F1 Simulator

Progetto di Sistemi concorrenti e distribuiti

28 Ottobre 2010

Sommario

Relazione sul progetto di Sistemi Concorrenti e Distribuiti.

Informazioni documento

Nome file	Relazione.pdf
Versione	2.1
Distribuzione	Prof. Vardanega Tullio
	Miotto Nicola
	Nesello Lorenzo

Indice

Elenco delle figure

1 Progetto

Il progetto riguarda l'analisi e la risoluzione delle problematiche di progettazione di un simulatore concorrente e distribuito di una competizione sportiva assimilabile a quelle automobilistiche di Formula 1. Il sistema da simulare dovrà prevedere:

- un circuito selezionabile in fase di configurazione, dotato della pista e della corsia di rifornimento, ciascuna della quali soggette a regole congruenti di accesso, condivisione, tempo di percorrenza, condizioni atmosferiche, ecc.
- un insieme configurabile di concorrenti, ciascuno con caratteristiche specifiche di prestazione, risorse, strategia di gara, ecc.
- un sistema di controllo capace di riportare costantemente, consistentemente e separatamente, lo stato della competizione, le migliori prestazioni (sul giro, per sezione di circuito) e anche la situazione di ciascun concorrente rispetto a specifici parametri tecnici
- una particolare competizione, con specifica configurabile della durata e controllo di terminazione dei concorrenti a fine gara.

2 Enunciazione problematiche

Nella seguente sezione verranno esposte le problematiche emerse nel corso dell'analisi del sistema da progettare. Tali problematiche derivano, oltre che dalla tipologia di progetto, anche dalle caratteristiche implicite del sistema:

- il sistema è concorrente. Di conseguenza presenterà un numero maggiore di 2 entità attive che eseguiranno concorrentemente;
- alcune risorse sono condivise e accedute quindi concorrentemente;
- il simulatore viene eseguito su un sistema operativo del cui scheduler non sono conosciute le specifiche;
- il sistema presenta delle componenti distribuite nella rete;

2.1 Gestione del tempo

Un simulatore di formula 1 racchiude intrinsecamente dei vincoli in termini di coerenza temporale. Una competizione è scandita da istanti di tempo: un istante iniziale, uno finale e vari istanti intermedi che segnano, per esempio, la fine della lap di un concorrente, oppure il passaggio di un concorrente da un settore

a quello dopo del circuito. Vi sono inoltre vincoli di coerenza temporale dati dai tempi accumulati nel corso della gara. Questi potrebbero essere il tempo di attraversamento di un tratto come il tempo necessario a effettuare un giro. I secondi, chiaramente, dipendono dai primi. Nel sistema simulato è quindi necessario che i tempi non presentino inconsistenze. Per esempio non deve risultare che un concorrente inizi la corsa all'istante 0, impieghi 4,5 e 15 secondi per percorrere i tre settori e poi il tempo di lap risulti essere 11.
Questo potrebbe risultare problematico dal momento che lo scheduler non rispetta vincoli di tempo definiti o comunque conosciuti a priori.

2.2 Sorpassi impossibili

Una problematica affrontata nel corso dell'analisi del simulatore da progettare è stata quella relativa ai sorpassi. Come accennato all'inizio della sezione, non possono essere fatte assunzioni di determinismo sullo scheduler. Ipotizzando un sistema in cui ogni concorrente corrisponda ad un'entità attiva (task) e in cui ogni tratto del circuito sia una risorsa condivisa fra i task a molteplicità limitata (come è plausibile pensare per un simulatore di F1), è possibile che si verifichino scenari anomali.
Per esempio:

1. un task concorrente dovrebbe, per questioni temporali, iniziare ad attraversare un tratto e ottenere quindi la risorsa;
2. lo scheduler prerilascia il task e assegna un quanto di tempo ad un altro task concorrente che in termini temporali è dietro. Il task ottiene la risorsa tratto (la stessa del task precedente);
3. il task ottiene altri quanti e attraversa;
4. lo scheduler prerilascia il task corrente e riassegna il processore al vecchio task il quale anche attraversa il tratto.

Non prestando attenzione a possibilità di questo tipo, se il tratto fosse di molteplicità 1 si avrebbe che un concorrente appare alla fine del tratto quando, per questioni fisico/temporali, avrebbe dovuto rimanere dietro.

2.3 Determinismo

Come definito in *Simulation: The Engine Behind The Virtual World, Roger D. Smith, Chief Scientist, ModelBenders LLC*:

"Simulation is the process of designing a model of a real or imagined system and conducting experiments with that model. The purpose of simulation experi-

ments is to understand the behavior of the system or evaluate strategies for the operation of the system."

Considerando quindi che la simulazione deve permettere di comprendere il comportamento del sistema, è necessario che il sistema abbia un comportamento prevedibile. Ciò non significa che ogni componente di non determinismo sia non desiderata. Il non determinismo può essere inserito in modo controllato, ovvero consapevoli del contesto e dei momenti in cui esso si possa verificare. Soprattutto, il non-determinismo deve rispecchiare un eventuale non determinismo presente anche nel sistema reale e non solo in quello simulato.
Il comportamento dello scheduler sottostante il sistema che verrà sviluppato non è prevedibile. Considerando che il non determinismo introdotto dallo scheduler non è controllabile, si presenta il problema di riuscire a progettare il simulatore in modo che il comportamento sia del tutto indipendente dall'architettura del sistema operativo su cui viene eseguito. In questo modo si potrà avere un sistema la cui correttezza sia verificabile e capace di fornire dati consistenti.

2.4 Componenti di non determinismo

Dopo aver analizzato le problematiche dovute al determinismo del progetto si possono pensare anche ai fattori che diano del non determinismo. Per non determinismo si intende la possibilità di non prevedere precisamente l'andamento della gara a priori. Questa componente può essere non desiderabile per certi aspetti mentre, se gestita, può dare del valore aggiunto alla simulazione. Nel caso del simulatore di formula 1 è desiderabile riuscire a inserire componenti che introducano il non determinismo. Esempi di non determinismo che si possono trovare progettando un simulatore e che possono portare valore aggiunto sono ad esempio interazioni dell'utente esterno al sistema tramite azioni esplicite . Tali azioni non sono prevedibili a priori e provocano quindi una mutazione delle condizioni di esecuzione del sistema. Se tali situazioni vengono gestite, e non precludono il proseguo della simulazione, il risultato finale sarà diverso da quello previsto (introducendo così il non determinismo) ma la correttezza del sistema sarà garantita.

2.5 Stalli

In un sistema concorrente lo stallo è una delle problematiche più importanti da affrontare. Per stallo si intende lo stato in cui nessun processo può più eseguire. Affinchè si verifichi lo stallo di devono verificare tutte le 4 pre-condizioni ben riconoscibili:

- Mutua Esclusione : assicura che, a ogni instante, non più di un processo abbia possesso di una risorsa (fisica o logica) condivisa.La sequenza di azioni che opera sulla risorsa è detta sezione critica. Nel realizzare un simulatore di Formula 1 bisognerà fare in modo che prevedendo l'uso esclusivo di risorse, come ad esempio pezzi di tracciato, queste non pregiudichino il buon funzionamento del sistema.

- Cumulazione di risorse : i processi possono accumulare risorse e trattenerle mentre attendono di acquisirne altre. In un simulatore concorrente di una gara di Formula 1 bisognerà controllare che le risorse acquisite siano solo quelle strettamente necessarie all'esecuzione (ad esempio il pezzo di tracciato e non tutta la pista)

- Assenza di prerilascio : le risorse vengono rilasciate solo volontariamente.

- Attesa Circolare : un processo attende almeno una risorsa in possesso del successivo processo in catena. Nella progettazione di un sistema distribuito bisognerà quindi evitare la formazione di catene chiuse di processi che attendono una risorsa.

Nella progettazione e realizzazione del sistema si dovrà quindi fare attenzione per evitare il verificarsi delle quattro condizioni, impedendo così gli stalli. Alcune condizioni dovranno essere permesse per il corretto svolgimento della competizione, ma in ogni istante bisognerà controllare che non si stiano verificando tutte e 4 insieme, impedendo cosi la presenza di una zona potenziale di stallo.

2.6 Realismo fisico

Il realismo fisico di un simulatore di formula uno è determinato principalmente da due fattori:

- realismo dato dall'interazione tra ogni concorrente e l'ambiente statico. Ovvero, l'impatto che hanno sulla corsa le caratteristiche fisiche dell'ambiente circostante quali, ad esempio, forma e caratteristiche della pista;

- realismo dato dall'interazione tra un concorrente e gli altri concorrenti. Questo tipo di realismo dipende quindi da caratteristiche dinamiche dell'ambiente e richiede che le valutazioni dei singoli concorrenti vengano fatte in rapporto allo stato degli altri concorrenti. Viceversa, le scelte dei singoli concorrenti devono poter influenzare i movimenti degli altri concorrenti.

2.7 Gestione delle istantanee di gara

Requisito essenziale per poter presentare i dati relativi all'andamento della competizione è riuscire ad ottenere una snapshot della gara in un determinato istante. Ovvero, dato un istante di tempo t o un evento e, bisogna poter risalire allo stato dei concorrenti e della gara in generale a partire da t o da e. In tal modo sarà reso possibile monitorare l'evolversi della competizione. Bisogna però tenere in considerazione che qualunque entità provveda allo scatto dell'istantanea sarà soggetta alle stesse problematiche legate alle altre entità attive dipendenti dallo scheduler. Di conseguenza è lecito pensare che uno snapshot possa risultare inconsistente se viene eseguito, per motivi di prerilascio, metà ad un istante t e l'altra metà ad un istante $t+f$, ad esempio. La distribuzione aggiunge un ulteriore livello di complessità al problema, poiché il ritardo potrebbe essere causato non solo dallo scheduler, ma anche dalla rete.

2.8 Problematiche di distribuzione

Progettando un sistema distribuito bisogna tenere in considerazione alcune problematiche che, se risolte, portano a un prodotto distribuito robusto. I punti principali su cui focalizzare l'attenzione sono

- Apparenza all'utente di un sistema unitario e non l'insieme di più elaboratori
- Comunicazione fra elaboratori nascosta all'utente
- Scelta del livello di distribuzione
- Architettura invariata rispetto al sistema in locale

Simulando una competizione di formula uno bisognerà fare in modo che la comunicazione (ad esempio concorrente - box) sia robusta, in quanto, come nella realtà, può precludere il buon andamento della gara. Inoltre bisogna riuscire a fornire funzionalità adatte ad eventuali osservatori esterni (che non influiscono sulla gara) per coinvolgerli in prima persona con la gara, un po' come avviene nella realtà con le tv collegate direttamente dall'autodromo ma visibili in tutto il mondo.
La comunicazione remota fra componenti dislocati in diversi nodi nella rete, quindi, presenterà certamente delle specifiche problematiche da affrontare. La più critica riguarda, appunto, la robustezza. È difficile o addiritura utopico sperare nell'affidabilità della rete. La rete presenta sempre dei fault, la cosa importante è gestirli e non farli propagare in errori. Sicuramente quindi bisognerà minimizzare la probabilità che un nodo distribuito, perdendo il contatto con il resto del sistema, possa provocare un malfunzionamento globale.

2.9 Avvio del sistema

L'avvio del sistema presenta dei problemi legati alla natura concorrente e distribuita dello stesso.
In dettaglio:

- la natura distribuita introduce il problema della messa in connessione dei nodi. Quando un nodo si connette è necessario che esso sappia dove sono dislocati gli altri nodi (o almeno quelli necessari) e che gli altri nodi possano reperire quello appena connesso.
- la natura concorrente invece introduce dei problemi relativi alla comunicazione delle entità attive. Tali entità saranno presumibilmente messe in comunicazione o tramite risorse condivise o in modo diretto. Si prospetta quindi la possibilità che in fase di avvio (essendo l'avvio un processo sequenziale) alcune entità richiedano la connessione con altre che non siano ancora pronte o allocate, causando un fault in alcuni casi, in altri un'esecuzione con risultati errati.

2.10 Stop del sistema

Nel caso specifico di un simulatore di formula 1, lo stop deve avvenire innanzitutto a livello logico. Ovvero, deve essere possibile al sistema poter capire quando la gara è finita in modo da poterlo annunciare.
Una volta che la competizione risulti essere completata, le risorse non più necessarie devono essere deallocate e i task fermati.
Quando il sistema viene interrotto definitivamente, non devono più essere presenti server in attesa di connessioni o thread attivi.

2.11 Problematiche di distribuzione

Un sistema che funzioni grazie alla comunicazione remota fra componenti dislocati in diversi nodi nella rete, presenta certamente delle specifiche problematiche da affrontare. La più critica riguarda di sicuro la robustezza. È difficile o addirittura utopico sperare nell'affidabilità della rete. La rete presenta sempre dei fault, la cosa importante è gestirli e non farli propagare in errori. Sicuramente quindi bisognerà minimizzare la probabilità che un nodo distribuito, perdendo il contatto con il resto del sistema, possa provocare un malfunzionamento globale.

3 Analisi delle soluzioni

3.1 Gestione del tempo

La competizione dovrebbe essere regolata e scandita da un orologio. La funzione di tale orologio è quella di permettere di tenere traccia della durata degli eventi. È necessario ad esempio ai (thread) concorrenti per decidere quanto attendere all'entrata di un tratto prima che esso si liberi.
L'orologio potrebbe essere di due tipi:

Assoluto :
ovvero un orologio che fa riferimento ad un tempo assoluto. L'orologio dispone di un suo flusso temporale costante interno a cui le entità esterne attingono.

Relativo :
in questo caso non esiste un unico orologio esterno di riferimento. Ogni entità possiede un orologio interno con il proprio istante zero che viene fatto procedere incrementalmente dall'entità stessa.

