gained`.
- Viene richiesto l'accesso al Segmento con la procedura `Enter` della risorsa protetta di tipo `Segment_Monitor` utilizzata.

Un esempio di realizzazione della procedura `Enter` è riportato nel listato 3.3.

Listing 3.3: Esempio di procedura `Enter` che realizza il protocollo di accesso.

```
...
procedure Enter(Train:Train,Segment:Segment) is
begin
  // Se l'oggetto Priority_Controller e' libero,
  // viene occupato e si procede oltre, altrimenti
  // il thread corrente si trovera' in attesa su
  // variabile di condizione Can_Access_FB o
  // Can_Access_Regional.
  Segment.Priority_Controller.Gain_Access(Train);
  // A questo punto il thread corrente e' l'unico
  // in esecuzione e quindi puo' inserire il
  // descrittore nella coda interna a Segment_Monitor.
  // Nel frattempo altri thread possono attendere
  // nelle code di attesa di Priority_Controller.
  Segment.Segment_Monitor.Add_Train(Train);
  // Una volta eseguito l'operazione Add_Train si
  // puo' liberare l'Access_Controller
  Segment.Priority_Controller.Access_Gained();
  // A questo punto viene fatta la richiesta
  // di Accesso. Anche in caso di prerilascio,
  // esso verra' attuato secondo l'ordine
  // descritto nella coda interna Trains_Order.
  Segment.Segment_Monitor.Enter(Train);
end;
```

La procedure protetta `Enter_Monitor` di `Segmen_Monitor`, introdotta nel listato 3.1, per garantire l'accesso secondo l'ordine riportato nella coda `Trains_Order`, deve essere quindi modificata. Il listato 3.4 riporta le modifiche necessarie.

Listing 3.4: Struttura della risorsa *monitor* che regola l'accesso al Segmento.

```
monitor Segment_Monitor
  ...
  procedure Enter(T : Train)
  begin
    // Se il Treno corrente T non e' il
    // prossimo designato, attende.
    while(Trains_Order.First_Element() /= T) loop
      wait(Not_First);
```

```
      end loop;
      Trains_Order.Dequeue(T1);
      // Invoca una signal su variabile di
      // condizione, risvegliando in maniera
      // ordinata i thread in attesa.
      // Fino a quando il thread corrente non
      // abbandonera' la risorsa (ovvero
      // terminera' l'esecuzione della procedura
      // corrente) essi non potranno eseguire.
      signal_all(Not_First);
      // Procedi con l'accesso secondo la
      // semantica descritta
      ...
    end;
end monitor;
```

Per semplicità di simulazione, si può assumere che vi sia uno spazio apposito dove i Treni possano attendere affinché si liberi il Segmento.

Uscita

L'uscita da una Segmento da parte di una entità Treno, è possibile mediante la procedura `Leave` disponibile nell'interfaccia di `Segment`, la quale invocherà la procedura protetta di `Segment_Monitor`, `Leave_Monitor`. Essa ha come prerequisito l'aver avuto accesso al Segmento, ed 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 stabilito da `Train_Queue`.
- Se è il prossimo (vengono confrontati i Descrittori), allora il Treno può abbandonare la risorsa protetta, e di conseguenza viene rimosso dalla coda `Train_Queue`.

  - Nel caso in cui il Treno corrente fosse l'ultimo in transito, allora viene posto `Free=True` e, nel caso vi siano thread in attesa presso la variabile di condizione relativa alla direzione opposta a quella del Treno uscito, allora viene effettuata una `signal` su di essa per ciascun thread in attesa.
  - Se il Treno corrente non era l'ultimo, allora invoca una `signal` sulla veriabile di condizione `Can_Retry_Exit` per ciascun thread in attesa.
- Altrimenti il thread relativo al Treno corrente viene posto in attesa sulla condizione di uscita `Can_Retry_Exit`. Una volta risvegliato da una `signal`, se il Treno con thread correntemente in esecuzione non è il prossimo Treno destinato ad uscire, esso verrà posto nuovamente in attesa su `Can_Retry_Exit`.

La semantica descritta, garantisce ingresso sequenziale a molteplicità $N \geq 1$, 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.

3.3.2.3 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
- Il Treno T uscito dal Segmento S, quindi è stato rimosso dalla coda `Train_Queue` di S.

Il diagramma di sequenza in figura 3.4 mostra le operazioni che portano alla richiesta ordinata di accesso, e all'ingresso presso una 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 politiche di scheduling che saranno adottate, è necessario introdurre un meccanismo che riesca a mantenere l'ordine di

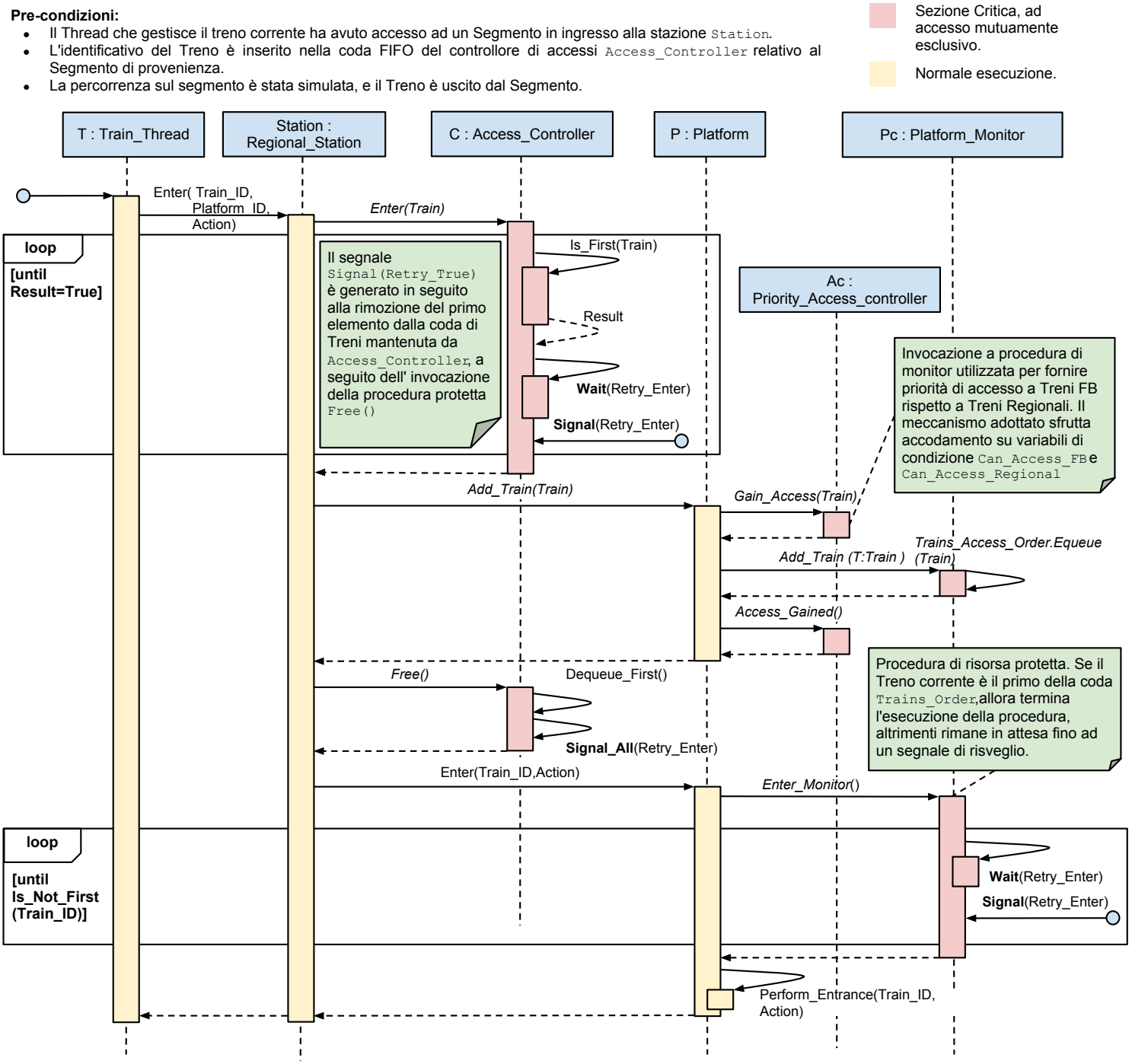

Figura 3.3: Diagramma di Sequenza, operazioni necessarie per l'ingresso in una Piattaforma.

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 *monitor* `Access_Controller` per ciascun Segmento entrante, la quale si occuperà di garantire l'ordine di accesso mantenendo a tale scopo una coda FIFO interna `Trains_Order`. Una volta che il Treno ha guadagnato il permesso di accedere in mutua esclusione alla risorsa protetta di controllo, esso avrà il permesso di interagire con la prossima Piattaforma, in modo concorrente con i Treni provenienti da altri Segmenti entranti. La coda `Trains_Order` sarà aggiornata dal Segmento mediante un'invocazione ad una procedura fornita dall'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. Un esempio di pseudo-codice per il Controllore di Accessi è riportata nel listato 3.5.

Listing 3.5: Controllore di Accessi che garantisce accesso ordinato.

```
monitor Access_Controller
  ...
  procedure Add_Train(T:Train)
  begin
    // Accoda il Treno in ingresso nella coda
    // FIFO Trains_Order.
    Trains_Order.Enqueue(T);
```

```
    end;

    procedure Enter(T:Train)
    begin
      // Controlla il primo elemento della coda.
      // Se coincide con il Treno corrente, il
      // thread puo' proseguire la propria
      // esecuzione.
      while(T.Id/=Trains_Order.First_Element())
      loop
        // Attesa su variabile di condizione,
        // in attesa di poter ritentare
        // l'accesso.
        wait(Retry_Enter);
      end loop;
    end;

    procedure Leave()
    begin
      // Risveglia tutti i thread in attesa
      // sulla variabile di condizione
      // Retry_Enter, in modo ordianto.
      signal_all(Retry_Enter);
    end;
  end monitor;
```

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 e scelta, è simile alla soluzione precedente, ma a differenza di essa crea una popolazione dinamica di oggetti `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 una 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 delle strutture di controllo sarà limitata al massimo al numero di Segmenti in ingresso.

Il controllore di accessi (`Access_Controller`) ha quindi il compito di far

passare solamente il Treno (il thread che lo esegue) che effettivamente è primo nella coda di ordinamento. Siano `Access` e `Free` rispettivamente le due procedure esposte da `Access_Controller`, allora l'accesso alla Piattaforma verrà regolato nel seguente modo:

- Il Treno T usa `Access` per poter accedere alla prossima Piattaforma.
- Se T è effettivamente il primo della Coda allora
  - prosegue con l'accesso alla Piattaforma.
  - libera il controllore di accessi `Access_Controller` con `Free`.
- Altrimenti viene messo in attesa del proprio turno su una condizione (`Retry_Enter`). I Treni in attesa sulla coda FIFO associata a questa variabile di condizione, verranno risvegliati ogni volta che un Treno effettuerà un accesso e una successiva chiamata a `Free` (la quale invocherà `signal(Retry_Enter)` per ciascun thread in attesa).

Ingresso in una Piattaforma

Dopo aver superato la barriera di controllo d'accesso alla Stazione, rappresentata dalla struttura `Access_Controller`, invocando la procedura protetta `Enter`, ciascun Treno può procedere nel tentativo di ottenere l'accesso alla Piattaforma desiderata. Tale accesso avverrà quindi:

- In maniera sequenziale tra tutti i Treni provenienti dallo stesso segmento.
- In maniera concorrente tra tutti i Treni provenienti da Segmenti diversi, fornendo precedenza di accesso ai Treni di tipo `FB`.

Si rende quindi evidente la necessità di garantire l'ordine di esecuzione tra tutti i thread provenienti da uno stesso segmento, indipendentemente dalle politiche di scheduling che verranno adottate. Non è inoltre desiderabile che il thread corrente effettui l'accesso prima di aver rilasciato il controllore di accessi per il Segmento di porvenienza. É quindi necessario, al rientro della procedura `Access` del controllore d'accessi, *prenotare* l'accesso alla Piattaforma per poi liberare la risorsa `Access_Controller` prima di proseguire con l'ingresso vero e proprio.

Una Piattaforma è modellata quindi come una struttura dati che contiene una risorsa protetta da *monitor* `Platform_Monitor` per la regolazione dell'ordine di ingresso, e che espone un'interfaccia composta da:

- `Add_Train`, aggiunge il Descrittore del Treno passato come parametro nella coda FIFO `Trains_Access_Order` della risorsa protetta `Platform_Monitor`. Questa coda manterrà l'ordine di accesso alla Piattaforma, non solo tra tutti i Treni provenienti dallo stesso Segmento, ma anche tra i Treni provenienti da Segmenti diversi.
- `Enter_Platform`, per eseguire l'accesso.
- `Leave_Platform`, per lasciare la piattaforma.

Per garatire inoltre che l'ordine di ingresso dei Treni sancito dalla coda `Trains_Access_Order` sia tale da fornire accesso preferenziale ai Treni di tipo `FB`, viene utilizzato un oggetto `Priority_Access_Handler` come per l'accesso al Segmento (si veda il listato 3.2), e di conseguenza il comportamento previsto per la procedura protetta `Add_Train` sarà quello riportato nel listato 3.6.

Listing 3.6: Esempio di procedure `Add_Train` per l'accesso alla Piattaforma.

```
...
// Procedura Add_Train per l'aggiunta
// di un descrittore alla coda interna
// alla risorsa protetta Platform_Monitor.
procedure Add_Train(T:Train)
begin
  Priority_Access.Gain_Access(T);
  // Viene stabilito l'ordine prenotando la
  // posizione nella coda Trains_Access_Order del
  // monitor a protezione della piattaforma.
  Platform_Monitor.Trains_Access_Order.Enqueue(T);
  Priority_Access.Access_Gained();
end;
```

Una volta aggiunto l'identificativo del Treno corrente in `Trains_Access_Order`, può essere liberata la risorsa di controllo (`Access_Controller`) mediante la procedura `Free` ed è quindi possibile richiedere l'accesso alla Piattaforma successiva prevista dal percorso (l'ordine di ingresso sarà rispettato), che avviene mediante la procedura `Enter_Platform` la quale:

- Richiede l'ingresso nella piattaforma tramite la procedura protetta `Enter_Monitor` di `Platform_Monitor`.
- Effettua l'operazione di **Discesa dei Viaggiatori** se previsto dal percorso.

La semantica di accesso ad una Piattaforma P è molto semplice; la procedura di monitor `Enter_Monitor` è la seguente:

- Se il Treno corrente T è il prossimo a poter entrare in P, allora termina l'esecuzione della procedura.
- Se invece T non è il prossimo Treno che può entrare, il Treno viene posto in attesa su variabile di condizione `Retry_Enter`.

Si può notare che la struttura dati di controllo per l'accesso ad una Piattaforma è molto simile a quella utilizzata per l'accesso al Segmento (listato 3.5).

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 precondizione l'accesso alla Piattaforma da parte di un Treno T descritta in precedenza, ed è realizzata dalle seguenti azioni (eseguite dal thread associato a T):

- Viene estratto ciascun Viaggiatore V dalla coda `Arrival_Queue`, e per ognuno 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 devono essere eseguite esternamente alla Sezione critica associata all'ingresso del Treno nella Piattaforma; se così non fosse infatti, essa risulterebbe occupata, e non sarebbe quindi possibile per i thread che sopraggiungono ottenere il blocco esclusivo sulla risorsa per eseguire `Add_Train`.