La prima soluzione non si presta ad un sistema come quello in analisi per i seguenti motivi:

- un orologio assoluto non tiene in considerazione i ritardi dati dalle caratteristiche del sistema su cui il simulatore dovrebbe essere eseguito. Ad esempio non tiene conto del tempo che intercorre da quando un thread è sulla coda dei pronti a quando gli viene assegnata la CPU. Di conseguenza un concorrente (che supponiamo essere rappresentato da un thread) che si annunci pronto ad attraversare un tratto all'istante t, potrebbe (con alta probabilità) iniziare effettivamente ad attraversarlo ad un istante $t+\epsilon$. In questo caso il risultato della simulazione sarebbe errato. Il tempo di attraversamento di tale concorrente, cioè, sarebbe slittato di ϵ.

- il flusso di un orologio assoluto è unico. Ovvero, l'orologio assoluto segue un'unica linea temporale. La natura concorrente del sistema invece vorrebbe che l'orologio segua una linea per ogni thread in esecuzione. Supponendo per esempio che ogni concorrente sia un thread, i concorrenti otterrebbero quanti di esecuzione in modo sequenziale anche se nella realtà simulata starebbero teoricamente svolgendo attività in parallelo. Nel caso quindi due o più thread debbano chiedere l'istante di tempo contemporaneamente, l'orologio verrebbe interrogato in modo sequenziale fornendo due tempi diversi.

I problemi sopra citati sarebbero invece superati implementando tempo in modo relativo. Durante lo svolgimento della competizione, determinati eventi richiedono un preciso intervallo di tempo per essere portati a termine (come l'attraversamento di un tratto da parte di un concorrente). Questi eventi influenzano altri eventi (ad esempio l'attraversamento dello stesso tratto da un altro concorrente). La gestione del tempo in modo relativo vuole, in questo caso, che l'istante in cui un tratto si libera sia dato non dall'istante reale in cui l'ultimo concorrente ha lasciato il tratto (o meglio, in cui l'ultimo thread ha rilasciato la risorsa tratto), ma dall'accumulo degli intervalli temporali richiesti da tutti i concorrenti per attraversarlo (assumendo che il tratto sia a molteplicità uno).
Da questo esempio si evince che la competizione non possa essere regolata da un unico flusso temporale assoluto. Ogni risorsa il cui uso coinvolga fattori temporali (una traiettoria ad esempio), deve possedere un "orologio" interno che venga incrementato di un dato offset ogni volta un evento avvenuto su tale risorsa influenzi i tempi prodotti dai suoi futuri utilizzatori. L'orologio interno chiaramente ha un istante zero uguale a tutti gli orologi presenti nel sistema e non ha vita propria. Cambia cioè solo se qualcuno lo aggiorna.

3.2 Sorpassi impossibili

Il problema dei sorpassi impossibili è dovuto essenzialmente a due fattori:

- natura concorrente del sistema
- risorse condivise (tratti) fra thread concorrenti

Esisterebbe un metodo semplice e veloce di risolvere entrambi i problemi elencati: ridurre tutti i concorrenti ad un unico thread. Dovrebbe cioè essere presente un thread destinato a calcolare lo svolgimento della gara, utilizzando come parametri decisionali le caratteristiche dei concorrenti e i dettagli del circuito. In questo modo sparirebbe il problema delle risorse condivise fra i thread designati a svolgere il ruolo di concorrenti di gara. Tuttavia sarebbe una scelta poco elegante che porterebbe ad annullare più che risolvere le problematiche di concorrenza.
Una soluzione più valida dovrebbe invece mantenere una reale concorrenza fra concorrenti, rispecchiando quanto accade nella realtà. Bisognerebbe quindi istanziare un task per ogni concorrente. Per simulare una gara di formula 1 adeguatamente sarebbe poi necessario fornire una serie di risorse che rappresentino il circuito. Emerge qui il concetto di **tratto**: un singolo segmento di pista. Per rendere la simulazione più realistica, un tratto dovrebbe avere molte-

plicità limitata per permettere ad un numero finito di macchine di attraversarlo contemporaneamente. Per ora quindi le entità usate sarebbero:

- **Concorrente**, entità attiva istanziata una volta per ogni concorrente;
- **Tratto**, entità passiva condivisa fra concorrenti e quindi ad accesso mutuamente esclusivo (con molteplicità arbitraria non infinita);

In questa soluzione ogni concorrente procede verso il tratto N solo dopo aver passato il tratto N-1. Per ottenere il tratto N deve attendere che la risorsa si liberi. Questa bozza di soluzione però espone un problema fondamentale: non essendoci accumulo di risorse, un concorrente che abbia ottenuto un tratto N e debba richiedere l'attraversamento del tratto N+1, dovrà rilasciare N e poi ottenere N+1. Nell'intervallo di tempo in mezzo ai due eventi, il thread che gestisce il concorrente potrebbe essere prerilasciato a far passare avanti altri concorrenti in modo incontrollato. In questo caso si potrebbe facilmente presentare lo scenario esposto durante l'enunciazione del problema nella sezione precedente.
Permettendo l'accumulo di risorse da parte dei thread si presenterebbe la possibilità di stallo. Con un numero di concorrenti pari al numero di segmenti, se ogni concorrente per procedere avesse bisogno del tratto corrente e di quello successivo e se ogni concorrente fosse su un tratto diverso, lo stallo non esiterebbe a presentarsi. Il problema dello stallo verrà comunque trattato più dettagliatamente qualche sezione più avanti.
Per gestire quindi il problema è necessario introdurre delle strutture che permettano di creare una certa dipendenza fra eventi. Più precisamente, bisogna garantire che ogni concorrente sappia quale sia il momento adatto per procedere alla richiesta del tratto successivo senza il rischio di teletrasportarsi davanti ad un altro concorrente o di creare incoerenza temporale. Allo stesso modo, il thread che rappresenta il concorrente deve poter essere prerilasciato senza che ciò implichi un potenziale problema. Come vedremo meglio nella sezione riguardante l'esplicazione della soluzione, il risultato desiderato viene ottenuto regolando l'accesso ai tratti tramite code. Questo, insieme agli orologi relativi, permetteranno ad un concorrente di sapere quando la sua esecuzione non potrà creare conflitti con altri concorrenti.

3.3 Determinismo

Il determinismo della simulazione non può essere ottenuto tramite una precisa idea o soluzione. Il suggerimento, come già enunciato, dovrebbe semplicemente essere che l'architettura e il funzionamento del sistema siano indipendenti da come il sistema operativo sottostante sia stato progettato. Si può già dire che implementando un orologio relativo, parte del non determinismo regalato dallo

scheduler venga eliminato. L'avanzamento il funzionamento dell'orologio relativo dipende esclusivamente dalle scelte progettuali effettuate per il simulatore di formula 1.
Allo stesso modo, l'implementazione di un protocollo di attraversamento dei tratti che segua le direttive suggerite nella sottosezione precedente dovrebbe evitare che la gestione dei processi dello scheduler possa introdurre non determinismo indesiderato.
Rimane ancora il problema del comportamento non prevedibile della rete. Per quanto riguarda i ritardi di rete, devono essere trattati come se tali ritardi fossero quelli introdotti dallo scheduler. Ovvero: il sistema non deve dipendere dalla puntualità delle chiamate remote.
Per quanto riguarda invece i fault di connessione (ad esempio un nodo per qualche motivo smette di essere connesso al sistema), il problema si potrebbe risolvere applicando ridondanza alle componenti remote, ovvero dislocando le stesse componenti su più nodi in modo da attivarne di alternative in caso di fault. Ma si è pensato che l'introduzione di questa caratteristica nel sistema avrebbe richiesto uno sforzo progettuale maggiore rispetto alle pianificazioni. Inoltre, questo tipo di problema potrebbe essere facilmente associato ad un problema esistente anche nella realtà per determinate scelte delle componenti distribuite. Come si vedrà in seguito, infatti, si è scelto di distribuire i box e le TV per la visione della gara. È plausibile che nella realtà una TV venga spenta (chiudendo il contatto con il sistema) o che un box perda il contatto radio con il proprio concorrente. In questi casi i fault vanno gestiti in modo che non risultino in errori durante la competizione.

3.4 Componenti di non determinismo

Come introdotto nel paragrafo 2.4 le componenti di non determinismo possono essere desiderabili se opportunamente gestite, o non desiderabili, se non portano valore aggiunto al prodotto. Nella realizzazione di un simulatore di formula 1 la scelta di introdurre delle componenti che possono modificare l'andamento della gara in maniera non prevedibile prima dell'esecuzione aiuta a simulare in maniera migliore l'andamento di una vera gara di automobili. Le soluzioni possibili che sono state considerate per realizzare il progetto *F1_Sim* sono principalmente tre:

- Nessuna possibilità di componenti di non determinismo
- Possibilità di componenti di non determinismo opportunamente gestite
- Possibilità di componenti che simulino il non determinismo opportunamente gestite

La prima opzione prevede la mancanza di interazione dell'utente che sta visualizzando la simulazione oltre che la mancanza di parti non deterministiche, cioè zone del progetto dove non si ha la padronanza e non si prevede il comportamento. Ad esempio affidarsi al metodo di gestione dei processi dello scheduler per far funzionare la simulazione porta inevitabilmente a problemi di non predicibilità. La seconda opzione permette zone del progetto non controllabili e verificabili mentre la terza opzione permette delle componenti che rendono l'esecuzione più reale e non verificabile a priori ma comunque gestita e controllata già a livello di progettazione. Un esempio di componente è quella che permette l'interazione di un utente che visualizza l'esecuzione del simulatore. Tale azione può provocare, ad esempio, il rientro ai box del concorrente e questo andrà certamente a influire in modo non prevedibile a priori sull'andamento della gara.
Una buona soluzione che può essere adottata è data quindi dalla presenza di possibilità di interazione dell'utente esterno con l'ambiente di simulazione ad esempio permettendo di provocare un pitstop non previsto e di impedire (tramite una buona progettazione) la presenza di componenti di che portano non determinismo al sistema, come ad esempio utilizzare le istruzioni di sleep per creare l'ordine di uscita da un tratto (cioè accodare i concorrenti man mano che vengono risvegliati).

3.5 Stalli

Per quanto riguarda gli stalli si è visto nel paragrafo 2.5 quali siano le condizioni perchè si verifichino. Le soluzioni da adottare sono, semplicemente, l'evitare che si presentino le 4 pre-condizioni simultaneamente non andando incontro, così, al verificarsi di uno stallo. Trattando nel progetto entità attive e passive si possono creare situazioni che, se non controllate, portano inevitabilmente al verificarsi di situazioni di stallo. Le entità attive in un sistema distribuito sono le parti che solitamente utilizzano le entità passive. Uno degli esempi è già descritto nel paragrafo 3.2. e verrà qui approfondito. Se fornissimo la possibilità a un concorrente di accumulare risorse per evitare il problema del teletrasporto (segnalato nel paragrafo 2.2) andremo incontro certamente a una situazione di stallo nonchè alla realizzazione di un sistema che rischia di perdere la concorrenza in quanto, avendo tutte le risorse, il concorrente che può eseguire sulla pista (insieme di tratti, definiti precedentemente) è solo uno. Nel caso non riesca ad ottenere tutta la pista (ad esempio perchè un altro concorrente è stato prerilasciato prima di liberare tutte le risorse) la simulazione rimarrebbe ferma in una situazione di stallo. La soluzione è quindi di studiare e progettare un sistema che preveda la mancanza assoluta di comportamenti anomali (come sorpassi impossibili), dia la possibilità di esecuzione simultanea a più concorrenti

e non presenti per questo situazioni di stallo. Impedire l'accumulo di tratti da parte di un concorrente ed evitare situazioni di attesa circolare sono quindi le basi su cui fondare la soluzione al problema degli stalli.

3.6 Realismo fisico

Come enunciato nella sezione precedente, il realismo fisico di un simulatore di formula 1 è ottenuto dall'interazione concorrente-ambiente statico e concorrente-concorrenti. Verrà ora analizzata la traccia di soluzione per entrambi:

Concorrente - ambiente statico:
: Per ottenere un minimo livello di realismo da questo punto di vista, è necessario fornire dei dettagli riguardanti l'aspetto fisico del circuito. Ogni concorrente, prima di valutare la traiettoria da percorrere per attraversare il tratto, deve poter conoscere gli aspetti caratterizzanti di tale traiettoria, quali ad esempio:

  - lunghezza
  - livello di aderenza
  - angolo

  Queste caratteristiche, unite a quelle dell'auto (benzina disponibile, tipo di gomme ed usura, massima accelerazione ecc.) possono essere utilizzate per dare realismo alla simulazione. Arricchendo quindi la descrizione del circuito e dei suoi tratti con le precedenti caratteristiche, il concorrente potrà valutare per ogni possibile traiettoria la benzina consumata, l'usura delle gomme, la velocità massima raggiungibile.

Concorrente - concorrenti:
: Un livello di interattività fra concorrenti potrebbe essere raggiunto se un concorrente potesse controllare, una volta sul tratto, quali concorrenti siano contemporaneamente presenti sul tratto per poi valutare quale traiettoria scegliere in base all'"affollamento". Esistono chiaramente vari modi di ottenere questo:

  - il concorrente interroga i diversi concorrenti per sapere quale sia la loro posizione sulla pista. In questo modo potrebbe avere una visione di insieme della situazione e valutare quindi la scelta della traiettoria. Per fare ciò, però, sarebbe necessario che l'istante t del concorrente corrisponda all'istante t dei concorrenti interrogati. In questo caso è necessario un orologio assoluto che come visto nella sezione 3.1 è difficilmente realizzabile in un sistema come quello descritto.

Alternativamente ogni concorrente dovrebbe mantenere una storia della sua corsa fino al momento corrente. Una volta interrogato sulla sua posizione all'istante t dovrebbe ripercorrere all'indietro la storia fino a collidere con l'istante richiesto e ritornare l'informazione. Ma una soluzione come questa risulterebbe alquanto inefficiente.

- il concorrente dovrebbe poter sapere solo l'informazione utile per l'attraversamento del tratto corrente. Non è necessario conoscere la posizione di ogni concorrente o quale concorrente si trova su una determinata traiettoria. È sufficiente che si possa sapere in che istante una data traiettoria si liberi. Questa soluzione può essere implementata aggiornando un valore temporale interno alla traiettoria ogni volta che essa venga attraversata, incrementando (o settando) tale valore in base al tempo di attraversamento. Il concorrente potrebbe quindi confrontare l'istante di tempo relativo al suo arrivo con quello di liberazione del tratto per poter poi effettuare una scelta.
  Questa idea è applicabile ad un sistema come quello dato poiché non necessita di un orologio assoluto.

3.7 Gestione delle istantanee di gara

Un'istantanea di gara è lo stato della gara ad un determinato istante di tempo. Intuitivamente l'idea più semplice per ottenere uno snapshot sembrerebbe essere quella di inoltrare la richiesta al sistema all'istante desiderato, mettere in pausa la gara, salvare lo stato e riprendere la competizione dall'istante di pausa. Si riconsideri però la problematica enunciata nella sezione 2.7:
[...]Bisogna però tenere in considerazione che qualunque entità provveda allo scatto dell'istantanea sarà soggetta alle stesse problematiche legate alle altre entità attive dipendenti dallo scheduler[...]
Il "mettere in pausa" la gara, quindi, non è banale. Ogni thread concorrente, ad esempio, potrebbe essere messo in pausa in istanti diversi rispetto al t richiesto, fornendo quindi uno stato non valido rispetto t. Se si considera poi che un orologio assoluto nel sistema si è dimostrato non plausibile da implementare, l'istante t richiesto per la pausa non corrisponderebbe ad un'istante di tempo assoluto di pausa per la gara. All'interno della competizione le singole entità (che ne necessitano) seguono un orologio interno.
È quindi opportuno che, una volta richiesto lo snapshot di un istante temporale, lo stato di ogni entità (utile all'istantanea) in quell'istante sia disponibile o, alternativamente, l'unità richiedente si metta in attesa fino alla disponibilità completa dello stato richiesto. Bisognerà tenere una storia della gara che permetta di risalire allo stato globale ad un dato istante.

3.8 Problematiche di distribuzione

Per quanto riguarda la robustezza del sistema distribuito le soluzioni possibili variano in base alla scelta dello standard di comunicazione, della scelta del tipo di dati che viaggia nella rete e dal livello di distribuzione scelta per il prodotto finale. Per quanto riguarda il tipo di middleware di comunicazione la scelta potrà cadere tra

- CORBA
- Distributed System Annex
- MOM

oppure con la costruzione ad hoc di un middleware di comunicazione scritto su misura per la nostra applicazione. La scelta dei dati da trasferire fra i vari nodi della rete dipende fortemente dal tipo di sistema di comunicazione che si sceglie, anche se può comunque essere orientata verso un utilizzo prevalente delle stringhe o di tipi primitivi o di una combinazione dei due. Il livello di distribuzione va deciso in base alle funzionalità e alle caratteristiche che si vogliono dare al prodotto in uscita. Si può pensare di spingere al massimo la distribuzione mettendo ogni entità in nodi separati oppure di ridurre al minimo la distribuzione mettendo solo degli schermi che visualizzino l'andamento della competizione. Ovviamente si può anche scegliere una mediazione fra le due soluzioni.Un esempio è dato dalla scelta di distribuire componenti che visualizzino l'andamento della gara (e possano anche interagire, come i box) e lasciare centralizzata la gara vera e propria, quindi con le entità attive e passive situate nella stessa macchina. Altra scelta fondamentale che va pensata e decisa in fase di progettazione è quella del tipo di chiamate da effettuare, se sincrone o asincrone, considerandone i vantaggi e gli svantaggi. Per quanto riguarda questo simulatore di formula uno concorrente e distribuito le chiamate sincrone forniscono uno strumento adatto e semplice per la sincronizzazione fra oggetti remoti. Inoltre se opportunamente gestite forniscono risultati senza la presenza di polling o altri meccanismi di recupero dei parametri o dei valori di ritorno.

3.9 Avvio del sistema

Indipendentemente da quali saranno, nel dettaglio dell'implementazione, le relazioni fra entità e il livello di distribuzione delle unità componenti il sistema, l'avvio del sistema dovrebbe seguire, per i due problemi enunciati alla sezione 2.9, le seguenti linee guida:

- Assumendo l'esistenza di un nodo centralizzato che ospiterà la maggior parte del calcolo computazionale legato alla gara e un numero di nodi

distribuiti che interagiranno con esso (anche bidirezionalmente), le possibilità per la messa in connessione dei nodi sono 2:

  - Il nodo centrale dispone già di una lista di indirizzi da utilizzare per contattare i nodi remoti. Ma questo richiederebbe che nella fase di init o tutti i nodi (come precondizione) siano già avviati, o che l'unità centrale effettui polling verso tutte le entità remote fino al loro avvio o, infine, che il nodo centrale si occupi anche dell'avvio dei nodi remoti.
    La prima idea richiede un'assunzione troppo forte: con un numero elevato di nodi remoti potrebbe diventare operazione lunga sapere quando tutti siano pronti. La seconda comporta uno certo spreco computazionale: un polling sequenziale dei nodi potrebbe far attendere per un certo intervallo di tempo un nodo pronto (nel caso venga avviato subito dopo essere stato interrogato). Con un insieme di polling paralleli il problema si risolverebbe, ma il sistema sarebbe comunque poco elastico se inaspettatamente uno dei nodi previsti dovesse cambiare locazione. L'ultima idea potrebbe essere una soluzione elegante, ma espone il problema (presente anche nelle altre due idee di soluzione) che se le entità remote venissero spostate, la lista dovrebbe essere aggiornata di volta in volta.
    Sia chiaro che questo non è un problema non risolvibile in alcun modo, ma si pensa potrebbe richiedere uno sforzo progettuale maggiore del previsto.
  - Il nodo centrale viene avviato e rimane in attesa che i nodi remoti lo contattino fornendo le proprie informazioni di posizionamento. In questo modo i nodi remoti, una volta avviati, dovrebbero conoscere l'indirizzo del nodo centrale e contattarlo. Così facendo, una volta che tutti i nodi necessari all'avvio della competizione saranno inizializzati, il nodo centrale avrà già a disposizione gli indirizzi per la comunicazione bidirezionale con la certezza che essi siano avviati. I nodi remoti potrebbero comunque dover effettuare polling in attesa che il nodo centrale sia avviato.

- Per gestire l'avvio delle entità concorrenti, una soluzione plausibile è quella di organizzare l'inizializzazione del sistema a step. Fra le entità intercorrono varie relazioni di dipendenza. Alcune necessitano di dati ottenibili a partire da altre, le quali a loro volta possono aver bisogno di accedere a risorse disponibili solo ad un determinato momento della fase di avvio. Per modellare la soluzione è quindi necessaria una classificazione delle componenti in base alle risorse richieste. Il primo step dell'avvio dovrà

occuparsi di inizializzare le risorse (attive, passive o reattive che siano) che non richiedano dati decisi a tempo di esecuzione. Durante gli step successivi si dovranno man mano avviare le risorse (o parti di risorse) che dipendono dai dati precedentemente inizializzati. Se un'entità attiva necessita di dati ottenibili in due o più step diversi per poter considerarsi pronta, sarà necessario che di step in step venga messa in attesa di tali dati. Questo può avvenire mettendo l'entità in attesa su una risorsa (finché questa non raggiunga uno stato adeguato per fornire i dati richiesi) oppure mettendo l'entità stessa in attesa di messaggi (che potrebbero contenere i dati richiesti o semplicemente segnalare che è possibile procedere). Come post-condizione alla fine dell'ultimo step è necessario che tutte le entità passive e reattive che verranno richieste dopo l'init siano disponibili e che le risorse attive siano pronte ad iniziare con la reale esecuzione.

3.10 Stop del sistema

Si è visto che lo stop del sistema è suddiviso in uno stop riferito alla fine della gara ed uno legato alla finalizzazione delle risorse allocate. Di seguito i dettagli:

- Per lo stop logico si potrebbe delegare alle entità attive che gestiscono la corsa dei singoli concorrenti l'onere di tenere conto dello stato della gara (il numero di giri effettuati ad esempio) e decidere, al momento opportuno, di procedere con lo stop di quelle che finiscono la competizione (o per numero di giri effettuati o per impossibilità a continuare).
- La finalizzazione delle risorse potrebbe avvenire anche prima che la competizione finisca se l'utente decide di interrompere anticipatamente l'esecuzione del simulatore. Di conseguenza, per permettere uno stop definitivo del sistema, sarà necessario che il processo del simulatore venga terminato nel momento in cui l'utente chiuda l'applicazione.

4 Soluzioni adottate

4.1 Entità Circuito

Nella soluzione proposta il circuito è un'unità composta da più entità passive ad accesso mutuamente esclusivo. Più precisamente:

- **Path**: rappresenta la traiettoria percorsa da un concorrente. Un insieme di **Path** costituisce un tratto e il loro numero all'interno del tratto identifica le possibili vie che si possono intraprendere per attraversarlo. Ogni path ha una molteplicità limitata da

lunghezza / 4.5
dove 4.5 è lo spazio longitudinale richiesto da un'auto di formula 1 per non entrare in collisione con altre auto. Per "*molteplicità* si intende il numero di auto che possono restare in fila all'interno di una traiettoria. Il **Path** è caratterizzato da:

- lunghezza
- angolo
- tenuta
- istanti di liberazione, ovvero gli istanti di tempo in cui le ultime N auto che sono entrate nella traiettoria (con N pari alla molteplicità del **Path**) sono previste uscirne;
- istante di liberazione, ovvero l'istante in cui il tratto è previsto essere libero da tutte le auto;
- massima velocità raggiungibile, determinata dal concorrente che è entrato più recentemente nel **Path**. Questo valore viene usato per limitare la velocità di un concorrente nel caso esso entri quando altre auto lo stanno attraversando.

I **Path** di uno stesso tratto possono essere uguali o differire per caratteristiche, rimanendo comunque entro i limiti globali del tratto (non si potrà ad esempio avere un **Path** lungo 11 km in un tratto lungo 42 m).

- **Checkpoint**: è una risorsa passiva ad accesso mutuamente esclusivo. Rappresenta la coda di accesso ad un tratto. Ogni posizione della coda è stata arricchita con delle informazioni che aiutano a gestire l'accesso al tratto:
  - **istante di arrivo**: l'istante di tempo più ottimista in cui l'auto è prevista arrivare oppure l'istante in cui l'auto realmente arriva al tratto (in base al valore della flag "arrivato", descritta in seguito);
  - **id concorrente**: l'id del concorrente presente nella posizione della coda;
  - **flag "arrivato"**: se valorizzata con *true*, significa che il concorrente sta effettivamente attendendo di accedere al tratto e che il valore temporale segnato nell'**istante di arrivo** è l'istante di arrivo effettivo. Altrimenti significa che il concorrente non è ancora arrivato ma arriverà ad un istante maggiore o uguale a quello segnato nell'**istante di arrivo**.

  Ogni **Checkpoint** inoltre punta ad un insieme di **Path**.

- **Racetrack iterator**: ogni concorrente dispone di un'istanza di questo iteratore per poter navigare il circuito e richiedere di accedere eventualmente ai box.

La struttura è stata pensata principalmente per venire a capo del problema dei sorpassi impossibili.
Come visto nella sezione 3.1, non esiste un orologio assoluto. Ogni concorrente segna sulla coda il tempo di arrivo in base ai tempi accumulati per l'attraversamento dei tratti precedenti. Quindi si può dire che ogni concorrente aggiorni un proprio orologio relativo all'andamento della sua corsa. Inoltre ogni concorre segna un tempo ottimistico di arrivo sui checkpoint che (salvo ritiro) attraverserà. Tale istante sarà di sicuro maggiore dell'istante segnato nel **Checkpoint** in cui il concorrente effettivamente è arrivato. Questo permette ad ogni concorrente di sapere, quindi, non solo i dettagli relativi alla sua corsa, ma anche quelli relativi agli altri concorrenti (quando necessario). Di conseguenza, se un concorrente è interessato ad attraversare un tratto, potrà confrontare il suo tempo di arrivo effettivo con i tempi di arrivo effettivi e previsti degli altri concorrenti. Nel momento in cui il suo istante di arrivo effettivo risulti cronologicamente il più basso della coda, sarà certo che, stando al suo istante di tempo e a quello relativo agli altri concorrenti, il suo turno per attraversare sia arrivato (verrà dimostrato formalmente in seguito). Ovvero che, in termini di tempo relativi, il suo istante, essendo il più basso, indica che è il primo arrivato al tratto in quell'istante, avendo così il diritto di accesso.
Quando un concorrente deve valutare la traiettoria (**Path**) da attraversare in un tratto, può farlo semplicemente valutando le caratteristiche del tratto. Le prime tre elencate qualche riga più sopra garantiscono un minimo di realismo fisico nell'interazione concorrente-ambiente statico. Le caratteristiche fisiche del tratto infatti influiranno sui consumi e i tempi dell'auto. Il concorrente dovrà scegliere in modo da minimizzare entrambi.
Gli ultimi parametri sono anche oggetto di valutazione in quanto servono all'utente per verificare lo stato di occupazione del tratto.
Un concorrente può certamente decidere di attraversare il tratto utilizzando una traiettoria su cui sia già presente un altro concorrente. Ma ne conseguirà che se il concorrente più veloce è dietro, esso verrà limitato dalla velocità di quello davanti.
Altra questione importante riguarda i tempi di liberazione: come già detto, una traiettoria può essere contemporaneamente occupata da al più N (con N limitato) macchine. Quando un concorrente inizia a percorrere una traiettoria, dovrà salvare in una posizione libera della lista dei tempi di liberazione il proprio istante di uscita dal tratto. Quando un concorrente vuole iniziare a percorrere

una traiettoria, di conseguenza, dovrà verificare che il suo tempo di arrivo sia maggiore almeno di uno dei tempi di liberazione segnati nella lista. Solo in questo caso potrà essere certo che la traiettoria può ospitare auto. Se tutte le traiettorie di un tratto sono sovraffollate, il concorrente dovrà uscire di pista per evitare una collisione e si dovrà quindi ritirare.
Questo garantisce il realismo fisico concorrente-concorrenti descritto nella sezione 3.6. I tempi segnati nelle code invece mettono in pratica, in parte, il concetto di orologio relativo enunciato nella sezione 3.1. Maggiori dettagli sull'interazione fra concorrenti e circuito e sull'algoritmo che regola l'attraversamento del circuito da parte dei concorrenti verranno forniti nelle sezioni seguenti.