Uscita da una Piattaforma

L'uscita da una Piattaforma P necessita dei seguenti prerequisiti:

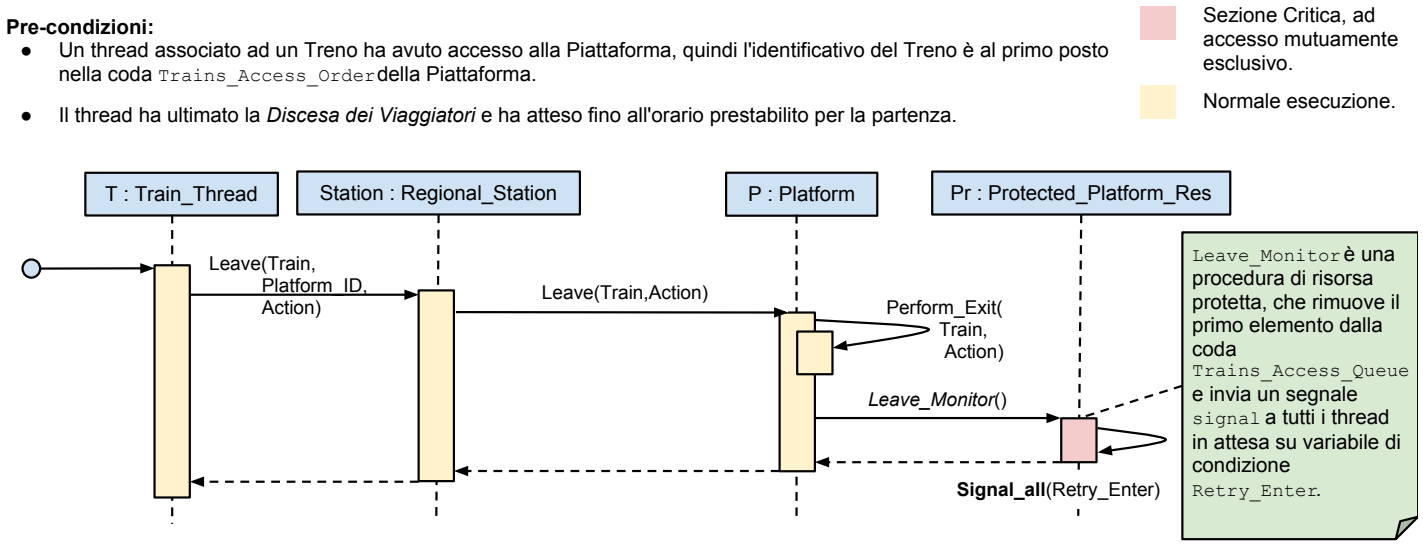

Figura 3.4: Diagramma di Sequenza, operazioni necessarie per l'uscita da una Piattaforma.

- Il thread che gestisce il Treno ha prima effettuato l'ingresso, e quindi il suo identificativo univoco è il primo elemento della coda `Trains_Access_Order` mantenuta dalla risorsa `Platform_Monitor` a protezione della Piattaforma.
- É già stata effettuata la Discesa dei Passeggeri (se prevista dal Percorso del Treno).
- É stato generato l'evento che notifica al Treno l'arrivo dell'orario previsto per la partenza.

A questo punto, le operazioni effettuate per completare la partenza sono:

- Viene effettuata la **Salita dei Viaggiatori** in attesa presso la Piattaforma.
- Il Treno corrente registra il proprio passaggio presso la Piattaforma, memorizzando in una struttura dati apposita `Last_Train_Run` il proprio identificativo univoco e l'identificativo univoco della Corsa corrente `Current_Run_Id`.
- La risorsa viene rilasciata dal Treno in esecuzione mediante la procedura di monitor `Free_Monitor`, la quale provvede a rimuovere il primo elemento della coda `Trains_Access_Order` e, se ci sono

thread in attesa sulla condizione `Retry_Enter`, a risvegliare tali thread.

La **Salita dei Viaggiatori** è simile alla discesa dei Passeggeri presentata al punto precedente. Essa prevede l'esecuzione delle 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 *eseguita* a differenza di quanto avviene nella fase di Discesa dei Passeggeri: in questo modo infatti si avrà la garanzia che tutti i Viaggiatori saliti a bordo del Treno saranno in attesa nella coda `Arrival_Queue` della Piattaforma di Stazione corretta, nel momento in cui il Treno vi accederà durante il proprio viaggio.
  - Se T invece non è il Treno atteso da V, allora viene verificato se il Treno per il quale esso è in attesa non sia già passato, qualora esso sia di tipo `FB`: per effettuare tale controllo viene acceduta la struttura dati `Last_Train_Run` e verificato se la corsa per la quale il Viaggiatore corrente è in attesa del Treno non sia stata già effettuata. In tal caso viene inserita l'operazione `BUY_TICKET` per il Viaggiatore corrente nella coda di `Traveler_Pool`, per permettere l'acquisto di un nuovo Biglietto per raggiungere la destinazione prevista a partire dalla Stazione corrente. In tutti gli altri casi, il Viaggiatore viene re-inserito nella coda `Leaving_Queue`.

3.3.2.4 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 3.3.2.3. Esse hanno quindi un effetto locale alle Regioni (Nodi) in cui vengono eseguite. Per

quanto riguarda invece la Discesa dei Viaggiatori, essa può prevedere un *trasferimento remoto* dei Viaggiatori una volta scesi, nel caso essi continuino il proprio viaggio in una Regione diversa.

Il Passaggio di un Treno da una regione alla successiva può essere schematizzato come segue: siano $G1$ e $G2$ Stazioni di Gateway connesse, che collegano le regioni $R1$ ed $R2$, con $G1 \in R1$, e $G2 \in R2$. 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 3.3.2.3, 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 (*marshalling* dei dati e invio mediante invocazione remota al Nodo di destinazione), 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 e la sua Tabella degli Orari vengono serializzati (*marshalling*), e inviati tramite invocazione remota al Nodo della Regione $G2$ alla quale appartiene la Stazione di Gateway 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 e la tabella degli orari vengono de-serializzati (*unmarshalling*).
  - I dati del Descrittore $D_{T'}$ presenti nella regione $R2$ vengono aggiornati con quelli ricevuti. Stessa cosa per la Tabella degli Orari.

  - Viene aggiunto l'identificativo di T nella coda `Trains_Order` della Piattaforma P.
  - 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 (viene rimosso il primo elemento della coda `Trains_Order`) 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 del 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 mediante la procedura `Leave`.

3.3.3 Creazione di un Biglietto

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.

Di seguito viene riportata una descrizione delle fasi previste dalla soluzione progettata.

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'operazione `CREATE_TICKET` (come descritto precedentemente in sezione 3.3.1.1). Tale operazione, effettua una richiesta di creazione alla biglietteria interna alla stazione di partenza S, la quale richiederà la creazione effettiva alla Biglietteria Regionale.

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 Biglietteria Regionale sono:

- 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 $Path = \{s_1, ..., s_N\}$. 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, ad esempio, 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 $i <= N$:
    - Ottieni i percorsi dei Treni (`Routes`) $R = r_1, ..., r_k$ 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

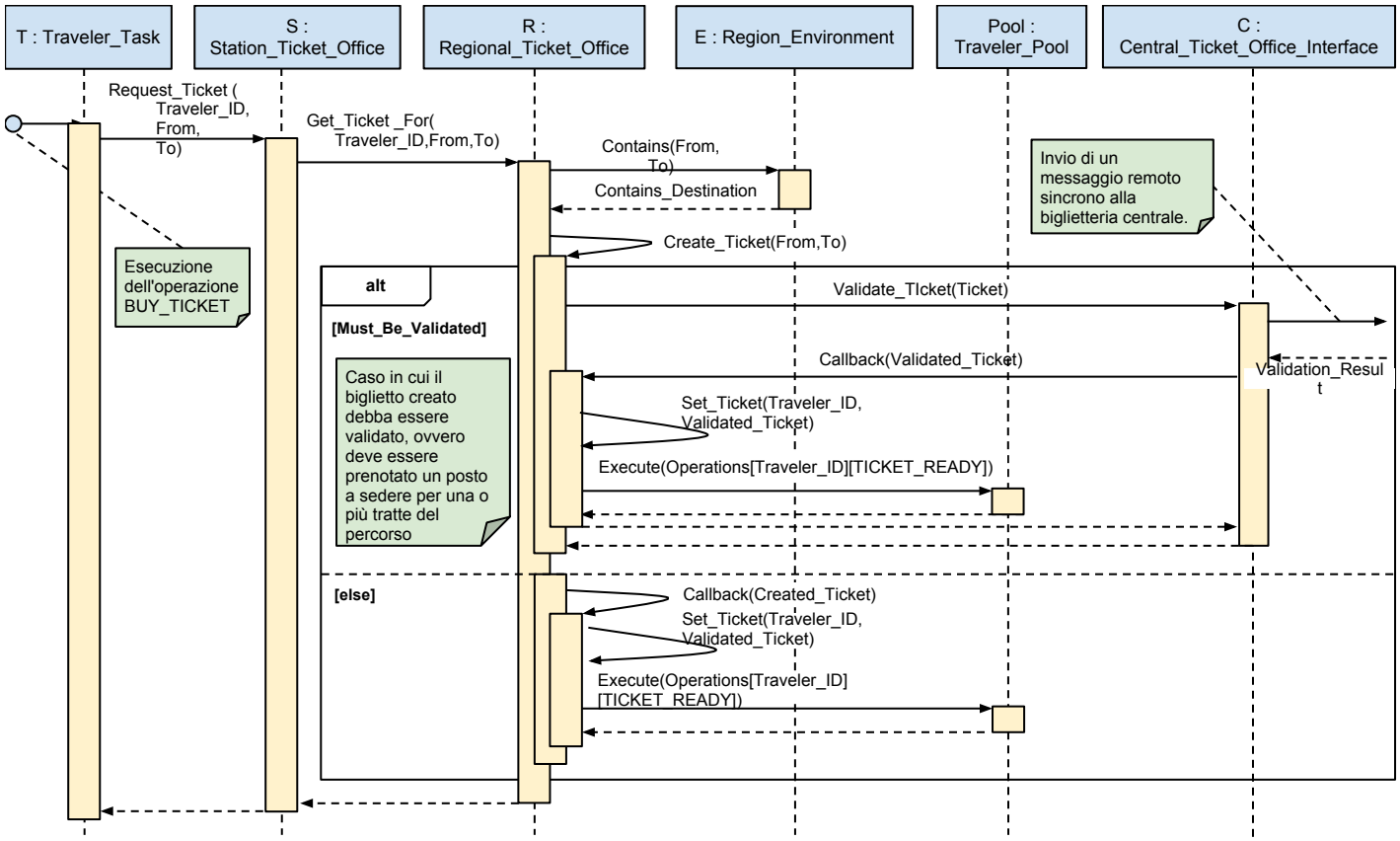

Figura 3.5: Diagramma di Sequenza, creazione di un Ticket per una destinazione *locale* alla regione di partenza.

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.