4.2 Entità Concorrente

Il concorrente è un'entità attiva, concepita come un task che utilizza l'entità **Circuito** per iterare sulla pista. In seguito verranno analizzate le interazioni che questa entità hanno fra di loro e con le entità passive che compongono il circuito. Questa componente è composta principalmente da tre parti che sono:

- Car : oggetto che rappresenta l'auto con tutte le caratteristiche
- Driver : oggetto che rappresenta il pilota
- Competitor_Details : oggetto che rappresenta il concorrente nel suo insieme, ovvero auto, pilota e componenti di strategia e di monitoraggio

4.3 Interazione concorrenti - circuito

I concorrenti (entità attive) utilizzano il circuito per gareggiare nella competizione. Ogni concorrente per gareggiare deve riuscire a ottenere il tratto di pista che vuole attraversare. La struttura di chiamate (insieme alla struttura del circuito già descritta nel paragrafo 4.1) è stata studiata per evitare le possibili situazioni di stallo e di sorpasso impossibile oltre che mantenere coerenza temporale nel corso della gara. Ogni concorrente utilizza un iteratore (**Racetrack_Iterator**) per iterare attraverso i tratti della pista. Grazie ad esso il concorrente può ottenere il **Checkpoint** del circuito nell'ordine stabilito dalla configurazione. Può inoltre ottenere informazioni relative al tratto di entrata e uscita ai box, oltre che ottenere il **Checkpoint** relativo ai box.
Come visto nella sezione 4.1, i checkpoint espongono delle code per gestire l'accesso al tratto. Per attraversare un tratto di pista il concorrente dev'essere il primo nella coda. Una volta che un concorrente si trova nella prima posizione della coda può utilizzare la risorsa passiva in mutua esclusione che rappresenta il pezzo di pista. Le operazioni di calcolo dei tempi e di scelta delle traiettorie

avvengono mantenendo il possesso della risorsa, con la certezza che nessuno modifichi le condizioni che si stanno valutando. Si evita inoltre che un concorrente possa valutare l'attraversamento del tratto quando spetterebbe (per motivi temporali) ad un altro concorrente.
Successivamente verrà spiegato più formalmente quanto detto.
Vediamo ora l'algoritmo per l'attraversamento di un tratto dal momento in cui il concorrente arriva fisicamente ad un checkpoint fino al raggiungimento di quello successivo.
Per chiarezza, la spiegazione verrà scomposta in due parti che poi, integrate, formeranno l'algoritmo completo.

Scelta traiettoria (Path):

In questo passo della simulazione, il concorrente ha già ottenuto accesso esclusivo alla collezione di **Path** e potrà quindi effettuare la valutazione senza rischio di race condition (il perché verrà spiegato successivamente).
L'algoritmo procede quindi nel seguente modo:

- Per ogni traiettoria (**Path**) a disposizione:
  - Calcola il tempo di attraversamento (in assenza di auto sulla traiettoria);
  - Confronta l'istante di uscita dato con tempo di attraversamento calcolato con l'istante di liberazione segnato sul tratto;
  - Se l'istante di liberazione è inferiore (il che significa che il tratto è già libero quando l'auto proverà ad attraversarlo), salva l'istante di uscita associato al **Path** come *istante di arrivo* + *tempo di attraversamento*;
  - Se l'istante di liberazione invece è superiore, verifica se ci sia spazio per entrare nella traiettoria. Per fare ciò, bisogna considerare la lista di istanti di liberazione. Se tutti gli istanti segnati sono maggiori dell'*istante di arrivo* del concorrente, significa che non sarà possibile usare quella traiettoria.
    In caso contrario, il concorrente può attraversare mettendosi dietro alla/e auto già presente/i nella traiettoria. Per questo la velocità sarà limitata da quella dell'auto davanti e l'istante di uscita associato al path corrisponderà all'istante di uscita della macchina entrata più di recente (ovvero *l'istante di liberazione del tratto*) + *il tempo che rappresenta la distanza minima fra due auto per evitare tamponamenti* (costante).
- Se almeno un **Path** è percorribile,
  - scegli quello ottimo (con istante di uscita minore);

- aggiorna il tempo di liberazione del tratto con quello appena calcolato;
- aggiorna la lista degli istanti di liberazione aggiungendo quello nuovo in una posizione il cui istante segnato sia inferiore all'istante di arrivo del concorrente (significa che quell'istante è già passato e non è più utile ai fini della valutazione dell'affollamento della traiettoria);

- Altrimenti l'utente deve uscire dal circuito per evitare collisioni, dovendo così ritirarsi.

Attraversamento tratto:

Premessa: un concorrente arrivato effettivamente su una coda ma non primo, si mette in attesa. Verrà risvegliato automaticamente dall'ultimo concorrente che avrà **modificato l'ordine** della coda nel caso risultasse primo dopo l'aggiornamento.
Precondizione: il concorrente ha già segnato il suo tempo di arrivo effettivo sulla coda del checkpoint che vuole attraversare.

1. il concorrente setta la flag "arrivato" nella coda del checkpoint sulla posizione a lui riservata;
2. controlla in che posizione è nella coda;
3. se non è primo, attende di essere primo;
4. quando è primo, richiede l'insieme di traiettorie (**Path**) da cui è costituito il tratto;
5. Calcola la traiettoria (**Path**) di attraversamento ottimale secondo l'algoritmo spiegato precedentemente. Si otterrà quindi un istante di uscita, che corrisponderà al tempo di arrivo effettivo al checkpoint successivo;
6. segna il tempo di arrivo effettivo sulla coda del checkpoint successivo, che verrà **riordinata** in base al nuovo tempo.
7. segna il tempo di arrivo limite su tutte le altre code (aggiungendo il minimo tempo richiesto dall'auto per andare da un checkpoint a quello successivo), da quella successiva al checkpoint che si sta per raggiungere a quella del checkpoint appena attraversato. Ad ogni aggiornamento del tempo limite, la coda viene **riordinata**.
8. Se nella coda del checkpoint l'istante di arrivo limite del concorrente risulterà maggiore di qualche altro, verrà riordinato in una posizione inferiore, lasciando eventualmente spazio ad altri di effettuare valutazioni su quel tratto.

9. delay per fini di simulazione;
10. ritorna al passo 1 per il checkpoint successivo;

Il diagramma di sequenza B.2.1 in appendice esplica a livello implementativo come l'attraversamento avviene. La strategia adottata si presta a simulare la corsa garantendo che un concorrente possa tenere in considerazione ad ogni istante le auto presenti sulla pista e sui tratti senza comportamenti anomali. È necessario, tuttavia, dimostrare che l'algoritmo sia corretto. Per farlo bisogna provare che i concorrenti non collidano (logicamente) nel corso dell'esecuzione. Ovvero:

- L'aggiornamento delle code sia mutuamente esclusivo
- La valutazione di un tratto sia effettuata in modo esclusivo;

Dimostriamo che quanto detto accade:

- *quando un concorrente valuta le traiettorie, lo sta facendo in modo esclusivo.* Perché due concorrenti x e y possano iniziare a valutare le traiettorie del tratto n contemporaneamente, deve verificarsi una delle seguenti due condizioni:
  Condizione 1:
  - entrambi i concorrenti sono effettivamente presenti sulla coda del **Checkpoint** e in attesa di attraversare;
  - i due concorrenti x e y, ordinati nella coda per tempo di arrivo crescente, hanno assegnati rispettivamente i tempi t_x e t_y, con $t_x < t_y$; per come è definito l'algoritmo, finché la disposizione è tale, y non potrà effettuare valutazioni sul tratto;
  - mentre x sta valutando la traiettoria da percorrere si ha che $t_y < t_x$ e y inizia quindi a valutare;

  Condizione 2:
  - il concorrente x risulta primo effettivo sulla coda del **Checkpoint** n;
  - i due concorrenti x e y, ordinati nella coda per tempo di arrivo crescente, hanno assegnati rispettivamente i tempi t_x e t_y, con $t_x < t_y$;
  - quindi y risulta in posizione inferiore ad x sullo stesso **Checkpoint**, ma non è effettivamente arrivato;
  - x inizia la valutazione del tratto n;
  - y cambia il suo istante di arrivo sul tratto n facendo risultare $t_y < t_x$;

– prima che x finisca di valutare il tratto o aggiornare i tempi nei **Checkpoint**, y arriva effettivamente sul tratto e come da algoritmo inizia a valutarlo;

Queste situazioni non si possono verificare e verrà ora dimostrato perché:

Dimostrazione. **Condizione 1**
Si è detto che la coda viene riordinata ogni qualvolta venga inserito un nuovo valore al suo interno o venga cambiato un'istante temporale relativo ad un concorrente. Si premette che, perché l'ordine tra x e y venga cambiato, o x o y devono effettuare un cambiamento. Un qualunque concorrente z che applichi modifiche alla coda cambierà l'ordinamento senza alterare l'ordine tra x e y.
Quando x sta eseguendo la valutazione, il concorrente y è bloccato, non potendo quindi in alcun modo cambiare l'ordinamento della coda.
Nel momento in cui x finisce di valutare, itera su tutte le code da $n+1$ a n per aggiornare i tempi limite di arrivo.
n, che è il **Checkpoint** dove y sta attendendo, viene modificata per ultima. A questo punto viene riordinata.
Nella peggiore delle ipotesi y risulta prima e il processo che gestisce il concorrente viene assegnato alla CPU immediatamente.
Non si presentano anomalie poiché x ha analizzato i **Path** e aggiornato il tempo di uscita su quello scelto. □

Per arrivare a contraddire lo scenario presentato nella condizione 2, è necessario dimostrare che una volta che un istante temporale viene segnato sulla coda di un **Checkpoint**, non può accadere che possa essere aggiornato con un'istante antecedente. In questo modo non sarebbe possibile per y aggiornare il tempo sulla coda n in modo da finire prima di x.

Dimostrazione. **Condizione 2**
Assumiamo un circuito a S checkpoint, con t_n l'istante di partenza per un concorrente generico y al tratto n. Il **Checkpoint** n avrà quindi nella coda associata alla posizione dedicata a y l'istante di tempo t_n. La posizione associata a y nelle code dei checkpoint successivi avrà segnato l'istante $t_n + \epsilon$ con ϵ monotonicamente crescente di checkpoint in checkpoint. Questo istante iniziale subirà le seguenti modifiche nel corso della competizione:

– Dopo l'attraversamento di un tratto, verrà sommata una quantità di tempo δ_t che rappresenta il tempo di attraversamento. Per ogni tratto attraversato, quindi, t_n verrà incrementato secondo una funzione crescente;

- Dopo aver segnato il tempo di arrivo effettivo sul checkpoint *n+1*, verranno aggiornati i tempi nei checkpoint successivi dello stesso ϵ crescente utilizzato all'inizio.
- Di conseguenza, quando il concorrente passerà al tratto *n+1*, segnerà un tempo $t_{n+1} = t_n + \delta_t > t_n$;
  nei checkpoint successivi verrà segnato un tempo $t_k = t_{k-1} + \epsilon > t_{k-1}$.
- Considerando che il primo tratto aggiornato con ϵ sarà t_{n+2}, si avrà $t_{n+2} = t_{n+1} + \epsilon > t_{n+1}$, e così per ogni k = *n+2, ...,s, 1, ...,n.*

Ne consegue che ogni volta che un concorrente segna un istante temporale, esso sarà maggiore di quello segnato precedentemente sul **Checkpoint**. □

- *non possono esserci teletrasporti.*
  Si premette che un teletrasporto avviene al verificarsi di queste tre condizioni:
  - due concorrenti x e y arrivano al checkpoint n ed entrano nella stessa traiettoria (**Path**);
  - l'istante di entrata per i due (segnato nel checkpoint n) è rispettivamente tx_n e ty_n con $tx_n < ty_n$ (banalmente);
  - l'istante di entrata segnato al checkpoint successivo (che corrisponde all'uscita dalla traiettoria del checkpoint precedente) è rispettivamente tx_{n+1} e ty_{n+1} con $ty_{n+1} < tx_{n+1}$;

  In questo caso è chiaro che sia avvenuto un teletrasporto, in quanto il concorrente x, stando all'ordine di entrata nella traiettoria, avrebbe dovuto arrivare per ragioni fisiche prima del concorrente y al checkpoint successivo. Nella peggiore delle ipotesi il concorrente y avrebbe dovuto rallentare ed arrivare al checkpoint *n+1* subito dopo x. Dimostriamo che questa situazione non si può verificare. Definiamo prima di tutto i seguenti simboli aggiuntivi:
  - $CheckIn_{(x,n)}$: tempo di arrivo effettivo di x al checkpoint n;
  - $CrossingTime_{(x,n,z)}$: tempo di attraversamento della traiettoria n per il concorrente x supponendo usi la traiettoria z;
  - $PathAvailableInstant_{(n,z)}$: istante di uscita dell'ultimo concorrente entrato della traiettoria z al tratto n ;
  - $ExitInstant_{(x,n,z)}$: istante di uscita di x dalla traiettoria z del tratto n;

- t_c : è una costante che rappresenta la distanza temporale minima fra due macchine per evitare tamponamento;

Assumiamo la presenza di due concorrenti x e y. Dimostriamo che non possono verificarsi teletrasporti se entrambi prendono la stessa traiettoria

Dimostrazione.
- x ha segnato sul checkpoint n il tempo t_x;
- y ha segnato sul checkpoint n il tempo t_y;
- quindi $CheckIn_{(x,n)} = t_x$ e $CheckIn_{(y,n)} = t_y$.
  Assumiamo $CheckIn_{(x,n)} < CheckIn_{(y,n)}$ per come i due concorrenti ipoteticamente sono arrivati al checkpoint;
- x è il primo ad attraversare. Mentre analizza le traiettorie si è visto che y non può eseguire operazioni sullo stesso tratto. Assumiamo che la traiettoria scelta sia 1;
- il tempo per andare dall'altra parte corrisponderà all'istante che verrà segnato nel checkpoint n+1. Quindi, per l'algoritmo visto,
  $ExitInstant_{(x,n,1)} = max\{PathAvailableInstant_{(n,1)}+t_c, CheckIn_{(x,n)}+ CrossingTime_{(x,n,1)}\}$
  assumendo che la traiettoria fosse libera, segue che
  $CheckIn_{(x,n+1)} = CheckIn_{(x,n)} + CrossingTime_{(x,n,1)}$;
- x aggiornerà l'istante di liberazione del **Path** nel modo seguente:
  $PathAvailableInstant_{(n,1)} = CheckIn_{(x,n)} + CrossingTime_{(x,n,1)}$;
  quindi avremo che
  $CheckIn_{(x,n)} < CheckIn_{(x,n+1)}$ con $CheckIn_{(x,n+1)} == PathAvailableInstant_{(n,1)}$
- come da algoritmo, x aggiornerà tutte le code dei checkpoint da n+1 a n (passando per il checkpoint 1) con tempi $CheckIn_{(x,n+1)} + \epsilon$ con ϵ crescente di checkpoint in checkpoint. Supponiamo che il tempo segnato sul checkpoint n sia sufficientemente alto da mandare x in una posizione posteriore a y, così che y possa effettuare l'attraversamento;
- dopo che y avrà computato i calcoli necessari per l'attraversamento del tratto, si avrà che:
  $ExitInstant_{(y,n,1)} = max\{PathAvailableInstant_{(n,1)}+t_c, CheckIn_{(y,n)}+ CrossingTime_{(y,n,1)}\}$;
  Se il valore massimo verrà rappresentato da $PathAvailableInstant_{(n,1)}+ t_c$, si avrà che
  $ExitInstant_{(y,n,1)} == CheckIn_{(x,n+1)} + t_c \rightarrow CheckIn_{(y,n+1)} == CheckIn_{(x,n+1)} + t_c$
  ne consegue che
  $CheckIn_{(y,n+1)} > CheckIn_{(x,n+1)}$, quindi non verrà effettuato alcun teletrasporto.

– Se invece il valore massimo era rappresentato da
$CheckIn_{(y,n)} + CrossingTime_{(y,n,1)}$, allora
$CheckIn_{(y,n)} + CrossingTime_{(y,n,1)} > PathAvailableInstant_{(n,1)} \rightarrow$
$CheckIn_{(y,n)} + CrossingTime_{(y,n,1)} > CheckIn_{(x,n+1)}$
e, con l'assegnamento $ExitInstant_{(y,n,1)} = CheckIn_{(y,n)} + CrossingTime_{(y,n,1)}$
si avrà
$CheckIn_{(y,n+1)} = CheckIn_{(y,n)} + CrossingTime_{(y,n,1)}$
$\rightarrow CheckIn_{(y,n+1)} > CheckIn_{(x,n+1)}$.

□

Nel caso i due concorrenti prendano due traiettorie differenti, non si potrebbe più parlare di teletrasporto, bensì di sorpasso.

Del rientro ai box non se ne è ancora discusso per evitare di rendere più confusionaria la spiegazione. Ne vengono forniti i dettagli nella sezione seguente.

4.3.1 Corsia dei box

L'entrata ai box è gestita in modo leggermente diverso rispetto a come si è gestita l'entrata in ogni tratto. Innanzitutto è necessario premettere le caratteristiche della corsia dei box:

- è possibile entrare nella corsia dei box solo in prossimità del **Checkpoint** corrispondente al **preBox** (proprietà booleana dei **Checkpoint**); tale checkpoint punterà quindi a due tratti: uno riferito al tratto successivo del circuito, l'altro riferito alla corsia dei box;
- quando si esce dai box, deve ottenere accesso al tratto preceduto dal **Checkpoint exitBox** (altra proprietà booleana dei **Checkpoint**);
- il box stesso è un checkpoint;

Il checkpoint del box è posizionato in linea con il goal. Di conseguenza se un concorrente rientra ai box durante l'ultima lap, finirà la gara ai box. Il checkpoint dei box è stato introdotto proprio per fornire questa possibilità.
Il tempo di permanenza ai box viene fornito dal box remoto (il quale si occupa di calcolare i tempi per il cambio gomme e il rifornimento), e viene aggiunto al tempo che verrà segnato sul checkpoint di uscita dai box.
Il tempo di attraversamento dei tratti relativi ai box è limitato superiormente a 80 km/h. Quindi si suppone che auto che entreranno insieme non proveranno comunque a sorpassarsi. L'unica possibilità di sorpasso è data dai tempi di rifornimento che possono creare più o meno ritardo sul tempo di percorrenza della corsia successiva.

4.3.2 Assenza di stallo

Il verificarsi dello stallo, nello specifico caso, sarebbe dato dal presentarsi delle seguenti condizioni, determinate dall'accumulo di risorse dei concorrenti che gareggiano nel circuito:

- un concorrente x è arrivato effettivamente sulla coda associata al **Checkpoint** n ed è in attesa di essere primo per poter valutare il tratto;
- davanti a x è segnato (ma non arrivato effettivamente) y, con un tempo di arrivo previsto, quindi, inferiore a x. Si può quindi dire che y sia in possesso della risorsa **Checkpoint** n, pur non usandolo direttamente;
- nel tratto *n+1*, x è primo non effettivo, avendo quindi in possesso la risorsa **Checkpoint** *n+1* pur non ancora usandola.
- dietro a x nel tratto *n+1* è presente y, arrivato effettivamente e in attesa di attraversare.

Si nota quindi come x e y possiedano entrambi una risorsa (poiché ne impediscono l'uso ad altri concorrenti) ma nel frattempo richiedano altre risorse per poter eseguire. In questo caso y e x resterebbero in attesa infinita. Dimostriamo che tale scenario non è possibile.

Dimostrazione.
Premessa: $t_{x,n} \rightarrow$ *tempo del concorrente* x *segnato sul checkpoint* n+1.
Se nella coda *n+1* ci sono x in testa seguito da y, avremo che $t_{x,n+1} < t_{y,n+1}$, poichè l'ordine della coda è determinato dai tempi di arrivo segnati. Se x è in attesa sulla coda *n+1*, vuol dire che all'iterazione precedente aveva aggiornato per ultimo l'istante segnato sulla coda del **Checkpoint** n e di conseguenza $t_{x,n} > t_{x,n+1}$ (come visto precedentemente). y invece è in attesa effettiva sulla coda n, quindi avrà aggiornato la coda del tratto successivo con un valore dato da $t_{y,n+1} + \delta_t$, con δ_t un valore minimo diverso da zero previsto per l'attraversamento da n a *n+1*. Quindi $t_{y,n} < t_{y,n+1}$. L'ipotesi di stallo prevede che nella coda *n+1* vi sia primo il concorrente y e poi x, quindi $t_{y,n+1} < t_{x,n+1}$. Avremo quindi:
$t_{x,n+1} < t_{x,n} < t_{y,n} < t_{y,n+1} < t_{x,n+1}$

che è una contraddizione, quindi la situazione non si può verificare. □

4.4 Interazione concorrenti - concorrenti

All'interno del progetto non è prevista una comunicazione diretta fra concorrenti. Esiste però un metodo per i concorrenti di capire dove sono gli altri

concorrenti in gara. I concorrenti interagiscono tra di loro ogni volta che attraversano un tratto, infatti ogni concorrente che esegue lascia segnato il proprio istante di liberazione della traiettoria percorsa. Inoltre ogni concorrente valorizza una posizione di un'array associato al **Path** con il proprio istante di uscita. Essendo l'array grande quanto il numero massimo di concorrenti che possono restare sul traiettoria in fila, può essere usato per capire se il tratto sia pieno o no. Queste informazioni salvate sul **Path** indicano agli altri concorrenti di valutare il livello di affollamento del tratto e valutare opportunamente i tempi di attraversamento. Maggiori dettagli riguardo all'entità tratto sono presenti nella sezione 4.1. Inoltre i concorrenti hanno la possibilità di valutare i tempi di arrivo ai checkpoint degli altri ispezionando la coda associata ad ogni **Checkpoint**.

4.5 Entità box

L'entità Box è una delle componenti che si è scelto di distribuire nella rete. Il Box si occupa di gestire la configurazione e la corsa di un concorrente. Durante la competizione, il Box verifica costantemente lo stato dell'auto e fornisce eventuali cambi di strategia se ritenuto opportuno. Inoltre decide quando giunge il momento del pitstop e aggiorna di conseguenza le impostazioni della macchina, ovvero benzina nel serbatoio e gomme nuove. Ogni Box è caratterizzato da uno fra 4 tipi di strategia, diversi per grado di "ottimismo" nelle valutazioni e nei calcoli dati lo stato della macchina, le medie calcolate e lo stile di guida del concorrente:

1. Cautious: cauto, sottostima il numero di giri ancora fattibili;
2. Normal: stima abbastanza realistica delle possibilità del concorrente, considera anche un margine di errore nei calcoli per effettuare una valutazione;
3. Risky: le stime vengono effettuate in base a calcoli esatti che di solito non tengono in considerazione fattori che nella realtà possono incidere in modo negativo;
4. Fool: nella realtà normalmente non si arriva a tanto, ma per fini di test è stato inserito anche un tipo di strategia che sovrastima le possibilità del concorrente, portandolo a ritiro quasi certa.

4.5.1 Interazione con il concorrente

L'entità box interagisce costantemente con il concorrente, anche se non in modo diretto. Il box suggerisce al concorrente durante la gara :

- stile di guida. Verrà suggerito uno stile più conservativo se i consumi si sono rivelati maggiori del previsto e viceversa;

- numero di lap al pitstop

Il Box riceve informazioni sullo stato del concorrente alla fine di ogni settore dal monitor di gara (il quale attinge le sue informazioni dalla componente dedicata all'accumulo delle statistiche, per dettagli vedere 4.6.1). Ad ogni aggiornamento, aggiorna delle statistiche interne. Viene ricalcola la strategia alla fine del secondo settore. Il concorrente richiede la nuova strategia al box in prossimità del checkpoint dove è possibile proseguire o andare ai box. E sembrata più realistica la scelta di non calcolare la strategia alla fine del terzo settore, perché si suppone che nella realtà non si possa essere così veloci da calcolare una nuova strategia istantaneamente alla fine del circuito con i dati del terzo settore. È piuttosto più probabile che qualunque cambio di strategia o richiesta di rientro ai box venga stabilita già alla fine del secondo settore, in modo che in prossimità dei box il concorrente possa ottenere l'informazione istantaneamente e possa quindi decidere come e dove procedere.

4.5.2 Distribuzione del box

La componente box è una delle parti che si è scelto di distribuire.
La comunicazione box-concorrente avviene su un piano separato rispetto alla comunicazione box-monitor. Il box reperisce i dati di ogni settore dalla componente **Competition_Monitor**, la quale si limita a richiedere le informazioni necessarie ai gestori delle statistiche (che vedremo in seguito).
Una parte separata rispetto all'**Update_Retriever** (così si chiama il task che recupera gli aggiornamenti) è il **Box_Radio**. Questa è un'entità reattiva (un server) che rimane in attesa della richiesta di strategia (chiamata remota) da parte del concorrente, per poi fornire la strategia più aggiornata.
Per impedire che un fault della rete (o del nodo) vada a intaccare la robustezza del sistema e il regolare svolgimento della competizione si è implementato un sistema che riesca a continuare la sua esecuzione anche se un nodo dovesse cadere. La competizione continua il suo svolgimento e l'unico problema visibile è la mancata comunicazione fra il concorrente e i box che, in caso di danni permanenti al nodo (ad esempio non torna on-line prima della fine della gara) porterà al mancato pitstop e cambio di strategia del concorrente. Questa è la soluzione ad una parte del problema dovuto al sistema distribuito, gestendo questi fault e garantendo la corretta e continua esecuzione della simulazione. Inoltre la consapevolezza di errori di rete porta alla presenza di componenti di non determinismo comunque controllato e gestito, come richiesto nel paragrafo 3.4.
Ulteriori dettagli relativi alla distribuzione possono essere reperiti in appendice alla seguente sezione B.3.

4.6 Gestione delle statistiche di gara

Come analizzato nella sezione 3.7, per poter reperire un'istantanea di gara ad un dato istante di tempo, è necessario disporre di una storia degli eventi avvenuti nel corso della competizione, strutturati in modo che risulti possibile risalire a qualunque evento (rilevante) già avvenuto.
Nel corso delle due seguenti sottosezioni verranno analizzate le soluzioni adottate per immagazzinare i dati relativi ai singoli concorrenti e calcolare quelli relativi alla gara.

4.6.1 Dati singolo concorrente

Come visto nel capitolo relativo all'entità **Circuito**, ogni concorrente aggiorna, ad ogni **Checkpoint**, un orologio relativo alla sua corsa. Per ognuno si può quindi sapere, per ogni checkpoint passato in un determinato giro, l'istante di arrivo.
Per implementare, in parte, la soluzione proposta si è quindi pensato di salvare per ogni concorrente una lista di dati cronologicamente ordinata. Ogni posizione della lista mantiene informazioni legate ad un determinato istante di tempo. In questo modo è possibile attuare discussa nella sezione 3.7:
[...]una volta richiesto lo snapshot di un istante temporale, lo stato di ogni entità (utile all'istantanea) in quell'istante sia disponibile[...]
Ogni posizione della lista fornisce i seguenti dati:

- Time: l'istante di riferimento delle informazioni;
- Checkpoint: il tratto puntato da questo **Checkpoint** che è stato attraversato (che si è cioè finito di attraversare) all'istante dato;
- Sector: il settore di appartenenza del tratto;
- Lap: il giro di riferimento;
- Gas_Level: il livello di gas presente nel serbatoio alla fine dell'attraversamento;
- Tyre_Usury: il livello di usura gomme alla fine dell'attraversamento;
- BestLapNum: la miglior lap dall'inizio della competizione fino all'istante dato;
- BestLaptime: il tempo della miglior lap dall'inizio della competizione fino all'istante dato;
- BestSectorTimes: i migliori tempi per ogni settore dall'inizio della competizione;

- Max_Speed: la massima velocità raggiunta dall'inizio della competizione;
- PathLength: la lunghezza della traiettoria attraversata.