* Una volta stabilita la corrispondenza di lunghezza massima, viene 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;
- run_number           : identificativo della Corsa
                         per la quale si effettua
                         prenotazione. Inizialmente
                         viene messa a 0 e modificata
                         successivamente se necessario;
- next_region          : regione corrente.
```

  * i viene incrementato di Max_Length, per poter procedere all'individuazione della prossima corrispondenza di lunghezza massima.

– Una volta creato il `Ticket`, allora se esso conterrà almeno una Tratta che prevede l'utilizzo di un Treno a prenotazione (ovvero di tipo `FB`), allora si dovrà procedere alla *Validazione*, descritta nella sezione 3.3.3.4.

– Altrimenti, 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. É opportuno che la richiesta di creazione sia asincrona, in modo tale da permettere al thread che la effettua di non dover attendere attivamente per una risposta, e quindi

di poter eseguire altre operazioni eventualmente presenti nella coda di `Traveler_Pool`.

In figura 3.5 è riportato un diagramma di sequenza che presenta le operazioni svolte per la creazione di un Ticket locale alla Regione dalla quale la richiesta è effettuata.

3.3.3.3 Richiesta di Creazione Remota

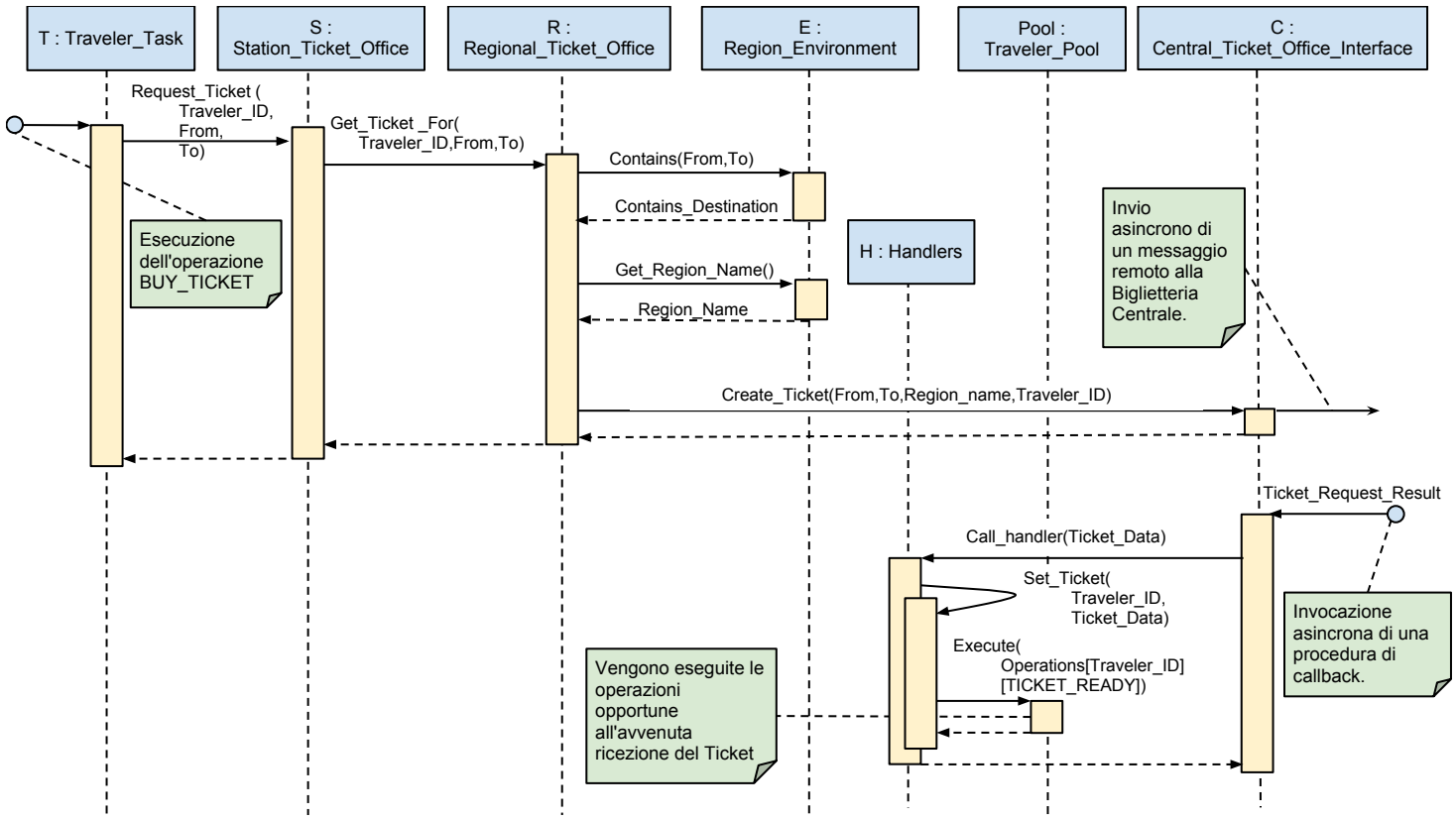

Figura 3.6: Diagramma di Sequenza, creazione di un Ticket per una destinazione *non appartenente* alla regione di partenza.

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 inviato un messaggio remoto, al 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 $R_1, ..., R_n$ di Regioni attraverso le quali costruire il percorso per raggiungere la destinazione (dalla mappa `Links`).
  - Per ciascuna Regione R_i viene inviata una richiesta remota *sincrona* per ottenere un Ticket che collega le varie stazioni di Gateway interne: i Biglietti ottenuti una volta uniti, permettono di raggiungere `To` a partire da `From`. La lista di Ticket ottenuta viene *Validata* in modo tale da prenotare posti a sedere se necessario, per ciascun Biglietto.
  - Il processo di unione dei vari `Ticket` raccolti, avviene fondendo Tappe che usufruiscono dello stesso Treno eliminando se necessario il passaggio per le stazioni di Gateway; il Biglietto risultante infine viene inviato come risposta al Nodo richiedente, presso il quale verrà assegnato al viaggiatore richiedente, per poi inserire l'operazione `TICKET_READY` nella coda di `Traveler_Pool`.

3.3.3.4 Validazione di un Ticket

La Validazione di un Biglietto è necessaria qualora esso preveda l'utilizzo di Treni a prenotazione (ovvero di tipo `FB`). Se infatti anche solo una delle Tappe del percorso di un Treno appartenente a questa categoria che il Biglietto prevede di percorrere, non ha un numero sufficiente di posti a sedere, esso non potrà essere erogato. La Biglietteria Centrale, per ciascun Treno a prenotazione T, mantiene $N > 1$ array di numeri interi, uno per ciascuna Corsa c, rappresentanti il numero di posti liberi a sedere per ciascuna Tappa del Percorso (`Route`) di T. Essa mantiene inoltre l'indice della Corsa correntemente percorsa da T, `Current_Run`, e un identificativo per essa, `Current_Run_Id`. Con questi dati è quindi possibile verificare la possibilità di prenotare un posto a sedere per un insieme di Tratte, e quindi prenotarli. Per

semplicità, ho assunto che la prenotazione avvenga in mutua esclusione e per un'intera lista di Biglietti (caso in cui il percorso sia unione di Ticket raccolti da Regioni diverse dall'algoritmo di creazione introdotto in sezione 3.3.3.3). Per ciascuna Tappa del Biglietto da validare, vengono quindi effettuate le seguenti operazioni:

- Vengono individuate le Tappe del Percorso del Treno T che verranno percorse secondo lo `Stage` corrente del Ticket in esame, siano esse $r_{\cdot j}, ..., r_{\cdot k}$.
- Viene individuata la corsa di riferimento per la prenotazione rc, ovvero quella che permette una possibile prenotazione del tratto composto da $r_{\cdot j}, ..., r_{\cdot k}$ in base all'orario in cui la richiesta viene effettuata alla Biglietteria Centrale.
- Con questi dati, viene verificata la disponibilità per le Tappe da j a k della Corsa rc:
  - Nel caso in cui non vi siano posti a sufficienza (almeno una delle tappe ha 0 posti liberi), la procedura di validazione termina con esito negativo, e viene comunicato un messaggio di errore al richiedente.
  - Se c'è disponibilità, allora vengono memorizzato i, j, rc, per effettuare le modifiche apposite successivamente.

Una volta terminata in maniere positiva la fase di verifica della disponibilità, con i dati memorizzati vengono apportate le modifiche per attuare la prenotazione, ovvero viene decrementato di 1 il numero di posti liberi per le Tappe individuate.

Nella definizione del processo di creazione e validazione dei Ticket, ho vagliato l'ipotesi di distribuire l'informazione relativa alla prenotazione dei Treni sulle varie regioni, in modo tale che ciascuna Biglietteria Regionale mantenesse il numero di posti liberi per una porzione di `Route` di ciascun Treno di tipo `FB` attraversante la Regione. Sebbene tale soluzione avrebbe permesso a ciascuna Biglietteria Regionale di risolvere localmente l'assegnazione di Ticket per destinazione interne alla Regione di appartenenza, essa avrebbe reso troppo complessa la validazione di un Biglietto che copre più Regioni in quanto:

- Le prenotazioni su Regioni differenti si dovrebbero accumulare, e in caso di fallimento della prenotazione si renderebbe necessaria una operazione di rollback per ripristinare i posti prenotati.

- Vi è la possibilità di prenotazioni incompatibili: siano A e B Stazioni appartenenti a due regioni diverse, e sia C Stazione che le collega, tale per cui esiste una Tratta a prenotazione da A a C e da C a B. Siano T_1 e T_2 Viaggiatori, il primo vuole raggiungere B da A, il secondo A da B. Se richiedono un Ticket in maniera concorrente, vi è la possibilità che T_1 occupi l'ultimo posto rimasto per la tratta $A - C$, e che T_2 occupi l'ultimo posto per la Tratta $C - B$. In questo modo nessuno dei due otterrà un Ticket.

3.3.4 Terminazione distribuita

La Terminazione del Sistema progettata, sfrutta la presenza dell'entità di Controllo Centrale, la quale, per ciascun nodo di simulazione, invia un messaggio di terminazione. Internamente a ciascun nodo, la terminazione può avvenire semplicemente inviando un messaggio di terminazione ai pool di thread `Train_Pool` e `Traveler_Pool`, in modo tale che ciascun thread in attesa sulla rispettiva coda di Treno e Operazioni da eseguire, interrompa tale attesa e concluda la propria esecuzione. Una soluzione che si limiti a terminare singolarmente tutti i nodi di esecuzione, tuttavia questo non è sufficiente ne a garantire che tutti i thread terminino la propria esecuzione, ne che lo stato risultante sia consistente. In primo luogo infatti, ci possono essere thread in attesa su una variabile di condizione, la quale può non essere più notificata a causa della terminazione del thread incaricato. É il caso dell'attesa per l'accesso ad una Piattaforma da parte di un Treno: infatti nel momento in cui il thread rappresentante il Treno che occupa la Piattaforma termina la propria esecuzione, nessun'altra entità provvederà ad invocare una `signal` per permettere l'accesso ai thread in attesa. Nonostante lo stato risulti consistente (un Treno occupa una Piattaforma, altri Treni sono in attesa), il sistema non può considerarsi terminato. Risulta quindi opportuno prevedere un meccanismo capace di interrompere l'attesa su condizione, discriminando il caso in cui essa avvenga per terminazione. In tutti gli altri casi in cui si può avere attesa (accesso ad un Segmento), la semantica descritta nelle sezioni precedenti garantisce che i Treni completeranno le operazioni in modo tale da non rimanere bloccati su variabili di condizione (evitando così situazioni di Stallo).

In secondo luogo, ci possono essere problemi legati alle richieste di natura asincrona come la creazione di Ticket, la quale coinvolge la Biglietteria Centrale e i nodi di simulazione: ad esempio, se supponiamo che i nodi di simulazione abbiano ricevuto un messaggio di terminazione e che alcuni di essi abbiano già terminato la propria esecuzione, la Biglietteria Centrale sarà impossibilitata a contattarli, generando un messaggio di errore sia in fase di creazione, sia in fase di consegna del Ticket creato. Questo può portare a situazioni non consistenti, come per esempio la situazione in cui un Viaggiatore in attesa (non attiva poiché il processo è asincrono, e quindi terminabile) di un Ticket non lo riceve per un errore della Biglietteria Centrale nella costruzione del Biglietto a causa dell'impossibilità di contattare nodi già terminati.

Per ovviare a problemi di questo tipo, è opportuno che il processo di terminazione coinvolga anche la Biglietteria Centrale. Una possibile soluzione al problema della terminazione è la seguente: definiamo `Marker` un messaggio

speciale utilizzato per indicare l'inizio della procedura di terminazione.

- Il Controller Centrale inizia la procedura di terminazione e invia un messaggio `Marker` alla Biglietteria Centrale.
- Una volta ricevuto tale messaggio, la Biglietteria Centrale inizia a memorizzare in una coda FIFO tutti i messaggi di richiesta (asincrona) di Creazione di un Biglietto. Essa inoltre, completa le richieste di creazione di un Ticket in corso e memorizza i messaggi in uscita relativi a tali richieste su una coda di output. Infine, la Biglietteria Centrale invia un messaggio di conferma di terminazione delle operazioni asincrone correnti al Controller Centrale.
- Il Controller invia quindi un `Marker` a tutti i nodi di simulazione, i quali:
  - ultimano le operazioni in corso;
  - inviano un messaggio di `Unbind` al Name Server;
  - inviano un messaggio di risposta al Controller Centrale;
  - infine terminano.
- Il Controller una volta ottenuta risposta da tutti i nodi di simulazione, invia un messaggio di terminazione al Server dei Nomi, e un nuovo `Marker` alla Biglietteria Centrale, la quale memorizzato lo stato delle code di input e output termina.
- Il Controller Centrale termina la propria esecuzione.

La soluzione presentata permette quindi la terminazione delle entità distribuite in uno stato consistente.

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. Di seguito verranno presentati i linguaggi di programmazione adottati e le tecnologie scelte.

4.1 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.

Internamente alle componenti distribuite, la concorrenza è stata gestita adottando il modello di concorrenza ad *Attori* fornito nativamente dal linguaggio.

4.2 Ada

Il linguaggio Ada è stato utilizzato per realizzare la componente di simulazione presente in ciascuna Regione. Ho preferito infatti il modello di concorrenza

fornito da questo linguaggio, il quale mi è sembrato più adatto per rappresentare le interazioni tra le entità introdotte al capitolo precedente, rispetto ai modelli offerti da altri linguaggi. In prima battuta, ho valutato l'utilizzo del modello di concorrenza ad *Attori* di Scala; tale modello avrebbe garantito una separazione naturale tra Attore e Thread che lo esegue, e quindi fornito un meccanismo di linguaggio scalabile per l'esecuzione delle entità Viaggiatore e Treno, da utilizzare in sostituzione a quello descritto 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.

4.3 Yami4

L'interazione tra componenti remote del sistema, realizzate con tecnologie eterogenee, è stata possibile utilizzando il middleware per lo scambio di messaggi *Yami4* (`http://www.inspirel.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 versatile, 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 *Agente* (oggetto fornito da *Yami4* che permette l'invio e la ricezione di messaggi remoti), sia per quanto riguarda l'invio che la ricezione di messaggi remoti, in ascolto ad un singolo indirizzo (adottando il protocollo TCP). Per ciascun *Agente*, è possibile infatti definire:

- `Object`: un oggetto remoto raggiungibile ad un dato indirizzo, identificato univocamente da una stringa.
- `Service`: 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 servizio richiesto. 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, è quello fornito da *Yami4*, e quindi lo scambio di dati in formato stringa chiave-valore; inoltre per l'invio di strutture dati complesse come Descrittori di Treni o Passeggeri, e Biglietti, ho utilizzo lo standard *JSON*. Per la codifica e decodifica di stringhe *JSON* mi sono servito di due librerie:

- 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.

Il formato *JSON* è utilizzato anche per la definizione dei file di configurazione che descrivono le entità del sistema.

Capitolo 5

Prototipo Realizzato

Di seguito verrà fornita una descrizione delle componenti principali che costituiscono il prototipo realizzato, e in particolare verrà descritto come le soluzioni progettate sono state realizzate nella pratica. Dove necessario, verranno riportati listati di codice sorgente a supporto della descrizione.

5.1 Simulazione

L'architettura di massima adottata per la realizzazione del core di simulazione, è visibile nel diagramma delle classi in figura 5.1 nel quale, per brevità, sono stati riportati solo le informazioni più significative, omettendo per esempio i package nei quali sono contenute gli oggetti creati duarante l'esecuzione.

Viene ora fornita una breve descrizione delle varie componenti che sono state utilizzate, procedendo per Package. Nelle sezioni seguenti si utilizzeranno le nozioni di *Task* e *Tipo protetto* definite dal linguaggio *Ada*, utilizzato per l'implementazione, e verrà mostrato come le soluzioni presentate nel capitolo 3 sono state tradotte con strumenti di linguaggio.

5.1.1 Message_Agent

Il package `Message_Agent` fornisce un'interfaccia per l'invio di messaggi remoti. Esso contiene una classe singleton `Message_Agent_Type`, la quale possiede i seguenti campi dato:

- `Client_Agent:YAMI.Agents.Agent_Access`
  Campo dati privato di tipo `YAMI.Agents.Agent_Access` che contiene una istanza di Agente fornito dalla libreria *Yami4*, e che viene utilizzato per l'invio e la ricezione di messaggi remoti.

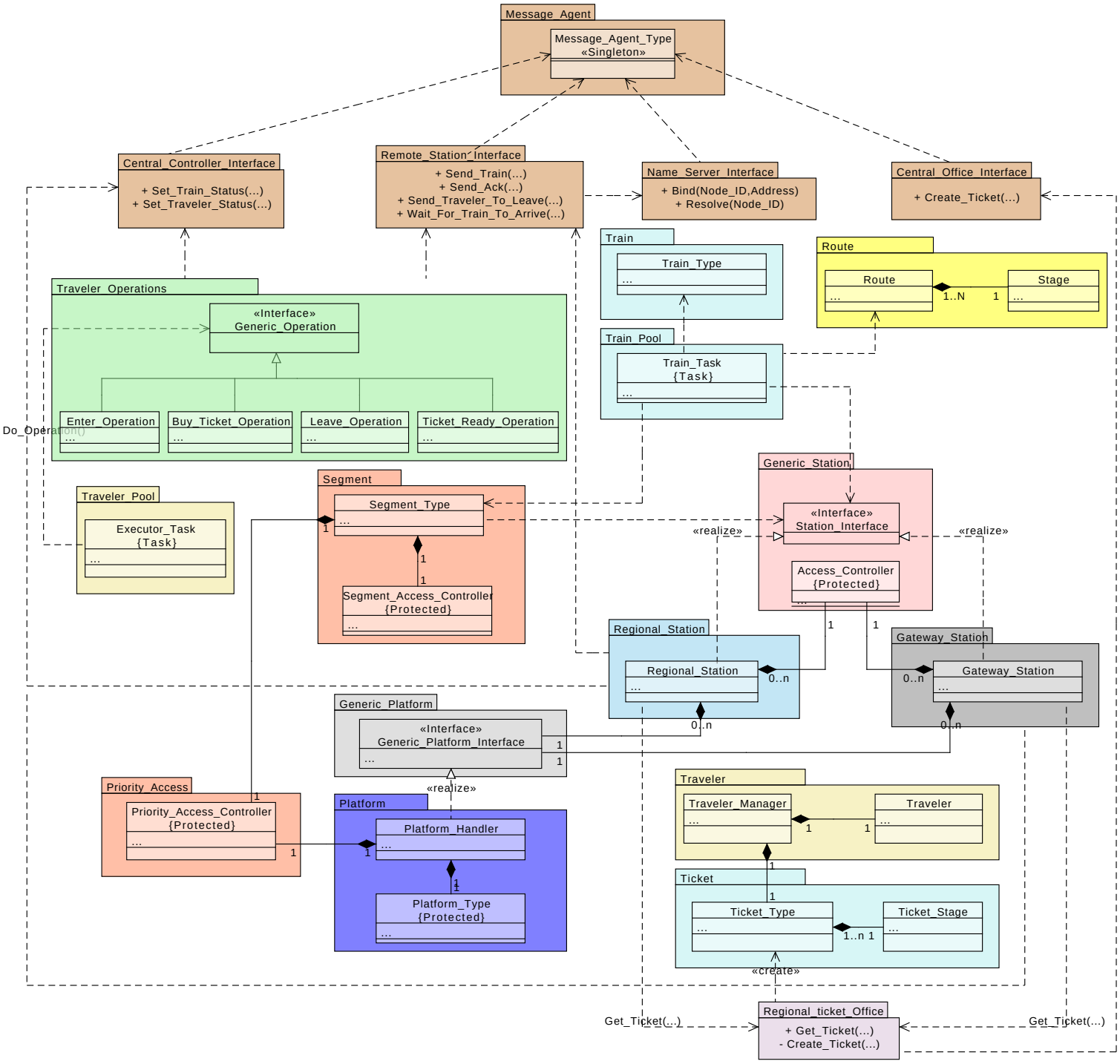

Figura 5.1: Diagramma delle classi che illustra le componenti e le relazioni più significative che coinvolgono la simulazione.

- `Handlers_Map:Map`
  Hash-map privata, la quale associa a chiavi di tipo `String`, valori di tipo *riferimento a procedura*; essa viene utilizzata per associare a ciascun servizio offerto dall'oggetto `Client_Agent` un handler per la sua gestione.

Essa offre inoltre i seguenti metodi:

+ `Listen_To(`
  `Server_Address:String)`
  Indica all'oggetto `Client_Agent` di rimanere in ascolto presso l'indirizzo `Server_Address`, attraverso il quale riceverà tutti i messaggi destinati al nodo. Nella fase di registrazione dell'oggetto remoto, viene definito un handler per la ricezione dei messaggi, il quale effettua il dispatching di ciascun messaggio ricevuto invocando la procedura corrispondente al servizio richiesto, definita nella mappa `Handlers_Map`.

+ `Close()`
  Chiude la connessione dell'oggetto `Client_Agent`.

+ `Send_Message(`
  `Destination_Address:String,`
  `Object:String,`
  `Service:String,`
  `Params:YAMI.Parameters.Parameters_Collection,`
  `Callback)`
  Invia un messaggio all'oggetto remoto indicato da `Object` all'indirizzo `Destination_Address`, richiedendo il servizio `Service`, e con i parametri indicati da `Params`. Se non nulla, la funzione di callback `Callback` viene invocata alla ricezione del messaggio di risposta, altrimenti quest'ultimo viene ignorato.

+ `Send_One_Way(`
  `Destination_Address : String,`
  `Object : String,`
  `Service : String,`
  `Params : YAMI.Parameters.Parameters_Collection)`
  Metodo simile a `Send_Message`, che non rimane in attesa della risposta al messaggio inviato.

Nessun meccanismo di serializzazione delle richieste è stato adottato, in quanto esso è già operato dagli oggetti della libreria *Yami4*.

5.1.2 Central_Controller_Interface

Il package `Central_Controller_Interface` fornisce un'interfaccia per permettere alle entità della simulazione di inviare un evento di notifica all'oggetto remoto `central_controller`. A tal proposito sono messi a disposizione procedure che le quali effettuano il marshalling dei dati in ingresso (in formato *JSON*) e inviano un messaggio remoto mediante il metodo `Send_One_Way` offerto dall'unica istanza di `Message_Agent_Type`.

5.1.3 Central_Office_Interface

Il package `Central_Office_Interface` fornisce un'interfaccia per permettere la comunicazione con la Biglietteria Centrale, mediante l'invio di messaggi all'oggetto remoto `central_ticket_server`.

```
+ Create_Ticket(
    From :  String,
    To :  String,
    Traveler_Index:  Integer)
```

Richiede la creazione di un Biglietto inviando un messaggio remoto mediante il metodo `Send_Message` offerto dall'unica istanza di `Message_Agent_Type`. Non viene specificata una procedura di callback, in quanto il risultato verrà inviato dalla Biglietteria Centrale una volta calcolato il Biglietto.

```
+ Validate(
    The_Ticket:Ticket,
    Callback:access procedure(The_Ticket:Ticket,Response:Boolean))
```

Richiede la validazione di un Biglietto inviando un messaggio remoto mediante il metodo `Send_Message` offerto dall'unica istanza di `Message_Agent_Type`. La richiesta è sincrona, e la risposta viene quindi elaborata e viene invocata la procedura `Callback` passando come parametri i risultati estratti.

```
+ Update_Run(
    Route_Index:Integer,
    Current_Run:Integer,
    Callback:access procedure(...))
```

Invia un messaggio remoto per comunicare l'aggiornamento della corsa corrente `Current_Run` per il percorso `Route_Index`. Nel caso in

cui la corsa sia l'ultima delle N mantenute, viene creata una nuova Time Table per il percorso `Route_Index` dalla Biglietteria Centrale che viene restituito al chiamante. Nel caso in cui essa venga ricevuta (in formato *JSON*), viene effettuato l'unmarshalling e creata l'istanza corrispondente di `Time_Table_Type` da sostituire a quella corrente.

+ `Load_Time_Tables(`
  `Callback:access procedure(Table:Time_Table_Array_Ref))`
  Invocazione di un messaggio remoto per richiedere l'insieme delle Time Tables per tutti i percorsi.

5.1.4 Name_Server_Interface

Package che fornisce un'interfaccia remota per la comunicazione con il Server dei Nomi che mantiene la lista delle Regioni di simulazione. Esso fornisce le seguenti procedure:

+ `Bind(`
  `Name_Server : String,`
  `Node_Name : String,`
  `Address : String)`

  Permette di registrare presso il Server dei Nomi che l'entità remota `Node_Name` è disponibile alla locazione indicata da `Address`.

+ `Resolve(`
  `Name_Server : String,`
  `Node_Name : String),`
  `Callback : access procedure(Result:String)`

  Permette di richiedere al Server dei Nomi la risoluzione della locazione alla quale si trova l'entità `Node_Name`. Una volta che la risposta è disponibile, viene invocata la procedura di callback `Callback`.

+ `Remove(`
  `Name_Server:String,`
  `Node_Name:String)`

  Effettua una richiesta di cancellazione del Nodo identificato da `Node_Name` al Server dei Nomi.

Per tutte le procedure elencate, viene inviato un messaggio all'oggetto remoto identificato da `name_server`, mediante il metodo `Send_Message` di `Message_Agent_Type`. Alla procedura `Resolve` inoltre, viene passata una procedura di callback che estrae il campo `address` dal messaggio di ritorno e lo passa all'invocazione di `Callback`. Presso il package viene mantenuta una hash-map, la quale permette di memorizzare le destinazioni risolte, in modo da limitare l'invio di messaggi remoti.

5.1.5 Remote_Station_Interface

Il package `Remote_Station_Interface` espone un insieme di procedure che permettono l'invio di messaggi remoti tra i Nodi che cooperano alla simulazione, che saranno diretti agli oggetti remoti `message_handler`.

```
+ Send_Train(
    Train_Descriptor_Index:Integer,
    Station:Integer,
    Platform:Integer,
    Next_Node_Name:String)
```

Tale procedura permette l'invio di un messaggio remoto al Nodo identificato da `Next_Node_Name` per effettuare il trasferimento del Treno indicato da `Train_Descriptor_Index` presso la Stazione `Station`, Piattaforma `Platform`.

```
+ Train_Left_Message(
    Train_Descriptor_Index:Integer,
    Station:Integer,
    Platform:Integer,
    Next_Node_Name:String)
```

Questa procedura permette l'invio di un messaggio remoto al Nodo identificato da `Next_Node_Name` per indicare la partenza del Treno `Train_Descriptor_Index` dalla Piattaforma `Platform` della Stazione `Station`.

```
+ Send_Traveler_To_Leave(
    Traveler_Index:Integer,
    Train_ID:Integer,
    Station:Integer,
    Platform:Integer,
```

```
Node_Name:String)
```

Permette l'invio di un messaggio remoto al Nodo identificato da `Node_Name` per aggiungere il Viaggiatore `Traveler_Index` in attesa presso la Piattaforma `Platform` della Stazione `Station` dell'arrivo del Treno `Train_ID` per poter lasciare tale Stazione.

```
+ Wait_For_Train_To_Arrive(
    Next_Station:Integer,
    Traveler_Manager_Index:Integer,
    Train_ID:Integer,
    Destination_Platform_Index:Integer,
    Next_Region:String)
```

Permette l'invio di un messaggio remoto al Nodo identificato da `Node_Name` per aggiungere il Viaggiatore `Traveler_Index` in attesa presso la Piattaforma `Platform` della Stazione `Station` dell'arrivo del Treno `Train_ID` per poter arrivare presso tale Stazione.

5.1.6 Queue

Il package `Queue` contiene la definizione di alcuni tipi di code utilizzati nell'intera simulazione:

`Terminable_Queue`

Tipo *protetto* costruito come *wrapper* della coda standard offerta dal package `Ada.Containers.Unbounded_Synchronized_Queues`, e che permette di interrompere l'attesa sulla guardia Booleana definita per l'*entry* `Dequeue`, la quale rimane chiusa nel caso in cui non vi siano più elementi al suo interno. Esso e definisce l'*entry*

```
Dequeue(Element:out Element_Type,Terminated:out Boolean)
```

con guardia Booleana: `Termination or Q.Current_Use > 0`, dove `Q` è una coda fornita dalle librerie standard di linguaggio, `Current_Use` è una funzione che restituisce il numero di elementi all'interno della coda, e `Termination` è un campo dati Booleano della risorsa protetta. In questo modo nel caso in cui il valore di `Termination` sia `True` l'attesa su coda viene interrotta, e viene restituito al chiamante il valore `True` attraverso il parametro passato per riferimento `Terminated`, altrimenti

la coda restituisce anche il valore del primo elemento rimosso dalla coda.

Per poter attribuire al campo dati `Termination` il valore `True`, viene fornita la procedura protetta `Stop`.

`Limited_Simple_Queue`
: Tipo di coda ad accesso non sincronizzato di dimensione limitata, realizzato mediante un array di elementi, e che fornisce un'interfaccia composta dai metodi:

  - `Enqueue(Element:Element_Type)` per accodare un nuovo elemento;
  - `Dequeue(Element:out Element_Type)` per rimuovere l'elemento dalla testa della coda;
  - `Get(Index:Integer)` per ottenere il valore dell'elemento nella posizione `Index`;

`Unlimited_Simple_Queue`
: Tipo di coda ad accesso non sincronizzato di dimensione illimitata, realizzato mediante un oggetto di tipo `Vector` definito dalle librerie standard `Ada.Containers.Vectors`. Esso presenta un'interfaccia del tutto simile a quella offerta dal tipo `Limited_Simple_Queue`, alla quale aggiunge il metodo `Is_Empty : Boolean` che indica se la coda è vuota.

5.1.7 Generic_Operation_Interface

Package che contiene la definizione di una interfaccia `Operation_Interface`, la quale espone un unico metodo `Do_Operation()`. Essa rappresenta una generica operazione. Viene definito inoltre un tipo puntatore ad operazione generica `Any_Operation`.

5.1.8 Traveler_Pool

Il package Traveler_Pool realizza il meccanismo per l'esecuzione delle entità Viaggiatore descritto in sezione 3.2.2. Esso mantiene

- una coda `Operations_Queue` di puntatori di tipo `Any_Operation` a operazioni. Tale coda è di tipo `Terminable_Queue`, definito nel package `Queue`.

- la definizione di un tipo record `Traveler_Pool_Type` contenente un array di oggetti task `Executor`, di dimensione fissata in fase di creazione. Ciascun task di tipo `Executor` eseguirà semplici operazioni ciclicamente:
  - Estrae il primo elemento dalla coda `Operations_Queue` mediante il metodo `Dequeue` da essa offerto.
  - Nel caso in cui il valore del parametro `Terminated` passato per riferimento abbia il valore `True`, allora viene interrotto il ciclo di operazioni.
  - Altrimenti, viene invocato il metodo `Do_Operation` sul puntatore ad operazione estratto.

5.1.9 Ticket

Package che contiene la definizione di un Biglietto. Esso definisce infatti il tipo record `Ticket_Type`, il quale è composto da un campo intero `Next_Stage` che indica la tappa corrente del percorso descritto dal Biglietto, e un puntatore ad un array di Tappe, ovvero di oggetti di tipo `Ticket_Stage`. Quest'ultimi mantengono le seguenti informazioni:

- `Start_Station` : Indice della Stazione di partenza;
- `Next_Station` : Indice della prossima Stazione;
- `Train_ID` : Identificativo del Treno da utilizzare per raggiungere la stazione `Next_Station`;
- `Start_Platform_Index` : Indice della Piattaforma di partenza;
- `Destination_Platform_Index` : Indice della Piattaforma di destinazione;
- `Region` : Nome della regione nella quale si colloca la stazione `Next_Station`;

Il package fornisce inoltre le funzioni necessarie per effettuare marshalling e unmarshalling degli oggetti di tipo `Ticket_Type` in/dal formato *JSON*.

5.1.10 Traveler

Package che contiene la definizione del tipo rappresentante un Viaggiatore. In esso infatti viene definito il tipo record `Traveler_Manager`, formato dai campi:

- `Next_Operation` : Indice della prossima operazione da eseguire;
- `Destination` : Nome della stazione di destinazione;
- `Start_Station` : Nome della stazione di partenza;
- `Start_Region` : Regione di partenza;
- `Traveler` : Campo di tipo record che contiene alcuni dati relativi al Viaggiatore, come nome e cognome.
- `Ticket` : Riferimento ad un oggetto di tipo `Ticket_Type`.

Il package `Traveler` contiene inoltre le funzioni necessarie per effettuare marshalling e unmarshalling secondo il formato *JSON*.

5.1.11 Regional_Ticket_Office

Il package `Regional_Ticket_Office` mantiene una tabella `Paths`, la quale per ciascuna Stazione della Regione corrente, definisce i percorsi più brevi per raggiungere ciascuna altra destinazione nella Regione. Esso inoltre espone la seguente interfaccia:

`+ Create_Ticket(From:String,To:String) : Ticket_Type`
Crea un istanza di oggetto `Ticket_Type` che rappresenta un biglietto per raggiungere la destinazione `To` a partire da `From` nella regione corrente. Essa realizza l'algoritmo di creazione di un Biglietto descritto in sezione 3.3.3.2.

`+ Get_Ticket (Traveler_Index:Integer,From:String,To:String)`
La procedure si occupa dell'effettiva creazione del Biglietto (oggetto di tipo `Ticket_Type`). Tale procedura effettua un controllo: se le Stazioni `From` e `To` sono contenute all'interno della Regione corrente, allora procede alla creazione del Biglietto vero e proprio attraverso la funzione `Create_Ticket`, che viene quindi assegnato al Viaggiatore di indice `Traveler_Index`; successivamente viene inserita l'operazione `TICKET_READY` nella coda di operazioni di `Traveler_Pool`. Se invece la Stazione di destinazione `To` non è contenuta nella Regione corrente, allora viene richiesta la creazione del Biglietto alla Biglietteria Centrale attraverso l'interfaccia `Central_Office_Interface`.

Il package `Regional_Ticket_Office` contiene inoltre la procedura `Init_Path_Map` utilizzata per caricare la mappa contenente i percorsi più brevi, usata dall'algoritmo di creazione del Biglietto.

5.1.12 Route

Il package `Route` contiene la definizione del tipo `Route_Type`, array di oggetti di tipo `Stage`. Quest'ultimo è un tipo *record* che si compone dei seguenti campi dato:

- `Start_Station`: Indice della Stazione di partenza.
- `Next_Station`: Indice della Prossima Stazione.
- `Start_Platform`: Indice della Piattaforma di partenza.
- `Platform_Index`: Indice della prossima Piattaforma.
- `Node_Name`: Identificativo del Nodo sul quale si trova la prossima Stazione.
- `Next_Segment`: Indice del prossimo Segmento da percorrere.
- `Leave_Action`: Azione da intraprendere alla partenza dalla Stazione.
- `Enter_Action`: Azione da intraprendere all'ingresso alla prossima Stazione.

Esso fornisce inoltre le procedure necessarie alla conversione da *JSON* a oggetto `Route_Type` e viceversa. Tutti gli oggetti Route sono mantenuti in un array nel package `Routes`.

5.1.13 Time_Table

Il package `Time_Table` contiene la definizione del tipo `Time_Table_Type`, che rappresenta una tabella di orari per una specifica `Route`. Esso è un *record* con i seguenti campi:

- `Route_Index`: Indice del Percorso per il quale la Tabella è definita.
- `Table`: Matrice di NxM oggetti di tipo `Ada.Calendar.Time`, dove N è il numero di Corse mantenute, e M è la lunghezza della `Route` per la quale la tabella è definita.
- `Current_Run`: Indice della Corsa corrente (da 1 a N).
- `Current_Run_Cursor`: Indice utilizzato per scorrere gli orari memorizzati nella Corsa `Current_Run`.
- `Current_Run_Id`: Identificativo univoco della Corsa.

Il package oltre a definire le funzioni necessarie per effettuare la conversione da *JSON* a `Time_Table_Type` e viceversa, definisce la procedura

`Update_Time_Table(Table:access Time_Table_Type)`

la quale aggiorna il valore di `Current_Run_Cursor`. Nel caso in cui la corsa corrente `Current_Run` sia esaurita (`Current_Run_Cursor + 1 >` M) allora `Current_Run` viene incrementato e viene comunicato ciò alla Biglietteria Centrale mediante la procedura `Update_Run` di `Central_Office_Interface`, la quale effettua una richiesta sincrona. L'insieme di tutte le tabelle è mantenuto in un array nel package `Environment`.

5.1.14 Train

Il package `Train` contiene la definizione del tipo `Train_Descriptor`, record contenente tutti i dati che caratterizzano una entità Treno, e alcune funzioni necessarie ad effettuare la conversione da *JSON* a oggetto `Train_Descriptor` e viceversa. I campi principali del record `Train_Descriptor` sono:

- `Id`: identificativo univoco del Treno.
- `Speed`: Velocità di percorrenza Corrente.
- `Max_Speed`: Velocità massima raggiungibile dal Treno.
- `Current_Station` Indice della stazione corrente.
- `Route_Index`: Indice del Percorso assegnato al Treno.
- `Next_Stage`: Prossima tappa del Percorso assegnato.
- `Sits_Number`: Numero di posti a sedere.
- `Occupied_Sits`: Numero di posti occupati correntemente.
- `T_Type`: Indica il tipo di Treno, tra `FB` e `REGIONAL`.
- `Start_Node`: Regione dalla quale il Treno parte.

Tali descrittori sono memorizzati in un array `Trains` presso il package `Trains`, in modo tale da poter essere acceduti in modo diretto dalle varie componenti del sistema.

5.1.15 Train_Pool

Il package `Train_Pool` definisce il pool di Task, che serve ad eseguire le operazioni previste per i Treni. Esso mantiene due code (di tipo `Terminable_Queue`) di valori interi, una per ciascuna categoria di Treno, nelle quali verranno inseriti gli indici dell'array di descrittori `Trains` contenuto nel package `Trains`, e definisce il tipo Task `Train_Executor_Task`, il quale effettua le seguenti operazioni ciclicamente:

- Preleva da una delle due code il prossimo indice dell'array di descrittori, a seconda del discriminante `Type` assegnato al Task in fase di creazione.