Lo stato del concorrente ad un istante t si può ottenere navigando la lista fino a trovare la posizione che contiene le informazioni dell'istante di tempo esatto (poco probabile). Altrimenti, se il tempo richiesto è compreso in un intervallo limitato dagli istanti indicati in due posizioni contigue, lo stato sarà anche compreso fra gli stati forniti dalle due posizioni. Più precisamente:

- per quanto riguarda la posizione sul circuito, si può indicare fra quali **Checkpoint** è il concorrente;
- per quanto riguarda i migliori tempi e altri dati statistici, valgono quelli indicati dalla posizione con istante di tempo più basso (è chiaramente un'approssimazione);

Ogni posizione della lista è stata progettata come una risorsa passiva ad accesso controllato con guardia. Vale a dire che la risorsa può essere acceduta solo nel momento in cui essa sia valorizzata con dei dati di gara. Fino a tal momento l'entità richiedente dovrà rimanere in attesa. Questa soluzione aiuta a mettere in pratica il suggerimento discusso durante l'analisi. Ovvero: se uno stato ad un istante t non è ancora disponibile, il richiedente dovrà rimanere in attesa. In questo caso, nel momento in cui una qualunque entità attiva avanzerà la richiesta di un'istantanea ad un dato istante, finché tutti i concorrenti non avranno fornito il proprio stato relativo a quell'istante, il richiedente verrà fatto attendere.
Nello specifico, la componente che si occupa dell'aggiornamento dei dati della lista è **Competitor_Computer**. Ogni concorrente mantiene un riferimento ad un istanza di **Computer** (è una parte della componente **Competitor_Computer**) che utilizza per aggiornare i dati ad ogni checkpoint.

4.6.2 Dati globali di gara

I dati globali di gara, concettualmente, altro non sono che una rielaborazione dei dati dei singoli concorrenti. Ad esempio, la classifica ad un dato istante viene calcolata a partire dalle posizioni dei singoli concorrenti in tale istante. Ancora, il miglior giro è dato dal tempo più basso fra i tempi di miglior giro di tutti i concorrenti. E così via. La componente che si occupa di questo è **Competition_Computer**. Venendo costantemente aggiornata (indirettamente) tramite i singoli concorrenti, è in grado di fornire, quando disponibile,

dati di gara calcolati sulle informazioni dei singoli concorrenti. A supporto è stata prevista inoltre una struttura denominata **Placement_Handler** che serve per tenere aggiornata costantemente la classifica e poterla fornire in relazione ad un giro o ad un istante temporale. Più in dettaglio, il **Placement_Handler** consiste di una lista di tabelle che rappresentano la classifica. Una per ogni lap. Ognuna mantiene una lista ordinata per tempo crescente di concorrenti che hanno finito il dato giro (anche l'istante di fine giro è salvato). Ognuna di queste, quindi, è stata implementata come una risorsa passiva ad accesso mutuamente esclusivo, per evitare race condition fra due o più concorrenti (thread) che stiano aggiornando la tabella di una lap in modo concorrente.

4.7 Reperimento istantanea di gara

In seguito a quanto detto nella sezione 4.6, è abbastanza facile intuire come venga effettuata un'istantanea della competizione. Si è visto che ogni concorrente genera una storia della sua gara fino all'ultimo **Checkpoint** raggiunto. Per ogni istante è (o sarà) quindi disponibile posizione e statistiche di ogni singolo concorrente.
La componente dedicata ai dati globali di gara (**Competition_Monitor**) si occupa invece di rielaborare i dati dei singoli concorrenti per estrarre informazioni statistiche sulla competizione, quali:

- classifica
- miglior giro con relativo tempo e concorrente
- migliori tempi per ogni settore con relativo concorrente per ognuno di essi

Di conseguenza, dato un istante di tempo t, è possibile reperire tutte le informazioni di gara relative a tale istante. Se qualche concorrente non ha ancora raggiunto l'istante dato, la richiesta dello snapshot (come si è visto) verrà messa in attesa.

4.8 Interazione utente-sistema

Nella simulazione si è scelto di permettere una interazione fra l'utente esterno e il sistema. Le modalità verranno esposte approfonditamente nei prossimi paragrafi. L'interazione che verrà analizzata è divisa in due tipologie:

- la prima riguarda l'osservazione della gara e la velocità di simulazione;
- la seconda riguarda interventi sulla gara da parte di un utente esterno, che completa le funzionalità di non determinismo permesse nei primi capitoli.

L'interazione avviene da postazioni remote. Ovvero l'utente avrà TV e Box dislocati in nodi separati rispetto alla competizione. Maggiori dettagli a riguardo possono essere trovati alla sezione in appendice .

4.8.1 Osservazione gara (lato box, lato tv)

Per quanto riguarda l'osservazione della gara questa avviene in due modalità differenti. La prima è la visione della gara nel suo insieme, con la visualizzazione istante dopo istante della posizione di tutti i concorrenti in pista, la classifica giro dopo giro e le migliori prestazioni. La seconda invece è la visualizzazione da parte dei box con i dati del solo concorrente a cui fa riferimento. Vengono visualizzate informazioni relative ai consumi e ai tempi di percorrenza. La sola osservazione della gara non introduce errori in quanto non l'unica azione prevista è il reperimento di informazioni già presenti nel sistema centrale. Nel caso non siano ancora presenti i dati richiesti, la chiamata (essendo sincrona) diventa bloccante in attesa delle informazioni richieste. La possibilità di richiesta di informazioni che non saranno mai disponibili nel sistema (ad esempio istanti di tempo a cui la gara non arriverà mai) non sono possibili in quanto le informazioni vengono richieste in maniera sequenziale e viene comunicato quando non sono più disponibili dati di interesse.

4.8.2 Intervento sulla gara (lato box, lato tv)

Gli interventi permessi sulla gara sono di due tipi. Lato tv del monitor della competizione viene permessa la modifica del tempo di simulazione, velocizzandolo o rallentandolo. Lato tv per la sola visualizzazione dell'andamento della gara non è previsto nessun intervento. Lato box invece è possibile richiamare la macchina ai box per un pitstop forzato, introducendo così l'ultima componente di non determinismo permessa. Ogni intervento che è stato permesso sulla gara non introduce altri errori in quanto in un caso va a modificare il delay inserito per rendere reale la simulazione (non vincolante quindi ai fini della correttezza temporale) e nell'altro forza una azione comunque prevista (e presente) nel sistema.

4.9 Inizializzazione competizione

Il seguente diagramma di sequenza spiega come avviene la sequenza di init della competizione, a partire dal main della competizione **main_competition.adb**. È stato creato uno script (in bash) che effettua l'avvio di tale main e che successivamente inizializza l'interfaccia di configurazione (in java) della competizione.

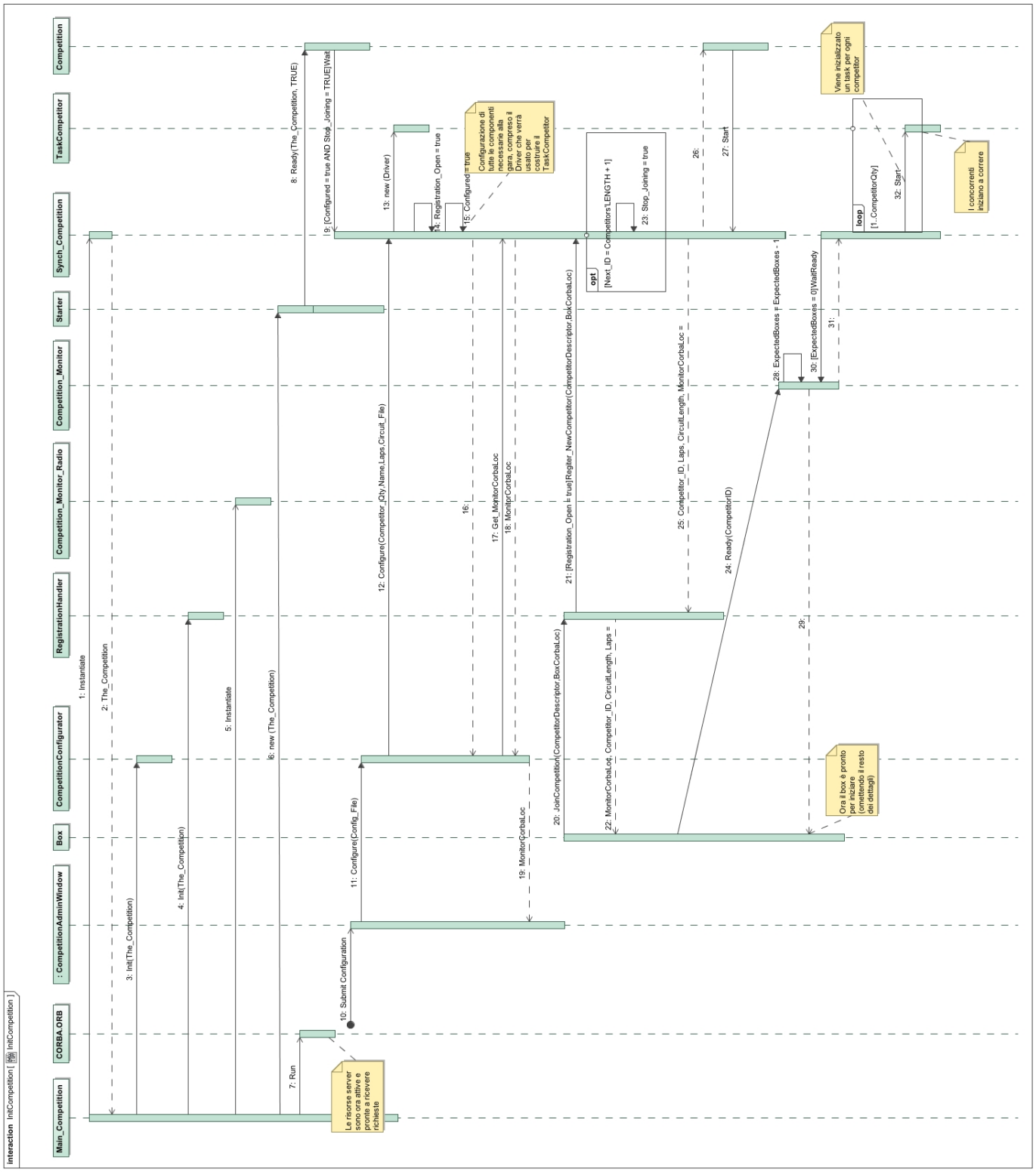

Figura 1: Sequence Diagram - Inizializzazione competizione

Una volta avviato il main della competizione, per prima cosa vengono inizializzati tutti i server, vale a dire **CompetitionConfigurator**, **Registration_Handler** e **Competition_Monitor_Radio**. Viene avviato un task **Starter** che rimane in attesa della configurazione e registrazione concorrenti avvenuti per dare lo Start alla gara. Viene inoltre allocata una risorsa protetta di tipo **Synch_Competition** pensata per agevolare la fase di init. Ne viene condivisa un'istanza fra **CompetitionConfigurator**, **Registration_Handler** e **Starter**. Tale risorsa permette di regolare i vari step dell'inizializzazione. Finchè la competizione non viene configurata (dal passo **10** al passo **16**), non sarà concesso di registrare concorrenti. Una volta configurata la competizione, viene aperta la guardia REGISTRATION_OPEN e il metodo Register_NewCompetitor potrà essere invocato (in **Synch_Competition**). L'iscrizione dei concorrenti avviene a partire dai *Box*. Nel diagramma l'azione iniziale del *Box* è la **20**. In realtà, nel nodo del box, viene prima avviato uno script che esegue il main del box seguito dal main dell'interfaccia (java) di configurazione del concorrente e del box. A configurazione avvenuta, i parametri vengono inviati al server della competizione invocando il metodo del server **Registration_Handler** Join_Competition, come scritto nel diagramma. A iscrizione avvenuta, i parametri di configurazione della competizione tornati vengono usati per inizializzare e avviare il box e relativi task grazie a cui successivamente sarà possibile mandare il messaggio di Ready alla competizione. Quando tutti i concorrenti sono stati registrati (**23**), bisogna rimanere in attesa della conferma di avvenuta inizializzazione dei *Box*. Ciò avviene quando il thread del task **Starter** invoca il metodo Start del **Synch_Competition**, dentro cui viene messo in attesa sul **Competition_Monitor** tramite Wait_Ready. Quando tutti i *Box* hanno dato l'ok, significa che sono pronti per ricevere richieste dai rispettivi competitor e la gara può cominciare. Vengono così avviati uno ad uno (**32**) e la competizione ha inizio.

4.10 Stop competizione

Lo stop del sistema avviene su 3 livelli: stop dei task, stop delle interfacce utente e stop dei server. Vediamo in dettaglio:

Stop dei task:

i task che devono essere fermati sono i **Competitor_Task** lato competizione e i task **Update_Retriever** e **Strategy_Updater** lato box. I primi si fermano in automatico quando le condizioni dell'auto non permettono di procedere (benzina finita o gomme troppo usurate) oppure a competizione finita (fine ultima lap). I secondi due si fermano quando l'aggiornamento fornito dal monitor della competizione segnala uno stato

dell'auto a causa di cui il concorrente non può procedere (poca benzina o molta usura gomme), oppure quando il concorrente finisce l'ultima lap (sempre grazie agli aggiornamenti);

Stop delle interfacce utente:
è sufficiente chiudere le interfacce con il tasto **x** in alto a destra;

Stop dei server:
una volta finita la gara e chiuse le interfacce utente, lo script di avvio procede l'esecuzione invocando un **killall -9** sui main avviati, interrompendo così anche i task server.