```
...
Trains.Queue.Dequeue(
  To_Get      => Current_Descriptor_Index,
  Terminated  => Terminated);
...
```

- Effettua la partenza dalla stazione corrente all'orario previsto dalla Time Table per il percorso assegnato al Treno.

```
...
-- recupera l'orario di partenza dalla Tabella
-- degli orari memorizzata nel package Environment.
delay until Time_To_Wait;
-- Richiesta di uscita alla stazione
Environment.Stations(Start_Station).Leave(
  Descriptor_Index => Current_Descriptor_Index,
  Platform_Index   => Start_Platform,
  -- Azione da intraprendere al momento della
  -- partenza, definita dalla tappa corrente
  -- del percorso
  Action           => Leave_Action);
...
```

- Richiede l'accesso al prossimo Segmento:

```
...
Segments.Segments(Next_Segment).Enter(
  Current_Descriptor_Index,
  Max_Speed,
  Leg_Length);
...
```

- Una volta ottenuto l'accesso, notifica la partenza dalla stazione corrente del Treno al Controller Centrale mediante la procedura `Set_Train_Left_Status` di `Central_Controller_Interface` e simula la percorrenza nel Segmento con il costrutto:

```
...
delay Duration (Time_In_Segment);
...
```

- Richiedi l'uscita dal Segmento.

```
...
Segments.Segments(Next_Segment).Leave(
    Current_Descriptor_Index);
...
```

- Richiede l'accesso alla prossima Stazione.

```
...
Environment.Stations(Next_Station).Enter(...);
...
```

- Passa alla prossima Tappa del Percorso e inserisce l'indice del Descrittore nella coda apposita.

```
...
Trains.Trains(Current_Descriptor_Index).Next_Stage :=
    Trains.Trains(Current_Descriptor_Index).Next_Stage + 1;
...
```

L'esecuzione dei Task viene interrotta quando l'operazione `Dequeue` invocata sulle code di indici di Descrittori di Treni restituisce `Terminated=True`. Per permettere invece l'interruzione delle operazioni in corso nel caso in cui l'accesso alla prossima Stazione comporti il trasferimento remoto del Treno indicato da `Current_Train_Descriptor`, viene utilizzata l'eccezione `Stop_Train_Execution` definita nel package `Remote_Station_Interface`.

5.1.16 Priority_Access

Il package `Priority_Access` contiene la definizione del tipo *protetto* `Priority_Access_Controller`, il quale viene sfruttato per garantire priorità

di accesso ai Treni di tipo `FB` rispetto ai Treni `Regionali` nell'accesso a Segmenti e Piattaforme. La priorità di accesso è ottenuta mediante accodamento dei Task che rappresentano i Treni ingresso presso entries private distinte, a seconda del tipo al quale tali Treni appartengono.

Viene fornita una entry pubblica `Gain_Access(Train_Index:Integer)`, la quale verifica il tipo del Treno di indice `Train_Index`, e se è `FB`, riaccoda il chiamante presso l'entry `Gain_Access_FB`, altrimenti presso `Gain_Access_Regional`, sfruttando il costrutto `requeue`. Queste due entry sono regolate da guardie booleane differenti. In particolare, `Gain_Access_FB` ha guardia booleana `Free`, variabile locale alla risorsa protetta che indica lo stato di occupazione della risorsa, mentre
`Gain_Access_Regional` ha come guardia booleana l'espressione `Free and Gain_Access_FB'Count=0`, ovvero un Task in attesa presso questa entry potrà eseguire se e soltanto se la risorsa sarà libera e se non vi sono Task in attesa presso la coda `Gain_Access_FB`. Entrambe le entry impostano lo stato `Free` a `False`. Una volta che un Task ha ottenuto il permesso di procedere, esso potrà eseguire operazioni in mutua esclusione rispetto a tutti i Task che hanno richiesto l'accesso con `Gain_Access`, fino all'invocazione successiva della procedura `Access_Gained` che ristabilisce il valore di `Free` a `True`. Inoltre, l'esecuzione al di fuori della risorsa protetta permette ai Task in attesa di accedervi e di venire accodati in modo tale da poter poi essere effettivamente attribuita preferenza d'accesso. Il listato 5.1 riporta la realizzazione della struttura presentata.

Listing 5.1: Realizzazione di `Priority_Access_Controller` per permettere esecuzione preferenziale a Task rappresentanti treni di tipo `FB`

```
protected body Priority_Access_Controller is

  entry Gain_Access(
    Train_Index : in Positive) when True is
  begin
    -- In base al tipo del Treno corrente, riaccoda
    -- il chiamante presso la entry corretta.
    if Trains.Trains(Train_Index).T_Type = Train.FB then
      requeue Gain_Access_FB;
    else
      requeue Gain_Access_Regional;
    end if;
  end Gain_Access;

  entry Gain_Access_FB(
    Train_Index : in Positive) when Free is
```

```
  begin
    Free := False;
  end Gain_Access_Fb;

  entry Gain_Access_Regional(
    Train_Index : in Positive)
  when Free and Gain_Access_FB'Count = 0 is
  begin
    Free := False;
  end Gain_Access_Regional;

  procedure Access_Gained is
  begin
    Free := True;
  end Access_Gained;

end Priority_Access_Controller;
```

5.1.17 Segment

Il package `Segment` contiene la definizione del tipo *tagged* `Segment_Type` che rappresenta il segmento di congiunzione tra due Stazioni introdotto in sezione 3.2.1. Esso contiene due strutture dati:

- `Access_Controller`, oggetto di tipo `Priority_Access_Controller` introdotto nella sezione 5.1.16;
- `Segment_Monitor`, oggetto di tipo `Segment_Access_Controller`.

`Segment_Access_Controller` è un tipo *protetto*, che serve a realizzare il protocollo di accesso multiplo al Segmento. Esso contiene i seguenti campi dato:

- `Id`: Identificativo univoco del Segmento;
- `Segment_Max_Speed`: Velocità massima di percorrenza del Segmento;
- `Current_Max_Speed`: Velocità massima alla quale i Treni percorrono il Segmento;
- `Free`: Indica lo stato di occupazione del Segmento, se `True` il Segmento è da considerarsi occupato, altrimenti libero.
- `Segment_Length`: Lunghezza del Segmento;

- `Current_Direction`: Direzione di percorrenza corrente;
- `Running_Trains`: Coda di tipo `Unlimited_Simple_Queue` che contiene gli identificativi dei Treni in transito;
- `Trains_Number`: Mantiene il numero di Treni attualmente in transito;
- `First_End,Second_End`: Stazioni che sono collegate dal Segmento;
- `Can_Enter_First_End, Can_Enter_Second_End`: Variabili booleane utilizzati come guardie per le enties `Retry_First_End` e `Retry_Second_End`.
- `Enter_Retry_Num`: Numero di Task in attesa per poter accedere alla risorsa protetta.
- `Exit_Retry_Num`: Numero di Task in attesa per poter liberare a risorsa protetta.
- `Train_Entered_Per_Direction`: Numero di Treni transitati per la direzione corrente.
- `Can_Retry_Leave`: Variabile booleana utilizzata come guardia per regolare l'*entry* `Retry_Leave`.
- `Trains_Order`: Coda di tipo `Unlimited_Simple_Queue` che mantiene l'ordine di accesso dei Treni.
- `Retry_Num`: Variabile intera utilizzata per memorizzare il numero di Task in attesa su di effettuare l'accesso.
- `Can_Retry`: Variabile booleana utilizzata come guardia per l'*entry* `Retry_Enter`.

Il tipo `Segment_Type` fornisce un'interfaccia pubblica accessibile ai Task `Train_Executor_Task` per regolare *ingresso* e *uscita* nel/dal Segmento rappresentato, come descritto nella sezione . Di seguito viene riportata una descrizione dettagliata della sua realizzazione.

5.1.17.1 Ingresso nel Segmento

L'ingresso è realizzato mediate l'utilizzo di un metodo pubblico `Enter`, la cui definizione è riportata nel listato 5.2.

Listing 5.2: Metodo `Enter` del tipo `Segment_Type`

```
procedure Enter(
  This        : access Segment_Type;
```

```
    To_Add       : in     Positive;
    Max_Speed    :    out Positive;
    Leg_Length   :  out Positive) is
  begin
    -- Per prima cosa utilizza l'oggetto
    -- Access_Controller per fornire
    -- priorita' di accesso ai Treni
    -- di tipo FB.
    This.Access_Controller.Gain_Access(To_Add);
    -- Una volta ottenuto l'accesso, viene
    -- aggiunto l'identificativo del Treno
    -- alla coda interna a Segment_Monitor,
    -- per definire l'ordine di esecuzione.
    This.Segment_Monitor.Add_Train(To_Add);
    -- Libera la risorsa Access_Controller
    This.Access_Controller.Access_Gained;
    -- Richiesta di accesso al Segmento,
    -- con Enter.
    This.Segment_Monitor.Enter(
      To_Add,
      Max_Speed,
      Leg_Length);
  end Enter;
```

Essa invoca per prima cosa l'*entry* della risorsa protetta `Access_Controller`, in modo tale da ottenere accesso preferenziale, in base alla tipologia ti Treno. Una volta superata la barriere rappresentata dalla *entry* `Gain_Access`, viene aggiunto l'identificativo del Treno corrente nella coda interna alla risorsa protetta `Segment_Monitor`, e quindi liberata la risorsa `Access_Controller` con `Access_Gained`. A questo punto, il Treno avrà un ordine assegnato e potrà quindi richiedere l'accesso vero e proprio, utilizzando l'*entry* pubblica `Enter` fornita da `Segment_Monitor`, la quale, con l'utilizzo dell'*entry* privata `Retry`, consente l'esecuzione ordinata dei Task, secondo l'ordine sancito da `Trains_Order` (listato 5.3).

Listing 5.3: Meccanismo che garantisce accesso ordinato secondo l'ordine sancito da `Trains_Order`.

```
potected body Segment_Access_Controller is
  ...
  entry Retry(
    To_Add       : in     Positive;
    Max_Speed    :    out Positive;
    Leg_Length   :    out Positive) when Can_Retry is
```

```
  begin
    -- Decremento del numero di Task che ritentano.
    Retry_Num := Retry_Num - 1;
    -- Se tale numero e' diventato 0, allora la
    -- guardia viene richiusa.
    if Retry_Num = 0 then
      Can_Retry := False;
    end if;
    -- Infine riaccoda presso Enter per
    -- ritentare.
    requeue Enter;
  end Retry;

  entry Enter(
    To_Add      : in     Positive;
    Max_Speed   :    out Positive;
    Leg_Length  :    out Positive) when True is
  begin
    -- Controllo sulla direzione di accesso del Treno.
    if  (Trains.Trains(To_Add).Current_Station /= First_End)
    and (Trains.Trains(To_Add).Current_Station /= Second_End)
    then
      raise Bad_Segment_Access_Request_Exception
        with "...";
    end if;
    -- Se e solo se il Treno corrente e' il primo della coda
    -- puo' continuare, altrimenti viene riaccodato su Retry.
    if Trains_Order.Get(1) /= To_Add then
      requeue Retry;
    end if;
    declare
      T : Positive;
    begin
      -- Rimuove il primo elemento della coda
      Trains_Order.Dequeue(T);
      -- Se vi e' almeno un Task in attesa su Retry,
      -- viene aperta la guardia.
      if Retry'Count > 0 then
        Retry_Num := Retry'Count;
        Can_Retry := True;
      end if;
    end;
    -- Riaccodamento all'entry privata che realizza la
```

```
    -- politica di accesso multiplo.
    requeue Perform_Enter;
  end Enter;
  ...
end Segment_Access_Controller;
```

Una volta che il controllo di accesso ordinato è stato superato, si può procedere al controllo di ingresso vero e proprio, mediante l'*entry* privata `Perform_Enter`. Essa controlla in primo luogo se il Treno accede dall'estremo `First_End` o `Second_End`, per poi poter proseguire all'accesso di conseguenza. Nel listato 5.4 riportato solo il codice per uno dei due casi, ovvero il caso in cui `Trains.Trains(To_Add).Current_Station = First_End`, in quanto sono analoghi.

Listing 5.4: Porzione della *entry* `Perform_Enter` per l'accesso dall'estremo `First_End`

```
potected body Segment_Access_Controller is
  ...
  entry Perform_Enter(
    To_Add       : in     Positive;
    Max_Speed    :    out Positive;
    Leg_Length   :    out Positive) when True is
  begin
    if Trains.Trains(To_Add).Current_Station = First_End
    then
      if Free then
        -- Il numero di Treni per direzione viene
        -- impostato a 1
        Train_Entered_Per_Direction := 1;
        -- Viene impostato ad occupato
        Free := False;
        -- Viene aggiornata la direzione di marcia
        -- corrente, con la Stazione di provenienza
        -- del Treno.
        Current_Direction :=
          Trains.Trains(To_Add).Current_Station;
      else
        if Trains.Trains(To_Add).Current_Station =
           Current_Direction
        then
          if Train_Entered_Per_Direction = Max then
            -- Se il numero di accessi e' il massimo
            -- consentito per direzione...
```

```
            if Retry_Second_End'Count > 0 then
              -- se vi sono task in attesa dall'estremo
              -- opposto del Segmento, allora il
              -- task corrente viene accodato presso la
              -- entry Retry_First_End, in attesa che
              -- arrivi il proprio turno.
              Can_Enter_First_End := False;
              requeue Retry_First_End;
            end if;
          else
            -- se il massimo numero di accessi per
            -- direzione NON e' stato raggiunto, allora
            -- viene incrementato il numero di accessi
            Train_Entered_Per_Direction :=
              Train_Entered_Per_Direction + 1;
          end if;
        else
          -- nel caso in cui il Treno corrente volesse
          -- accedere nel senso opposto al senso di
          -- marcia, dovra' attendere.
          requeue Retry_First_End;
        end if;
      end if;
    else
      -- Secondo caso simmetrico.
      ...
    end if;
    -- A questo punto il Treno ha avuto il permesso
    -- di accedere, e vengono effettuate le opportune
    -- inizializzazioni.
    ...

  end Perform_Enter;
  ...
end Segment_Access_Controller;
```

Nel caso in cui il Segmento sia libero (`Free=True`) allora il Treno può accedere al Segmento. Nel caso in cui invece il Segmento non sia libero, allora viene verificata la possibilità di accesso multiplo, ovvero se:

- La direzione di percorrenza del Treno è la stessa di quella corrente e il numero massimo di accessi per direzione (`Max`) non è stato raggiunto

- oppure se nessun Task è accodato all'estremo opposto, ovvero sulla *entry* `Retry_Second_End`.

In tutti gli altri casi il Task corrente viene riaccodato sulla *entry* `Retry_First_End` che avrà guardia booleana chiusa.

Se il Task corrente non è stato riaccodato ad un'altra *entry*, allora viene incrementato di uno il contatore dei Treni in transito, e viene inserito nella coda `Running_Trains` l'identificativo del Treno corrente. Vengono infine aggiornati i dati relativi alla velocità di percorrenza (il codice viene omesso per brevità).

Le *entries* private `Retry_First_End` e `Retry_Second_End` sono molto simili e sono utilizzate per mantenere accodati i Task relativi ai Treni in attesa di accedere al Segmento, rispettivamente presso l'estremo `First_End` e `Second_End`. Viene riportato nel listato 5.5 il codice che realizza l'*entry* `Retry_First_End`.

Listing 5.5: *entry* utilizzata per accodare i Task che rappresentano Treni in attesa di accesso presso il primo estremo del Segmento.

```
potected body Segment_Access_Controller is
  ...
  entry Retry_First_End(
    To_Add     :in    Positive;
    Max_Speed  :  out Positive;
    Leg_Length :  out Positive) when Can_Enter_First_End
  is
  begin
    -- Decremento del numero di Task che
    -- possono ri-tentare l'accesso
    Enter_Retry_Num := Enter_Retry_Num - 1;
    -- una volta che tale numero e' 0,
    -- la guardia viene richiusa.
    if Enter_Retry_Num = 0 then
      Can_Enter_First_End := False;
    end if;
    -- il nuovo tentativo viene effettuato ri-accodando
    -- il task corrente presso l'entry Enter.
    requeue Enter;
  end Retry_First_End;
  ...
end Segment_Access_Controller;
```

L'attesa dei task presso questa *entry* è regolata dalla guardia booleana `Can_Enter_First_End`, mentre il numero di tentativi di nuovo accesso al Segmento, viene regolato dal parametro `Enter_Retry_Num`.

5.1.17.2 Uscita dal Segmento

L'uscita da un Segmento da parte di un Treno, ha il prerequisito fondamentale per cui esso ha prima avuto accesso a tale Segmento, e quindi il suo identificativo univoco sarà contenuto nella coda `Running_Trains` della risorsa protetta `Segment_Monitor`. Il tipo *tagged* `Segment_Type` espone il metodo `Leave` che permette ad un Treno di richiedere l'uscita dal Segmento, il quale semplicemente invoca l'*entry* omonima di `Segment_Monitor`. Il processo di uscita avviene secondo l'ordine di percorrenza dei Treni all'interno del Segmento. La *entry* `Leave` della risorsa `Segment_Monitor` (listato 5.6), per prima cosa controlla che il Task correntemente in esecuzione sia effettivamente il prossimo a dover uscire dal Segmento. Tale controllo viene effettuato confrontando il primo elemento della coda `Running_Trains` con l'identificativo del Treno rappresentato dal Task in esecuzione: se essi sono uguali allora l'identificativo viene rimosso dalla coda e viene effettuata l'uscita, altrimenti il Task corrente viene accodato presso l'*entry* privata `Retry_Leave`.

Listing 5.6: Uscita di un Task dal Segmento.

```
potected body Segment_Access_Controller is
  ...

  entry Retry_Leave(
    Train_D : in Positive) when Can_Retry_Leave is
  begin
    -- Decremento del numero di task che
    -- possono ritentare l'uscita.
    Retry_Num := Retry_Num - 1;
    -- una volta che tale numero e' 0,
    -- la guardia viene richiusa per permettere
    -- nuovo accodamento
    if(Retry_Num = 0) then
      Can_Retry_Leave := False;
    end if;
    -- infine viene riaccodato il task corrente
    -- presso Leave, per ritentare l'uscita.
    requeue Leave;
  end Retry_Leave;

  entry Leave(
```

```
    Train_D : in Positive) when not Free is
  begin
    if Running_Trains.Get(1) = Trains.Trains(Train_D).ID
    then
      -- il primo elemento della coda viene rimosso
      declare
        T : Positive;
      begin
        Running_Trains.Dequeue(T);
      end;
      -- Se c'e' almeno un Task in attesa presso la
      -- guardia di uscita, essa puo' essere aperta
      -- per permettere al prossimo Task di uscire.
      if(Retry_Leave'Count > 0) then
        -- viene memorizzato il numero di task
        -- che possono tentare l'uscita.
        Exit_Retry_Num := Retry_Leave'Count;
        Can_Retry_Leave := True;
      end if;
      -- Decremento del numero di Treni che
      -- percorrono il Segmento.
      Trains_Number := Trains_Number - 1;
      if Trains_Number = 0 then
        -- Nel caso in cui il Semento si sia
        -- svuotato, la velocita' massima
        -- di percorrenza viene aggiornata
        Current_Max_Speed := Segment_Max_Speed;
        -- Inoltre se vi sono treni in attesa di
        -- accedere dalla direzione opposta,
        -- la guardia viene aperta.
        if Current_Direction = First_End then
          -- caso in cui la direzione corrente sia
          -- proveniente dal primo estremo del segmento.
          if Retry_Second_End'Count > 0 then
            -- se ci sono task in attesa presso la
            -- entry Retry_Second_End, allora essi
            -- potranno riprovare l'accesso.
            Enter_Retry_Num := Retry_Second_End'Count;
            -- viene aperta la guardia booleana
            Can_Enter_Second_End := True;
            Can_Enter_First_End := False;
          else
            -- se non vi sono treni in attesa, allore
```

```
            -- il Segmento viene dichiarato libero.
            Free := True;
          end if;
        else
          -- Caso simmetrico
          ...
        end if;
        -- Viene posto a libero il segmento.
        Free := True;
      end if;
      -- Viene aggiunto il Treno alla coda di arrivo
      -- presso la prossima Stazione.
      if Current_Direction /= First_End then
        Environment.Stations(First_End).Add_Train(
          Train_ID  => Train_D,
          Segment_ID  => Id);
      else
        Environment.Stations(Second_End).Add_Train(
          Train_ID  => Train_D,
          Segment_ID  => Id);
      end if;
    else
      -- Requeue alla entry Retry_Leave per
      -- rispettare l'ordine di percorrenza.
      requeue Retry_Leave;
    end if;
  end Leave;
  ...
end Segment_Access_Controller;
```

L'uscita viene completata nel seguente modo: per prima cosa viene verificata l'eventuale presenza di Task in attesa sulla guardia (chiusa) della *entry* `Retry_Leave`; successivamente, viene verificato se il Segmento è vuoto o se vi sono altri Treni in percorrenza. Nel primo caso, viene verificata l'eventuale presenza di Treni (Task) in attesa presso l'estremità opposta del Segmento, e in tal caso la loro guardia viene aperta per permettere di ritentare l'accesso. Se il Segmento è vuoto viene re-impostato il valore di `Free` a `True`. Infine, deve essere comunicato l'ordine di uscita alla successiva Stazione. Questa operazione si traduce nell'invocazione della procedura `Add_Train` messa a disposizione dall'interfaccia `Station_Interface` del package `Generic_Station`.

La soluzione presentata sfrutta gli strumenti offerti dal linguaggio Ada per garantire una semantica adeguata di accesso, percorrenza e uscita.

5.1.18 Generic_Platform

Il package `Generic_Platform` definisce l'interfaccia di classe `Platform_Interface` implementata da ciascun tipo rappresentante una Piattaforma.

5.1.19 Platform

Il package `Platform` definisce il tipo protetto `Platform_Type`, usato per regolare gli accessi alla Piattaforma. Esso mantiene i campi dato:

- `Can_Retry`: Variabile booleana usata come guardia per regolare i tentativi di accesso alla Piattaforma.
- `Retry_Count`: Variabile a valori interi usata per la regolazione dei tentativi di accesso alla Piattaforma.
- `Trains_Order`: Coda di interi FIFO, di tipo `Unlimited_Simple_Queue`, usata per memorizzare gli identificativi dei Treni che intendono accedere alla Piattaforma.
- `Terminated`: Variabile booleana che indica se è stata richiesta la terminazione del sistema.

`Platform_Type` si compone delle seguenti procedure protette:

`Add_Train(Train_ID:Integer)`
: Aggiunge l'identificativo `Train_ID` nella coda `Trains_Order`.

`Leave`
: Imposta il campo `Free` a `False` liberando la risorsa, e, nel caso in cui ci fossero Task in attesa presso la *entry* `Retry`, apre la guardia `Can_Retry` memorizzando il numero di tali task nel campo `Retry_Count`.

`Terminate`
: L'effetto di questa procedura è impostare il valore del campo dati `Terminated` a `True`.

e delle seguenti *entries*:

`Enter(Train_Descriptor_Index:Integer)`
: Ha guardia booleana sempre aperta (`True`). Essa verifica se il primo elemento della coda corrisponde al Treno corrente. In tal caso viene permesso l'accesso viene impostato lo stato della Piattaforma ad occupato (`Free = False`). Nel caso in cui invece tale condizione non si verifichi, il Task corrente viene riaccodato sulla *entry* `Retry`.

`Retry(Train_Descriptor_Index:Integer)`
É regolata dalla guardia booleana `Can_Retry or Terminated`. Se `Terminated=False`, essa decrementa il valore di `Retry_Count` e, nel caso in cui esso sia a 0, imposta il valore di `Can_Retry` a `False`. Infine riaccoda il Task corrente presso la *entry* `Enter`. Nel caso in cui `Terminated` sia `True` invece, non fa nulla.

Il package `Platform` definisce inoltre il tipo *tagged* `Platform_Handler`, che viene espone dei metodi alla Stazione per permettere di interagire con le Piattaforme. Esso contiene gli oggetti

- `The_Platform`, di tipo `Platform_Type`.
- `Priority_Controller`, di tipo `Priority_Access_Controller` (introdotto in sezione 5.1.16).

e implementa i metodi dell'interfaccia `Platform_Interface`:

`+ Add_Train(Train_Index:Integer)`
Aggiunge l'identificativo del Treno di indice `Train_Index` nella coda `Trains_Order` di `The_Platform`, il cui ordine regolerà gli accessi. Per fornire priorità, viene prima ottenuto accesso con

```
Priority_Controller.Gain_Access(Train_Index)
```

, e successivamente viene invocata la procedura `Add_Train` sull'oggetto `The_Platform`. Infine viene invocata la procedura protetta `Priority_Controller.Access_Gained`.

`+ Enter(`
`Train_Descriptor_Index:Integer,`
`Action:Route.Action)`
Per prima cosa cerca di ottenere l'accesso alla Piattaforma mediante l'*entry* `Enter` invocata sull'oggetto `The_Platform`, poi notifica l'avvenuto accesso al Controller Centrale e invoca il metodo privato `Perform_Entrance` per effettuare le operazioni necessarie all'arrivo del Treno.

`+ Leave(`
`Train_Descriptor_Index:Integer,`
`Action:Route.Action)`
Invoca il metodo privato `Perform_Exit` per effettuare le operazioni necessarie all'uscita del Treno dalla Piattaforma, e successivamente invoca la procedura protetta `Leave` sull'oggetto `The_Platform`.

+ `Add_Outgoing_Traveler(Traveler:Integer)`
  Aggiunge l'indice del Viaggiatore `Traveler` alla coda `Leaving_Queue`, che mantiene i Viaggiatori in attesa di partire dalla Piattaforma.

+ `Add_Incoming_Traveler(Traveler:Integer)`
  Aggiunge l'indice del Viaggiatore `Traveler` alla coda `Arrival_Queue`, che mantiene i Viaggiatori in attesa di arrivare alla Piattaforma.

- `Perform_Entrance(`
  `Train_Descriptor_Index:Integer,`
  `Action:Route.Action)`
  Effettua le operazioni necessarie all'ingresso di un Treno alla Piattaforma. In particolare, nel caso il cui il campo `Action` assuma il valore `Route.ENTER`, viene effettuata la discesa dei Viaggiatori. Tale operazione prevede che venga svuotata la coda `Arrival_Queue`, un elemento alla volta; per ciascun passeggero estratto viene verificato se sia in attesa o meno del treno corrente: se non lo è allora viene accodato nuovamente, altrimenti viene decrementato il numero di posti occupati del Treno e viene aggiornata la prossima tappa del viaggio (rispetto al Ticket posseduto), se prevista, altrimenti viene richiesta l'esecuzione dell'operazione `BUY_TICKET` a `Traveler_Pool`. Nel caso in cui la tappa successiva sia definita, viene richiesta invece l'esecuzione dell'operazione `LEAVE` a `Traveler_Pool`

- `Perform_Exit(`
  `Train_Descriptor_Index:Integer,`
  `Action:Route.Action)`
  Effettua le operazioni necessarie alla partenza di un Treno dalla Piattaforma. In particolare, nel caso il cui il campo `Action` assuma il valore `Route.ENTER`, viene effettuata la salita a bordo dei Viaggiatori. Tale operazione prevede che venga rimosso un Viaggiatore alla volta dalla coda `Leaving_Queue` e per ciascuno di essi verifica che sia in attesa o meno del Treno corrente. Nel caso in cui non lo sia, allora viene effettuato un controllo per verificare che il Treno atteso dal Viaggiatore in esame non sia già passato (se prenotato): in tal caso viene richiesta l'esecuzione dell'operazione `BUY_TICKET` a `Traveler_Pool`, altrimenti il Viaggiatore viene semplicemente riaccodato su `Leaving_Queue`.
  Nel caso in cui il Treno sia quello atteso dal Viaggiatore ma che la corsa corrente sia maggiore di quella prenotata, allora il viene richiesta l'esecuzione dell'operazione `BUY_TICKET` a `Traveler_Pool`. Se nessuno degli altri casi si è verificato, viene incrementato il numero di posti occupati all'interno del Treno e viene aggiornata la prossima tappa del

viaggio (rispetto al Ticket posseduto), se prevista, altrimenti viene richiesta l'esecuzione dell'operazione `BUY_TICKET` a `Traveler_Pool`. Nel caso in cui la tappa successiva sia definita, viene eseguita l'operazione `ENTER` per il Viaggiatore, in modo tale da garantire che esso si troverà in attesa presso la Stazione successiva prima che il Treno vi arrivi.

5.1.20 Generic_Station

Il package `Generic_Station` contiene la definizione dell'interfaccia `Station_Interface`, che dovrà essere implementata da tutti i tipi di Stazione. Essa espone i seguenti metodi di classe:

```
Enter(
    Descriptor_Index:Integer,
    Platform_Index:Integer,
    Segment_ID:Integer,
    Action:Route.Action)
```
Metodo invocato dai Task di tipo `Train_Task` per poter effettuare l'accesso alla piattaforma `Platform_Index` della Stazione.

```
Leave(
    Descriptor_Index:Integer,
    Platform_Index:Integer,
    Action:Route.Action)
```
Metodo invocato dai Task di tipo `Train_Task` per poter effettuare la partenza dalla piattaforma `Platform_Index` della Stazione.

```
Wait_For_Train_To_Go(
    Outgoing_Traveler:Integer,
    Train_ID:Integer,
    Platform_Index:Integer)
```
Metodo utilizzato per posizionare il Viaggiatore di indice `Outgoing_Traveler` in attesa del treno `Train_ID` presso la Piattaforma `Platform` per poter partire.

```
Wait_For_Train_To_Arrive(
    Incoming_Traveler:Integer,
    Train_ID:Integer,
    Platform_Index:Integer)
```
Metodo utilizzato per posizionare il Viaggiatore di indice `Outgoing_Traveler` in attesa del treno `Train_ID` presso la Piattaforma `Platform` per simulare la percorrenza all'interno del Treno `Train_ID`.

`Add_Train(`
`Train_ID:Integer,`
`Segment_ID:Integer)`
Metodo utilizzato per poter definire l'ordine di arrivo di un Treno `Train_ID` proveniente dal Segmento `Segment_ID`.

`Buy_Ticket(`
`Traveler_Index:Integer,`
`To:String)`
Metodo che permette ad un viaggiatore `Traveler` di poter effettuare una richiesta per ottenere un Ticket per la destinazione `To`.

Il package contiene, oltre all'interfaccia descritta, anche la definizione del tipo protetto `Access_Controller`. Questo tipo di risorsa protetta viene utilizzata dalle Stazioni per regolare l'ordine di richiesta d'accesso alle Piattaforme, in base all'ordine di uscita dal Segmento al quale tale risorsa fa riferimento. `Access_Controller` mantiene una coda `Trains_Order` di identificativi di Treni, di tipo `Unlimited_Simple_Queue`, alla quale è possibile aggiungere elementi mediante la procedura protetta esposta `Add_Train`. Esso inoltre espone l'*entry* `Enter(Train_Index:Integer)` con guardia sempre aperta (`True`) per permettere il primo tentativo di accesso, che avviene soltanto se il primo elemento della coda coincide con l'identificativo del Treno corrente. Nel caso il Treno non sia il prossimo secondo l'ordine definito da `Trains_Order`, allora il task corrente viene posto in attesa su guardia presso l'*entry* privata `Wait`, mediante il costrutto `requeue`. La procedure `Free` rimuove dalla coda il primo elemento, e, nel caso in cui vi siano Task in coda sulla guardia presso l'*entry* `Wait`, apre tale guardia in modo tale da permettere ad essi di ritentare l'accesso.

5.1.21 Regional_Station

Il package `Regional_Station` contiene la definizione del tipo classe (`tagged` nel linguaggio Ada) `Regional_Station_Type` il quale implementa l'interfaccia `Station_Interface`. Ciascuna istanza di classe `Regional_Station` contiene una tabella di hash, `Segments_Map_Order`, la quale per ciascun Segmento entrante associa un'istanza di `Access_Controller`; inoltre, essa contiene un array di oggetti di tipo `Platform_Type` (ovvero Piattaforme), in numero definito da configurazione, e una istanza di oggetto `Notice_Panel`. Ciascuna Stazione fornisce quindi da interfaccia all'esterno per l'accesso alle componenti interne.

Le operazioni svolte del metodo `Enter` sono le seguenti (si veda il listato 5.7:

Listing 5.7: Metodo `Enter` fornito dal tipo `Regional_Station_Type`