5 Analisi del supporto tecnologico

5.1 Scelta dei linguaggi

Per il progetto sono stati utilizzati principalmente due linguaggi (+2 in piccola parte):

- **ADA**: il linguaggio è stato utilizzato per sviluppare i layer sottostanti allo strato di presentazione. Si è valutato infatti, dopo le tematiche affrontate durante il corso, che sarebbe stato più agevole sviluppare un sistema multi-threaded e caratterizzato da forti vincoli di affidabilità e determinismo utilizzando tale linguaggio. Il forte sistema di tipi e le primitive a supporto della gestione dei thread e delle risorse condivise si sono infatti rivelati decisivi perr avere un maggior controllo sul sistema in fase di sviluppo già a compile-time. Più precisamente, il linguaggio stesso ci avrebbe (e ci ha) vincolato a scrivere del codice logicamente corretto senza troppe iterazioni di debug.

- **Java**: per lo sviluppo delle interfacce grafiche è stato invece utilizzato Java. La scelta è stata dettata da vari motivi. Primo fra tutti la competenza. È sembrato più intuitivo poter gestire un argomento come l'interfaccia utente usando tale linguaggio. Non risultando infatti critiche a livello di logica e occupando il 15% dell'intero progetto, si è pensato che per le interfacce sarebbe stato più che sufficiente. Anche per la competenza già acquisita di librerie come *awt* e *swing*.
  Seconda motivazione, la portabilità. La parte logica del sistema è stata concepita per girare su una macchina con il supporto di un numero di nodi massimo pari alla quantità di concorrenti. La TV per la visualizzazione della gara potrebbe invece essere potenzialmente avviata su 1000 macchine diverse. È parsa quindi una scelta abbastanza furba offrire la possibilità di

utilizzare tale interfaccia in qualunque computer appoggiandosi alla JVM invece che doverla ricompilare e magari riadattare per ogni macchina.

- **Bash**: questo linguaggio di scripting è stato usato per la configurazione, l'avvio e lo stop del sistema.
- **IDL**: il linguaggio è stato usato per generare il codice finalizzato all'interoperabilità fra linuguaggi.

5.2 Distribuzione e interoperabilità fra linguaggi

Per quanto riguarda la parte distribuita si è scelto di utilizzare *Polyorb* con profilo *CORBA*. La scelta è caduta su questa tecnologia dopo aver valutato i pregi e i difetti delle altre tecnologie come ad esempio *Distributed System Annex* e *MOM*. La prima tecnologia è stata scartata in quanto si voleva realizzare un prodotto con comunicazione fra linguaggi diversi (in questo caso Ada e Java) e per la parte Java si è trovato solo un articolo scientifico che parlasse di binding di Distributed Annex per Java. La seconda tecnologia menzionata è stata scartata in quanto non sufficientemente conosciuta (a differenza delle altre viste a lezione) e presente solamente nella versione J2EE di Java e non, come per CORBA, nella versione standard di Java e anche nella OpenJDK (presente in molti sistemi Unix). La scelta è quindi caduta su CORBA, supportato nella API di Java e supportato anche, tramite Polyorb, in Ada.

A Architettura ad alto livello

Nella seguente sezione verrà illustrata l'architettura ad alto livello del sistema sviluppato, escludendo i dettagli implementativi e legati al linguaggio. Viene fornito un diagramma delle componenti per fornire un'idea generale.

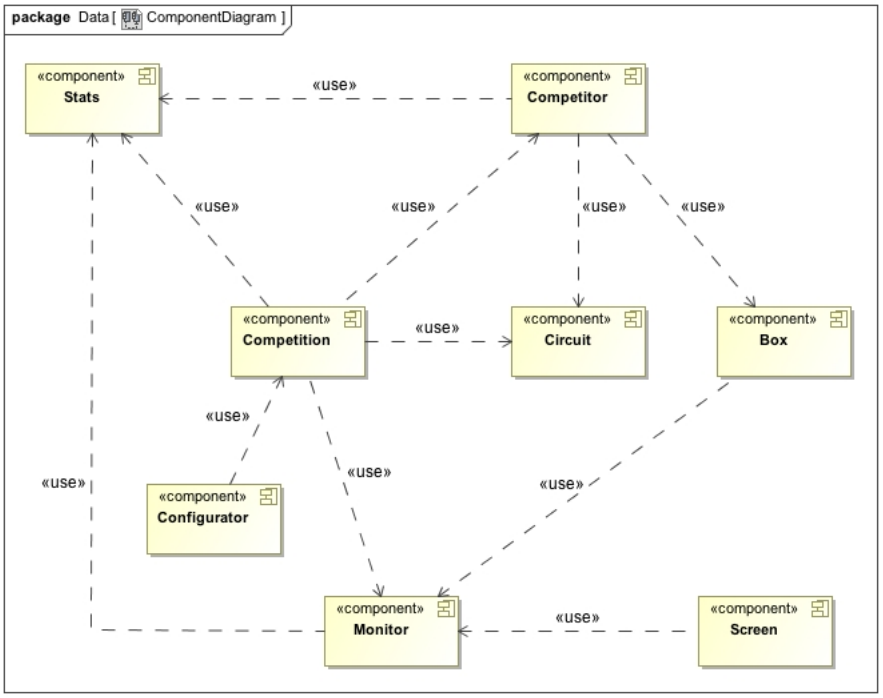

Figura 2: Diagramma delle componenti

A.1 Componenti di sistema

Le principali componenti del sistema sono:

A.1.1 Competition

La *Competition* è l'unità atta ad orchestrare l'avvio e la conclusione della corsa. Tale componente, dunque, è stata concepita per offrire le seguenti funzionalità:

- Configurazione parametri di gara:
  - numero di giri;

  - numero di concorrenti;
  - circuito;

- Gestione della sessione di iscrizione e accettazione concorrenti (configurati a livello della componente *Box*)
- Avvio delle componenti necessarie al monitoraggio della gara (quali ad esempio *Monitor*)
- Avvio controllato della competizione vera e propria nel momento in cui tutti i prerequisiti di inizio sono soddisfatti, ovvero:
  - la competizione è stata configurata correttamente;
  - le componenti di controllo e gestione della competizione sono attive e in attesa di comandi;
  - i concorrenti sono stati correttamente registrati alla competizione e in attesa di partire;

A.1.2 Competitor

Il *Competitor* è l'entità pensata a svolgere la gara. È caratterizzato dalle seguenti sotto-componenti:

- **Auto**, ovvero tutte le caratteristiche fisiche legate all'auto:
  - motore;
  - capacità del serbatoio;
  - massima accelerazione;
  - massima velocità;
  - gomme montate;
  - livello usura gomme;
  - livello della benzina nel serbatoio;
- **Pilota**, cioè le informazioni che descrivono più dettagliatamente il concorrente in gara:
  - nome e cognome pilota;
  - nome scuderia
- **Strategia**, ovvero la strategia che sta adottando il pilota, suggerita dai box e dinamica nel corso della gara:

  - stile di guida, variabile tra conservativo, normale, aggressivo, a seconda dello stato della macchina e delle previsioni fatte dai box
  - numero di lap prima del pit stop
  - addizionalmente, quando viene fatto un pit stop, la strategia determina anche la quantità di benzina da avere nel serbatoio dopo lo stop e il tempo impiegato per il pit stop (le gomme si assume vengano cambiate ad ogni pitstop).

Tutte queste informazioni insieme creano quello che viene definito il concorrente di gara. Tali informazioni verranno poi usate nel corso della gara per:

- scegliere al momento giusto la miglior traiettoria da seguire, in base alle sue caratteristiche (lunghezza, angolo ..) e in base alla presenza o meno di altri concorrenti;
- fornire costantemente aggiornamenti sul suo stato (tramite una parte del modulo *Stats*, informando il computer di bordo riguardo a:
  - livello di usura gomme;
  - livello di benzina;
  - checkpoint attraversato con tempo di arrivo;
  - insieme al checkpoint verranno aggiornate le informazioni relative a settore e lap;
  - velocità massima raggiunta.
- contattare ad ogni giro il box per ottenere una strategia aggiornata;
- se suggerito dai box, effettuare un pitstop;
- ritirarsi dalla gara una volta che le condizioni dell'auto non permettano di poter correre ulteriormente;
- banalmente, continuare a correre fino alla fine delle lap prestabilite, dopodichè fermarsi.

A.1.3 Circuit

Il circuito è una risorsa finalizzata ad offrire il piano su cui svolgere la competizione. È condivisa fra tutti i concorrenti in gara e offre un insieme di funzionalità per poter conoscere le caratteristiche dei vari tratti della pista (compresi i concorrenti presenti al momento dell'attraversamento). È composto dalle seguenti sotto-componenti:

- **Checkpoint**: i *Checkpoint* rappresentano punti di arrivo intermedi del circuito. Come degli intervalli da 1 secondo possono discretizzare 1 minuto, così i *Checkpoint* discretizzano il circuito. Ogni *Checkpoint* introduce un tratto della pista potenzialmente diverso da quello precedente. Per esempio, il $Checkpoint_n$ potrebbe essere il punto di entrata di un tratto della pista accessibile ad un numero massimo di 4 concorrenti insieme, mentre il successivo $Checkpoint_{n+1}$ potrebbe esporre un tratto più stretto e quindi accessibile solo a 2 concorrenti. Schematizzando, il *Checkpoint* è caratterizzato da:
  - **molteplicità**: ovvero il numero di concorrenti che possono trovarsi contemporaneamente nel tratto a seguire;
  - **posizione nella pista**: un *Checkpoint* può essere il traguardo, l'inizio del settore, la fine di un settore, all'uscita dei box, l'entrata per i box, i box oppure un punto intermedio fra altri due *Checkpoint*;
  - **tempi di arrivo**: ogni *Checkpoint* tiene traccia dell'istante in cui un concorrente ci è passato sopra.
- **Path**: rappresenta una delle possibili traiettorie da usare per andare da un *Checkpoint* a quello successivo. La traiettoria presenta un numero di *Path* uguale alla molteplicità del *Checkpoint* che la precede. Ogni *Path* è descritto da:
  - lunghezza
  - angolo
  - grip, ovvero l'aderenza sul tratto

  Normalmente i path appartenenti allo stesso tratto differiscono di poco.
- **Iteratore**: l'unità permette di sapere la struttura della pista. Si suppone venga usata per sapere quale *Checkpoint* ne segue un altro, oppure per sapere dove sia il *Checkpoint* di inizio box.

A.1.4 Stats

Questa componente mantiene la storia della gara e offre un insieme di funzionalità che permettono di elaborare tali dati per offrirne differenti viste:

- migliori performance in un determinato istante di tempo
- classifica aggiornata per istante di tempo
- informazioni sui concorrenti relative ad un particolare lap, checkpoint o settore (in una specifica lap);

A.1.5 Box

Il *Box* è l'entità che si occupa di gestire la configurazione e la corsa di un concorrente. Durante la competizione, il *Box* verifica costantemente lo stato dell'auto e fornisce eventuali cambi di strategia se ritenuto opportuno. Inoltre decide quando giugne il momento del pitstop e aggiorna di conesguenza le impostazioni della macchina, ovvero benzina nel serbatoio e gomme nuove.
Ogni *Box* è caratterizzato da uno fra 4 tipi di strategia, diversi per grado di "ottimismo" nelle valutazioni e nei calcoli dati lo stato della macchina, le medie calcolate e lo stile di guida del concorrente:

1. **Cautious**: cauto, sottostima il numero di giri ancora fattibili;
2. **Normal**: stima abbastanza realistica delle possibilità del concorrente, considera anche un margine di errore nei calcoli per effettuare una valutazione;
3. **Risky**: le stime vengono effettuate in base a calcoli esatti che di solito non tengono in considerazione fattori che nella realtà possono incidere in modo negativo;
4. **Fool**: nella realtà normalmente non si arriva a tanto, ma per fini di test è stato inserito anche un tipo di strategia che sovrastima le possibilità del concorrente, portandolo a squalifica quasi certa.

Ciò che il box suggerisce al concorrente durante la gara è:

- stile di guida. Verrà suggerito uno stile più conservativo se i consumi si sono rivelati maggiori del previsto e viceversa;
- numero di lap al pitstop

Il *Box* riceve informazioni sullo stato del concorrente alla fine di ogni settore e ricalcola la strategia alla fine del secondo settore. Il concorrente richiede la nuova strategia al box in prossimità del checkpoint dove è possibile proseguire o andare ai box.
È sembrata più realistica la scelta di non calcolare la strategia alla fine del terzo settore, perchè si suppone che nella realtà non si possa essere così veloci da calcolare una nuova strategia istantaneamente alla fine del circuito con i dati del terzo settore. È piuttosto più probabile che qualunque cambio di strategia o richiesta di rientro ai box venga stabilita già alla fine del secondo settore, in modo che in prossimità dei box il concorrente possa ottenere l'informazione istantaneamente e possa quindi decidere come e dove procedere.

A.1.6 Monitor

La componente *Monitor* è costituita dall'insieme delle unità concepite per esporre le informazioni e i dati prodotti a basso livello dal simulatore. Ne esistono due tipi:

- **monitor di competizione**: utilizzato per offrire informazioni riguardanti la gara e i singoli concorrenti;
- **monitor di box**: utilizzato per esporre i dati relativi alle computazioni del box e le informazioni grezze legate al concorrente appartenente alla sua scuderia e i dati rielaborati di settore in settore per formulare nuove strategie.

A.1.7 Screen

Gli *Screen* sono la parte più alta del sistema. Servono a mettere in connessione l'utente finale e la parte logica del simulatore. Tramite gli *Screen* è possibile ricevere aggiornamenti grafici (di base) sullo stato di avanzamento della simulazione sotto il punto di vista dei singoli box o della gara complessiva.