```
procedure Enter(
  This             : in out Regional_Station_Type;
  Descriptor_Index : in     Positive;
  Platform_Index   : in     Positive;
  Segment_ID       : in     Positive;
  Action           : in     Route.Action) is
begin
  -- Per prima cosa, richiede l'accesso all'access
  -- controller associato al segmento di porvenienza.
  This.Segments_Map_Order.Element(Segment_ID).Enter(
    Train_Index => Descriptor_Index);
  -- Aggiunge il Treno alla coda interna alla risorsa
  -- che gestisce ingresso e uscita dalla Piattaforma.
  This.Platforms(Platform_Index).Add_Train(
    Train_Index => Descriptor_Index);
  -- Una volta aggiunto il Treno, il controllore
  -- di accessi per il segmento Segment_ID puo'
  -- essere liberato.
  This.Segments_Map_Order.Element(Segment_ID).Free;
  -- Viene richiesto l'ingresso effettivo alla
  -- piattaforma corretta.
  This.Platforms(Platform_Index).Enter(
    Train_Descriptor_Index  => Descriptor_Index,
    Action          => Action);
  -- Notifica al Pannello informativo.
  This.Panel.Set_Train_Accessed_Platform(
    Train_ID  => Trains.Trains(Descriptor_Index).ID,
    Platform  => Platform_Index);
end Enter;
```

- Viene effettuato l'accesso alla Stazione, mantenendo l'ordine di uscita dal Segmento.
- Una volta ottenuto l'accesso, il Task corrente aggiunge l'identificativo del descrittore presso la coda interna alla piattaforma che regola l'ordine di ingresso in essa. Si noti che questa invocazione di procedura viene eseguita in modo concorrente tra tutti i Treni provenienti da Segmenti diversi che vogliono accedere alla stessa piattaforma, ma in mutua esclusione tra tutti quelli che provengono dallo stesso Segmento.

- Una volta che il descrittore è stato aggiunto alla coda interna alla piattaforma desiderate, il controller degli accessi alla Stazione può essere rilasciato, poiché l'ordine di accesso alla Stazione è garantito.

- Viene richiesto l'accesso vero e proprio alla Piattaforma che avverrà secondo l'ordine definito dalla propria coda interna.

- Notifica al pannello informativo dell'avvenuto ingresso del Treno alla Piattaforma.

Il metodo `Leave` si limita a liberare la Piattaforma desiderata mediante l'invocazione della procedura protetta `Leave` della risorsa a protezione della Piattaforma, e a notificare l'avvenuta dipartita del Treno al Pannello informativo della Stazione.

I metodi ridefiniti `Wait_For_Train_To_Go` e `Wait_For_Train_To_Arrive` aggiungono l'indice del Viaggiatore passato come parametro nelle code interne alla Piattaforma riservate alla partenza e all'arrivo dei Viaggiatori rispettivamente.

La ridefinizione del metodo `Add_Train` aggiunge l'identificativo del Treno passato come parametro alla coda interna al controllore di accessi creato per il Segmento di origine del Treno. Se tale controllore non esiste (la creazione avviene non per tutti i Segmenti in ingresso ma solo se essi sono effettivamente utilizzati) viene creato, e aggiunto quindi alla tabella `Segments_Map_Order`.

Infine, la ridefinizione del metodo `Buy_Ticket` effettua una invocazione della procedure `Get_Ticket` del package `Regional_Ticket_Office`.

5.1.22 Gateway_Station

Il package `Gateway_Station` contiene la definizione del tipo *tagged* `Gateway_Station_Type`, il quale implementa l'interfaccia `Station_Interface`. Il tipo `Gateway_Station_Type` contiene, come `Regional_Station_Type`, un array di oggetti di tipo `Platform_Type`, una mappa `Segments_Map_Order` contenente per ogni Segmento il controllore di accessi (`Access_Controller`) relativo, e un oggetto di tipo `Notice_Panel_Type`, che rappresenta il pannello informativo della Stazione. L'implementazione dei metodi astratti, è molto simile a quella fornita dal tipo `Regional_Station`; di seguito verranno descritte le differenze principali:

- Il metodo `Enter` esegue le operazioni descritte per l'equivalente metodo esposto da `Regional_Station`. Vi è però la possibilità che il Treno corrente possa essere trasferito su un Nodo diverso, se previsto dal percorso. A tal proposito, dopo aver avuto accesso alla Piattaforma desiderata, vengono recuperati, se previsti dal percorso, la prossima stazione

da raggiungere e il Nodo, ovvero la regione, sulla quale essa si trova. Per come un percorso è definito, la prossima stazione sarà la relativa Stazione di Gateway nel Nodo della tappa successiva a quella corrente. L'eventuale trasferimento del Treno viene effettuato mediante la procedura `Send_Train`, esposta dal package `Remote_Station_Interface`.

- Il metodo `Leave` una volta liberata la Piattaforma indicata dal parametro `Platform_Index`, controlla se il Treno proviene da un Nodo diverso da quello corrente. In questo caso, viene inviata un messaggio al Nodo di provenienza per comunicare l'avvenuta dipartite del Treno dalla Piattaforma con la procedura `Train_Left_Message` esposta da `Remote_Station_Interface`.

- Il metodo `Wait_For_Train_To_Go` controlla se la tappa successiva nel Ticket del Viaggiatore indicato da `Outgoing_Traveler` porta in una Stazione appartenente ad un Nodo diverso da quello corrente. In tal caso viene invocata la procedura `Send_Traveler_To_Leave` del package `Remote_Station_Interface` per far attendere il Viaggiatore presso la Piattaforma della Stazione di Gateway corrente appartenente a tale Nodo.

- Viene aggiunto il metodo `Occupy_Platform` che viene utilizzato per poter occupare una data Piattaforma, all'avvenuto trasferimento remoto di un Treno.

5.1.23 Handlers

Il package `Handlers` contiene un insieme di procedure utilizzate per la gestione dei messaggi remoti ricevuti. Ciascuno di esse aderisce alla seguente firma:

```
procedure *_Handler(Msg:Incoming_Message'Class)
```

`Station_Train_Transfer_Handler`
: Viene utilizzata per la gestione del messaggio `train_transfer`, che richiede il trasferimento di un Treno nel Nodo corrente, per poter poi proseguire il percorso. Tale procedure, estrae i dati dal messaggio ricevuto, ed esegue le seguenti operazioni:

  - Aggiorna i dati relativi al Treno passato;
  - aggiorna la Time Table per il percorso specificato;
  - occupa la Piattaforma nella Stazione di Gateway indicata;

- inserisce l'indice del Treno passato nella coda di esecuzione di `Train_Pool`.

Nel caso in cui le operazioni siano state svolte in modo corretto, viene inviata una risposta al messaggio comunicando l'esito positivo, altrimenti viene inviato un messaggio di errore.

`Station_Train_Transfer_Left_Handler`
Viene utilizzata per la gestione del messaggio di richiesta del Servizio `train_left_platfrom`, utilizzato come notifica della partenza di un Treno dalla Stazione di Gateway corrispondente, posizionata in un altro Nodo; tale procedura quindi libera la Piattaforma indicata nel messaggio e, nel caso in cui tale operazione si sia svolta in modo corretto, invia una risposta al messaggio comunicando l'esito positivo, altrimenti invia un messaggio di errore.

`Station_Traveler_Leave_Transfer_Handler`
Viene utilizzata per la gestione di messaggi relativi al Servizio `traveler_leave_transfer`, inviati per posizionare un Viaggiatore in attesa di un Treno per partire da una Piattaforma di una Stazione remota. Tale procedura estrae i dati passati dal messaggio, ed esegue le seguenti operazioni:

- Aggiorna i dati relativi al Viaggiatore e il suo Biglietto di Viaggio;
- pone il Viaggiatore in attesa presso la Piattaforma della Stazione indicate, mediante il metodo `Wait_For_Train_To_Go`

Infine, nel caso in cui le operazioni sopraelencate si siano svolte in modo corretto, invia una risposta al messaggio comunicando l'esito positivo, altrimenti invia un messaggio di errore.

`Station_Traveler_Enter_Transfer_Handler`
Viene utilizzata per la gestione di messaggi relativi al Servizio `traveler_enter_transfer`, inviati per posizionare un Viaggiatore in attesa di arrivare ad una Piattaforma di una Stazione remota. Tale procedura estrae i dati passati dal messaggio, ed esegue le seguenti operazioni:

- Aggiorna i dati relativi al Viaggiatore e il suo Biglietto di Viaggio;
- pone il Viaggiatore in attesa presso la Piattaforma della Stazione indicate, mediante il metodo `Wait_For_Train_To_Arrive`

Infine, nel caso in cui le operazioni sopraelencate si siano svolte in modo corretto, invia una risposta al messaggio comunicando l'esito positivo, altrimenti invia un messaggio di errore.

`Get_Ticket_Handler`
: Viene utilizzata per la gestione di messaggi relativi al Servizio `ticket_creation`, inviati dalla Biglietteria Centrale per ottenere un Ticket che collega due Stazioni interne alla Regione corrente. Tale procedura estrae gli identificativi delle due Stazioni dai parametri del messaggio ricevuto e crea un Ticket utilizzando la procedura `Create_Ticket` fornita dal package `Regional_Ticket_Office`.

`Is_Station_Present_Handler`
: Viene utilizzata per la gestione di messaggi relativi al Servizio `is_present`, inviati dalla Biglietteria Centrale per richiedere se una data Stazione è presente all'interno della Regione o meno.

`Ticket_Ready_Handler`
: Viene utilizzata per la gestione di messaggi relativi al Servizio `ticket_ready`, inviati dalla Biglietteria Centrale per recapitare il Biglietto creato, conseguentemente ad una richiesta di creazione (il processo è asincrono). Tale procedura estrae dal messaggio il Biglietto e l'indice del Viaggiatore per il quale esso è stato creato, assegna al Viaggiatore il biglietto creato, ed infine richiede l'esecuzione dell'operazione `TICKET_READY` per il Viaggiatore. Nel caso in cui nessun Biglietto fosse stato creato, viene atteso un tempo random dal Viaggiatore prima di inviare una nuova richiesta.

`Termination_Handler`
: Viene utilizzata per la gestione di messaggi relativi al Servizio `terminate`, inviati dal Controller Centrale per richiedere la terminazione del Nodo. Tale procedura, una volta ricevuto il messaggio, richiede la terminazione dei pool definiti nei package Task `Train_Pool` e `Traveler_Pool`, invocando le procedure `Stop`.

5.1.24 Traveler_Operations

Il package `Traveler_Operations` contiene la definizione di un insieme di tipi *tagged* che implementano l'interfaccia `Operation_Interface`, e quindi il suo metodo `Do_Operation`. Ciascuno di essi, contiene l'indice del Viaggiatore `Traveler_Manager_Index` per il quale eseguono l'operazione. I tipi definiti sono i seguenti:

`Leave_Operation_Type`
: Corrisponde all'operazione `LEAVE` e permette al Viaggiatore di indice

`Traveler_Manager_Index` di posizionarsi presso la coda di attesa della Piattaforma nella Stazione prevista dal percorso, per poter partire, invocando su quest'ultima il metodo `Wait_For_Train_To_Go`; nel caso in cui la Regione specificata dalla prossima tappa sia diversa dalla Regione corrente, allora viene invocato l'omomima procedura del package `Remote_Station_Interface`.

`Enter_Operation_Type`
: Corrisponde all'operazione `ENTER` e permette al Viaggiatore di indice `Traveler_Manager_Index` di posizionarsi presso la coda di attesa della Piattaforma nella Stazione prevista dal percorso, per poter arrivare, invocando su quest'ultima il metodo `Wait_For_Train_To_Arrive`, se tale Stazione è locale al Nodo corrente, altrimenti invocando l'omonima procedura del package `Remote_Station_Interface`.

`Buy_Ticket_Operation_Type`
: Corrisponde all'operazione `BUY_TICKET` e permette al Viaggiatore di richiedere la creazione di un Ticket presso la Stazione di partenza, dopo aver atteso un tempo random.

`Ticket_Ready_Operation_Type`
: Corrisponde all'operazione `TICKET_READY` e permette al Viaggiatore, una volta ricevuto il Biglietto creato, di poter eseguire l'operazione `LEAVE`.

5.2 Biglietteria Centrale

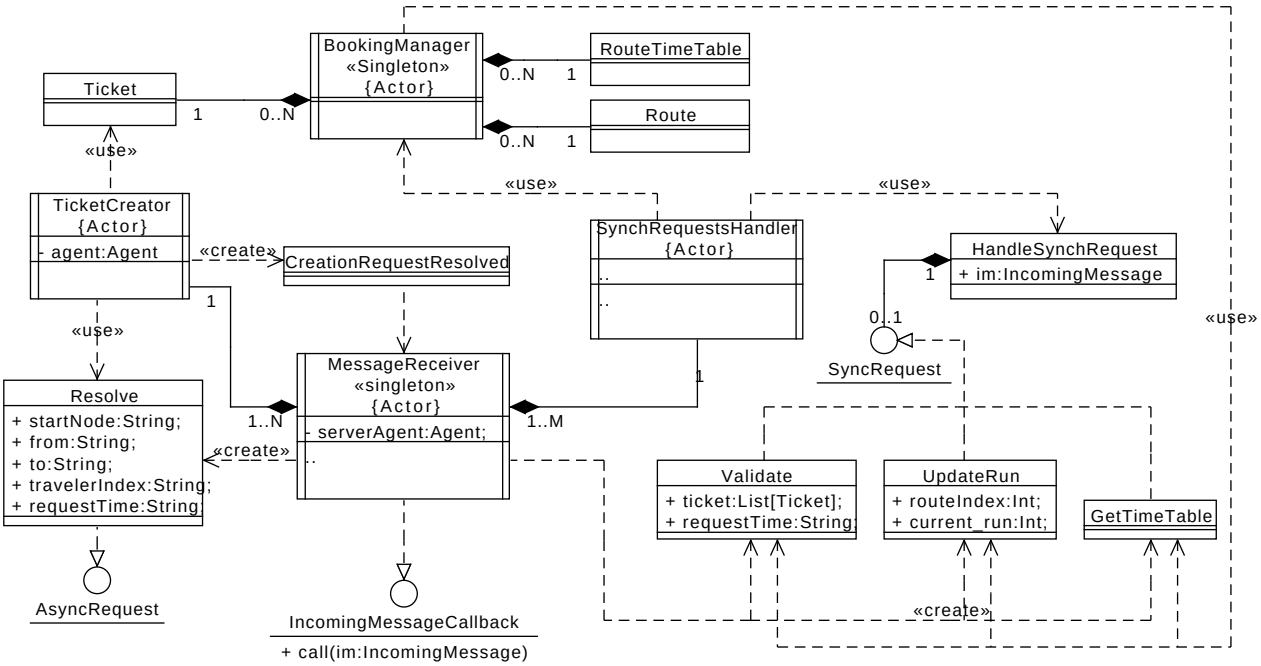

Figura 5.2: Diagramma delle classi semplificato che mostra le relazioni principali tra le classi che compongono la Biglietteria Centrale.

Il diagramma delle classi in figura 5.3 presenta l'architettura di massima della componente. La dicitura `Actor` indica che la classe estende la classe astratta `scala.actors.Actor`. Di seguito verrà effettuate una breve descrizione delle classi più importanti, e verrà utilizzato il concetto di *Attore* utilizzato dal linguaggio Scala.

5.2.1 MessagesReceiver

La classe Singleton `MessagesReceiver`, rappresenta un *Attore* il cui compito è ricevere messaggi remoti (implementa infatti l'interfaccia `IncomingMessageCallback`). Tra i parametri principali mantenuti dall'unico oggetto di tipo `MessagesReceiver`, vi sono un'istanza di agente messo a disposizione dal middleware *Yami4* per la ricezione dei messaggi remoti, e due Pool di *Attori*, rispettivamente di tipo `TicketCreator` e `SynchRequestsHandler`, memorizzati in due array. In

base al tipo di richiesta remota ricevuta, l'oggetto `MessagesReceiver` delega la gestione del messaggio ad uno degli attori nei due array, a seconda se la richiesta è di natura asincrona o sincrona. I tipi di messaggio remoto ricevuti possono essere:

- `create_ticket`: richiesta di creazione di un Ticket, che viene delegata ad uno degli attori di tipo `TicketCreator` mantenuti, inviando ad esso un messaggio di tipo `Resolve`. Il risultato dell'operazione richiesta viene restituito in maniera asincrona al richiedente.
- `get_time_table`: richiesta effettuata da un Nodo di simulazione per ottenere la tabella degli orari. La natura della richiesta è sincrona, e viene quindi delegata ad uno degli attori di tipo `SynchRequestsHandler`, inviando ad esso un messaggio di tipo `HandleSynchRequest`.
- `update_run`: richiesta di aggiornamento della corsa corrente, anch'essa di natura sincrona e quindi delegata ad un *Attore* di tipo `SynchRequestsHandler`, inviando ad esso un messaggio di tipo `HandleSynchRequest`.
- `validate`: richiesta di validazione di un dato Ticket, anch'essa di natura sincrona e quindi delegata ad un *Attore* di tipo `SynchRequestsHandler`, inviando ad esso un messaggio di tipo `HandleSynchRequest`.
- `marker`: messaggio effettuato per attuare la procedura di terminazione della Biglietteria Centrale; alla ricezione del primo `marker`, l'oggetto `MessagesReceiver` inserisce le successive richieste di creazione di un Ticket in una coda apposita, alla seconda ricezione richiede la terminazione di tutti gli attori attivi mediante l'invio di un messaggio `Stop`.

5.2.2 TicketCreator

La classe `TicketCreator` rappresenta un *Attore* incaricato di creare un Biglietto. Esso svolge le seguenti operazioni:

- Ricerca del Nodo (ovvero della Regione) nel quale la stazione di destinazione è contenuta.
- Una volta individuato il Nodo, costruzione di un *percorso* attraverso più Nodi per il raggiungimento della destinazione.
- Richiesta di creazione di porzioni di Ticket ai Nodi del *percorso*.

- Validazione dei Biglietti ottenuti attraverso l'invio di un messaggio all'attore `BookingManager`.
- Unione dei vari Ticket in un unico Biglietto.
- Restituzione del Biglietto creato o segnalazione di errore se la procedura non è stata completata.

5.2.3 SynchRequestsHandler

La classe `SynchRequestsHandler` rappresenta un `Attore` incaricato di gestire le richieste sincrone di validazione di un biglietto, di aggiornamento della corsa corrente, e di ottenimento della tabella degli orari. Esso attende fino al completamento dell'operazione assegnata, al termine della quale invia un messaggio di risposta opportuno al richiedente, tramite l'oggetto di tipo `IncomingMessage` contenuto nel messaggio `HandleSynchRequest` ricevuto.

5.2.4 BookingManager

L'*Attore* `BookingManager` rappresenta una entità *server* a protezione delle tabelle orarie e delle prenotazioni. Esso ha il compito di serializzare gli accessi a queste risorse, in modo tale da evitare modifiche inconsistenti ai dati in esse contenuti. Esso gestisce messaggi di tipo `UpdateRun` per aggiornare l'indice della corsa corrente di un Treno, di tipo `Validate` per verificare la disponibilità di posti a sedere per la lista di Ticket ricevuta, e di tipo `GetTimeTable` per l'ottenimento della tabella degli orari.

5.3 Server dei Nomi

Il Server dei Nomi è realizzato mediante un unico oggetto *Attore* `NameServer`, il quale implementa l'interfaccia `IncomingMessageCallback`. Esso mantiene un'istanza della classe `Agent`, per poter ricevere messaggi remoti ad un dato indirizzo, e una tabella per memorizzare le coppie `<entità,indirizzo>`. I messaggi remoti che esso può ricevere sono:

- `bind`: Permette al richiedente di registrare una nuova coppia `<entità,indirizzo>`; se l'identificativo dell'entità è già presente, esso viene sovrascritto.
- `resolve`: Dato un identificativo di entità, restituisce, se presente nella tabella, l'indirizzo al quale quell'entità si trova.
- `list`: Restituisce tutte le coppie `<entità,indirizzo>` possedute.

- `remove`: Rimuove dalla tabella la voce corrispondente all'identificativo passato.
- `terminate`: Il Server dei Nomi termina la propria esecuzione.

5.4 Controllo Centrale

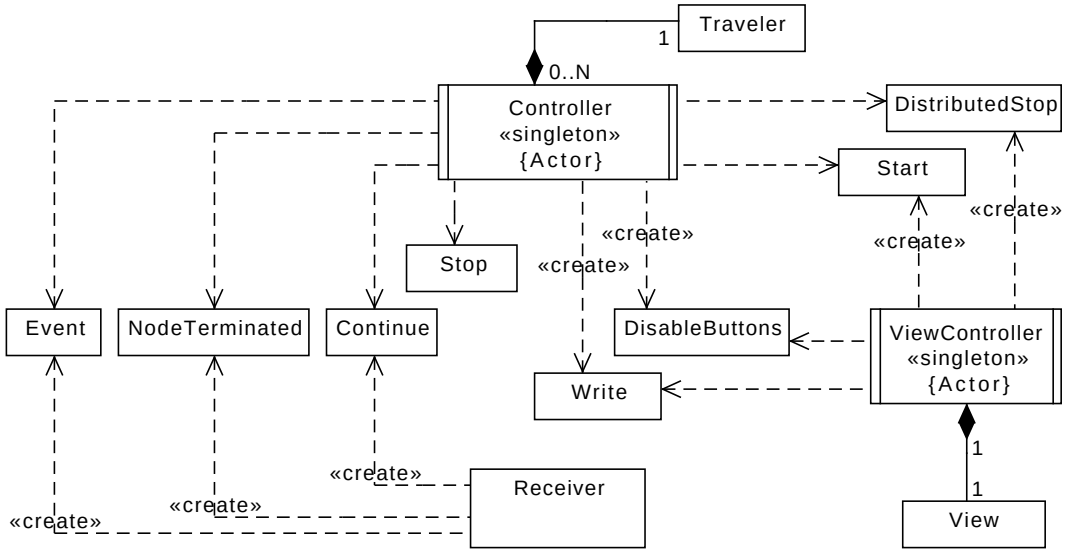

Figura 5.3: Diagramma delle classi semplificato che mostra le relazioni principali tra le classi che compongono il Controller Centrale.

L'entità di Controllo Centrale ha i seguenti compiti:

- ricevere gli eventi generati dai Nodi di Simulazione;
- gestire il meccanismo `Publish/Subscribe` per l'invio di eventi push in modalità push;
- gestire Avvio e Terminazione dell'intero sistema.

Essa è composta dall'oggetto `ControllerMain`, il quale mantiene una istanza della classe `Receiver`, utilizzata per la ricezione dei messaggi remoti, dall'oggetto *Attore* `Controller` che gestisce la pubblicazione di eventi e la Terminazione, e dall'oggetto *Attore* `ControllerView`, il quale gestisce una interfaccia grafica per permettere l'interazione dell'utente.

5.4.1 Receiver

La classe `Receiver` contiene un istanza di Agente remoto, per la ricezione dei seguenti tipi di messaggio:

- `event`: Rappresenta un evento, che viene inviato all'oggetto `Controller` mediante un messaggio di tipo `Event`.
- `central_office_ack`: Rappresenta il messaggio inviato dalla Biglietteria Centrale per comunicare l'avvenuta terminazione delle richieste pendenti; alla ricezione di questo messaggio viene inviato un messaggio `Continue` all'oggetto `Controller` per portare a termine la procedura di Terminazione.
- `node_terminated`: Messaggio inviato da un Nodo per notificare l'avvenuta terminazione. Alla ricezione di tale messaggio, viene inviato un messaggio `NodeTerminated` all'oggetto (attore) `Controller`.

5.4.2 Controller

L'oggetto `Controller`, mantiene tre tabelle, nelle quali memorizza lo stato della simulazione:

- `trainsOccupation`: per ciascun Treno, mantiene la lista dei Viaggiatori in transito;
- `segmentsTrain`: per ciascun Segmento, mantiene la lista dei Treni in percorrenza;
- `stationsTrains`: per ciascuna Stazione, mantiene una lista di coppie `(Treno,Piattaforma)`.

Il contenuto di tali tabelle viene semplicemente stampato su output standard. Esso gestisce inoltre i seguenti messaggi:

- `Event`, che rappresenta un evento di simulazione, il quale viene pubblicato tramite l'istanza di Agente remoto mantenuta. Vengono inoltre estratti i dati contenuti per poter aggiornare le tabelle descritte.
- `DistributedStop`, inviato da `ControllerView` a seguito dell'interazione dell'utente con la componente grafica. Alla ricezione viene inviato un messaggio remoto `marker` alla Biglietteria Centrale.
- `Continue`, alla cui ricezione viene completata la procedura di terminazione, inviando a ciascun Nodo di simulazione una richiesta di terminazione.

- `NodeTerminated`, che indica la terminazione di uno specifico Nodo. Una volta che tutti i Nodi sono terminati, viene inviato un secondo `marker` alla Biglietteria Centrale.
- `Stop` il quale serve a completare la procedura di terminazione interna del Controller Centrale.
- `Start` il quale serve per avviare la procedura di avvio dell'intero sistema.

Appendice A

Istruzioni

A.1 Requisiti

Per compilare ed eseguire il prototipo realizzato, è necessario disporre dei seguenti requisiti:

- Sistema operativo Unix-Like.
- compilatore Ada `gnatmake` versione 2012.
- compilatore `gcc` versione 4.7.2.
- compilatore `g++` versione 4.7.2.
- distribuzione Scala, in particolare i compilatori `scalac` e `fsc` versione 2.10.1.
- libreria `Gnatcoll 1.5w`.
- server MVC *Play* per Scala, versione 2.1.0 o superiore.
- browser web *Chrome*, versione 26.0.1410.63.

Per semplicità, sono state riportate le versioni delle librerie utilizzate per lo sviluppo, tuttavia non è escluso che possa essere utilizzato anche con versioni più recenti.

A.2 Installazione

Il sistema viene fornito in una directory `TrainProject`. Da tale directory, è sufficiente lanciare il comando `sh compile_all.sh`, il quale effettuerà la compilazione di tutte le componenti.

A.3 Avvio

L'avvio del sistema si effettua eseguendo i seguenti script presenti nella cartella principale:

- `run_play.sh` : Avvia il server *Play*, necessario per la rappresentazione grafica della simulazione. Il suo avvio può richiedere alcuni secondi.
- `run_all.sh` : Avvia le entità `Name_Server`, `Central_Controller` e `Central_Ticket_Office`.
- `run_simulation.sh` : Avvia i nodi di simulazione.

In seguito, è necessario aprire il browser *Chrome* e aprire la pagina all'indirizzo `localhost:9000`; infine per avviare la simulazione è sufficiente premere il pulsante *Start* dall'interfaccia grafica fornita dal Controller Centrale.

Viene fornita una configurazione di base composta da 3 Regioni; tuttavia è possibile utilizzare configurazioni diverse, modificando opportunamente i file `JSON` presenti nella cartella `configuration` presente nella root del progetto. Vengono fornite le seguenti regole:

- `routes.json`: deve contenere un array di percorsi, identificato con chiave `routes`. Ciascun percorso dovrà essere dotato di:
  - un campo `id` contenente l'identificativo univoco.
  - un array con chiave `route` composto da tappe, ciascuna con i seguenti campi:
    - `start_station`: Indice della Stazione di partenza;
    - `next_station`: Indice della Stazione di destinazione della tappa;
    - `start_platform`: Indice della Piattaforma di partenza;
    - `platform_index`: Indice della Piattaforma di destinazione;
    - `next_segment`: Indice del Segmento da utilizzare per raggiungere `next_station`;
    - `node_name`: Nome del Nodo (Regione) sul quale si trova `next_station`;
    - `leave_action`: Azione da intraprendere alla partenza (`ENTER` o `PASS`);
    - `enter_action`: Azione da intraprendere all'arrivo (`ENTER` o `PASS`).

- `time_table.json`: deve contenere per ciascun percorso definito in `routes.json` una tabella degli orari, in un array identificato con `time_table`. Per ciascuna tabella devono essere specificati `route`, indice del percorso di riferimento, e una matrice di orari di partenza `time`, di dimensione N x M, dove N è il numero di corse, che dev'essere ≥ 2, ed M è il numero di Tappe del percorso.

- `trains.json`: deve contenere un array di treni, identificato da `trains`. Per ciascun Treno si devono specificare i seguenti campi:

  - `id` : Identificativo univoco, di valore intero positivo;
  - `speed`: Velocità > 0 del Treno;
  - `max_speed`: Velocità massima > 0 del Treno;
  - `current_station`: Indice della stazione di partenza;
  - `next_stage`: Deve valere sempre 1;
  - `route_index`: Indice dell'array di percorsi definito in `routes.json` che rappresenta il percorso assegnato;
  - `sits_number`: Numero di posti a sedere > 1;
  - `occupied_sits`: Deve valere 0;
  - `type`: Tipo del Treno, tra `regionale` o `fb`;
  - `start_node`: Nome del nodo di partenza.

- `travelers.json`: deve contenere un array di Viaggiatori, identificato con `travelers`. Ciascun Viaggiatore conterrà un sotto-oggetto `traveler` per il quale vanno specificati `id`, `name` e `surname`, e i seguenti altri campi:

  - `next_operation`: inizialmente sempre 1;
  - `destination`: Stazione di destinazione;
  - `start_station`: Stazione di partenza;
  - `start_node`: Nome del nodo di partenza.

- `[Node_Name]-stations.json`: deve contenere un array di stazioni `station`, e per ciascuna di esse va specificato `name`, nome univoco, `platform_number`, numero di piattaforme, e `type`, che può essere `R` o `G`.

- `[Node_Name]-segments.json`: deve contenere un array `segments` di Segmenti, e per ciascuno di essi va specificato:

  - `id`: Identificativo univoco intero positivo;
  - `max_speed`: Velocità massima di percorrenza;
  - `max`: Valore intero positivo, massimo numero di accessi per estremo;
  - `length`: Lunghezza del Segmento (si consiglia una misura tra 0 e 1000);
  - `first_end`: Indice della prima Stazione collegata;
  - `second_end`: Indice della seconda Stazione collegata.

- `[Node_Name]-topology.json`: deve contenere un array `topology` di elementi, e indica per ciascuna Stazione, le stazioni raggiungibili e la lunghezza del Segmento per raggiungerle. Ciascun elemento dell'array deve contenere i campi:
  - `station`: Indice della stazione di riferimento;
  - `neighbours`: Array di indici delle Stazioni limitrofe;
  - `distance`: Array di distanze, per raggiungere ciascuna Stazione in `neighbours`.

- `links.json`: Contiene per ciascuna coppia ordinata di Regioni `<region1,region2>`, la lista delle stazioni di gateway che le collegano. Ciascun elemento di questa lista è una coppia `<regione,stazione_gateway>`.

Una volta definita la configurazione, è necessario lanciare lo script `run.sh` nella cartella `configuration/shortest_path`, specificando il file `[Node_Name]-topology.json` e il nome del nodo (ad esempio, `sh run.sh Node_1-topology.json Node_1`), per ciascun nodo di simulazione. Questo genererà i file `[Node_Name]-paths.json` contenenti i cammini minimi utilizzati per la creazione dei Tiket. Infine, è necessario modificare lo script `run_simulation.sh` aggiungendo i comandi per inizializzare i nodi di simulazione: per ciascun nodo dovrà essere aggiunto un comando nella forma:

```
echo "Loading [Node_Name]..."
gnome-terminal --title="[Node_Name]"
    --command="sh railway/run.sh [Node_Name] [Node tcp address]"
echo "done"
```

A.4 Visualizzazione della Simulazione

Per poter avere una rappresentazione grafica degli eventi generati dalla simulazione, è sufficiente aprire in una finestra del browser *Chrome* la pagina

all'indirizzo `http://localhost:9000`. Essa è composta da una parte dove è visibile la configurazione con Stazioni, Piattaforme, Treni e Passeggeri, e da una parte dove sono mostrati i Pannelli di notifica delle Stazioni. Nel corso della simulazione, è possibile che graficamente alcuni Treni si sovrappongano, ma questo è solo dovuto alla rappresentazione. La rappresentazione grafica fornita, è relativa alla configurazione iniziale; nel caso in cui si volesse realizzare una nuova configurazione, è necessario creare una nuova applicazione grafica per essa.

A.5 Terminazione

La terminazione viene iniziata premendo il pulsante `Stop` disponibile nell'interfaccia grafica messa a disposizione dal Controller Centrale; al termine di tale procedura, esso può essere terminato semplicemente chiudendo la finestra grafica.