A.1.8 Configurator

Questa componente offre la possibilità di configurare le parti parametriche del sistema, ovvero:

- concorrenti
- competizione (i parametri elencati nella sezione *Competition*)
- box

Il *Configurator* ha inoltre l'onere di inviare i parametri così configurati alle componenti opportune.

A.2 Interazione fra le componenti

Nella seguente sezione verranno spiegate le interazioni principali fra le componenti.

A.2.1 Configurator-Competition

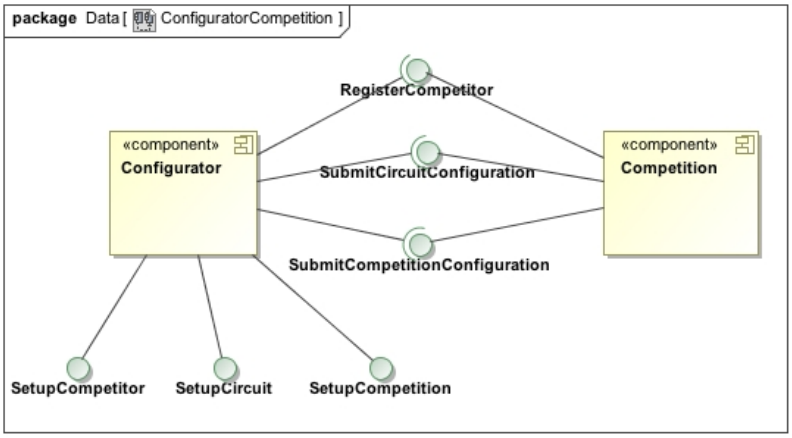

Figura 3: Protocol / Interface diagram - Configurator/Competition

Al livello più alto della fase di configurazione c'è la componente *Configurator*, la quale viene utilizzata per impostare i parametri relativi alla *Competition*. Tali parametri vengono prima impostati dall'utente tramite le interfacce **SetupCompetitor**, **SetupCompetition** e **SetupCircuit** e successivamente inviati alla componente *Competition* per l'inizializzazione.
RegisterCompetitor è utilizzata una volta per ogni concorrente da iscrivere.

A.2.2 Competition-Competitor

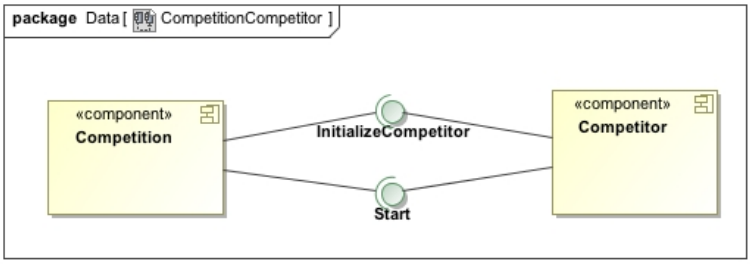

Figura 4: Protocol / Interface diagram - Competition/Competitor

L'interazione fra la *Competition* e il *Competitor* avviene, come per le altre interazioni *Competition*-componente, in fase di configurazione e avvio. La *Competition* si occupa di ricevere i parametri relativi a ogni *Competitor* dal *Configurator* (come verrà spiegato fra poco) per poi inizializzare il competitor. Quando tutti i concorrenti sono pronti e anche i *Box*, l'interfaccia **Start** verrà utilizzata per dare il via ai concorrenti.

A.2.3 Competition-Monitor

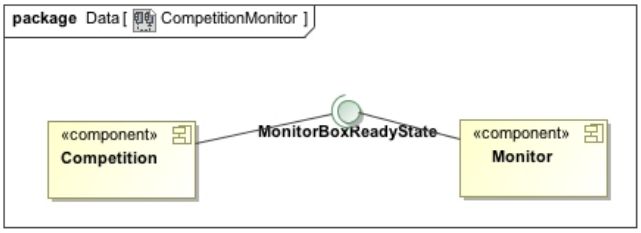

Figura 5: Protocol / Interface diagram - Competition/Monitor

L'interazione *Competition-Monitor* riguarda solo una piccola fase dell'inizializzazione, processo che verrà esplicato in dettaglio in seguito. A concorrenti registrati, la *Competition* sfrutterà un interfaccia offerta dal monitor per sapere quando tutti i *Box* sono pronti e avviati. L'interfaccia è l'unica illustrata in figura.

A.2.4 Competition-Circuit

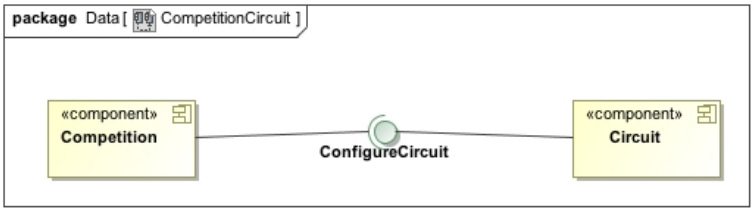

Figura 6: Protocol / Interface diagram - Competition/Circuit

Anche in questo caso, l'interazione *Competition-Circuit* si ha in fase di inizializzazione quando le due componenti entrano in contatto per la configurazione. La *Competition* inizializza cioè il circuito impostando i parametri che lo caratterizzano, quali:

- numero di checkpoint
- caratteristiche dei tratti fra checkpoint (lunghezza, angolo ...)
- posizione dei checkpoint di entrata e uscita box
- posizione del checkpoint traguardo

A.2.5 Competition-Stats

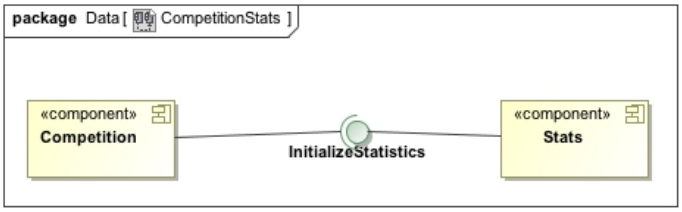

Figura 7: Protocol / Interface diagram - Competition/Stats

La componente *Stats* ottiene i parametri di configurazione in fase di inizializzazione dalla *Competition*, tramite **InitilizeStatistics**.

A.2.6 Competitor-Stats

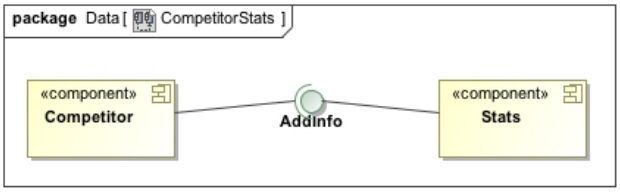

Figura 8: Protocol / Interface diagram - Competitor/Stats

La componente *Stats* offre al *Competitor* l'interfaccia **AddInfo** che il concorrente utilizza per fornire costantemente a *Stats* dati aggiornati riguardo alla gara in corso (dati relativi al singolo concorrente). Ad ogni checkpoint quindi il concorrente manda un aggiornamento a *Stats* che poi verranno utilizzate per effettuare calcoli di insieme riguardo alla gara o per essere mandate a chi le richiedesse.

A.2.7 Competitor-Circuit

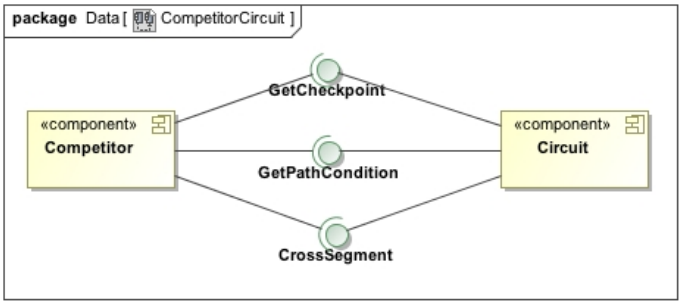

Figura 9: Protocol / Interface diagram - Competitor/Circuit

L'interazione *Competitor-Circuit* avviene durante lo svolgimento della competizione. Il concorrente sfrutta le interfacce del *Circuit* per ottenere informazioni sul circuito e "informarlo" degli spostamenti nel corso della gara.
GetCheckpoint garantisce che il concorrente ottenga sempre il checkpoint corretto a seconda della posizione corrente. **GetPathCondition** permette di ottenere informazioni sulle caratteristiche statiche e dinamiche della tratto da

attraversare. Le caratteristiche dinamiche sono legate ai concorrenti attualmente presenti sul tratto. **CrossSegment** assicura che il tratto possa essere attraversato senza collisioni e che il *Circuit* possa tracciare l'avvenuto passaggio dell'auto.

A.2.8 Monitor-Stats

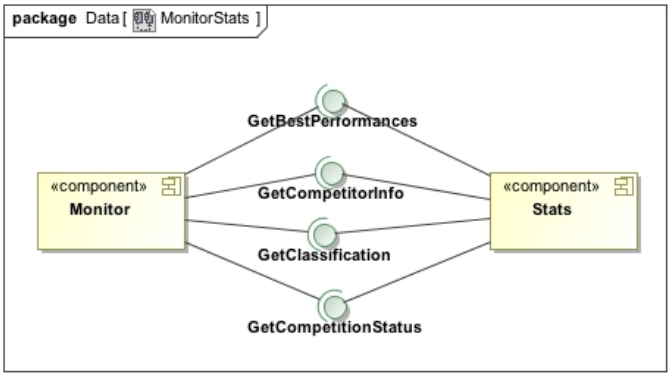

Figura 10: Protocol / Interface diagram - Monitor/Stats

La componente *Monitor* si appoggia a *Stats* per poter reperire le informazioni da essere esposte. Per questo motivo *Stats* offre un insieme di interfacce finalizzate a fornire i dati grezzi di competizione sotto forma di viste utili alla "pubblicazione".
GetBestPerformance fornisce i migliori tempi relativi a settori e giro.
GetCompetitorInfo reperisce informazioni legate al singolo concorrente, come ad esempio lo stato della macchina ad un determinato istante.
GetPlacement, come dice il nome, ritorna informazioni legate alla classifica.
GetCompetitionStatus espone informazioni legate alla competizione nel suo insieme, come ad esempio la posizione dei concorrenti nel circuito in un determinato istante di tempo.

A.2.9 Configurator-Box

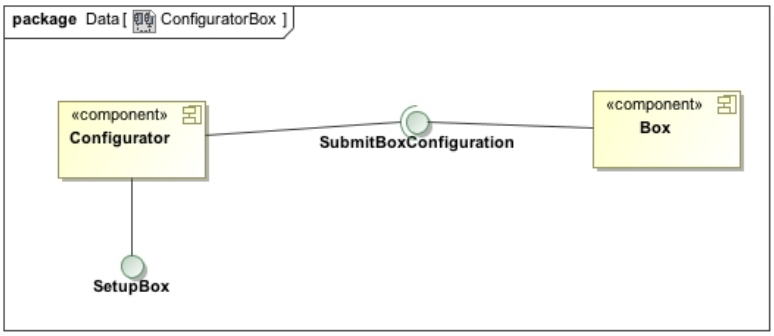

Figura 11: Protocol / Interface diagram - Configurator/Box

La componente *Configurator* offre anche la possibilità di configurare un *Box*. Per questo motivo il *Box* espone un'interfaccia da utilizzare per sottomettere i parametri di configurazione.

A.2.10 Box-Monitor

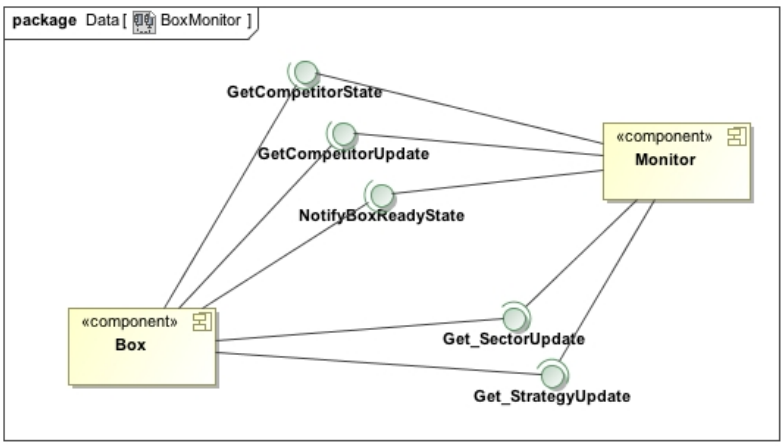

Figura 12: Protocol / Interface diagram - Box/Monitor

La prima interazione che la componente *Box* ha con il *Monitor* è in fase di inizializzazione della gara. La sequenza di azioni verrà esplicata dettagliatamente in seguito, per ora basti sapere che il *Box*, una volta configurato e pronto per

monitorare la gara ed elaborare i dati del suo concorrente, dovrà mandare una notifica tramite la componente *Monitor* utilizzando **NotifyBoxReadyState**.
Le rimanenti due interfacce offerte dal *Monitor* al *Box* sono utilizzate per reperire informazioni sullo stato del concorrente (livello di benzina rimasta, usura gomme...) e aggiornamenti riguardo al posizionamento e tempi del concorrente durante la gara.
Vi sono poi altre due interfacce offerte dal *Box* al *Monitor*. Per quanto possa sembrare un po' paradossale, l'architettura acquisisce senso se si pensa che la componente *Monitor*, ad alto livello, è stata pensata per pubblicare informazioni. Tali informazioni possono riguardare il singolo concorrente e quindi essere utili al *Box*. Altre possono invece riguardare il *Box* e i suoi calcoli per essere esposte ad un utente (per esempio). Le due interfacce offerte dal *Box* infatti servono per ottenere aggiornamenti sulle operazioni interne del *Box* qualora esse dovessero essere esposte ad un qualunque client. Come vedremo più in dettaglio in seguito, questa componente è in realtà costituita da due sotto-componenti, una dedicata a *Competition* e l'altra a *Box*.

A.2.11 Screen-Monitor

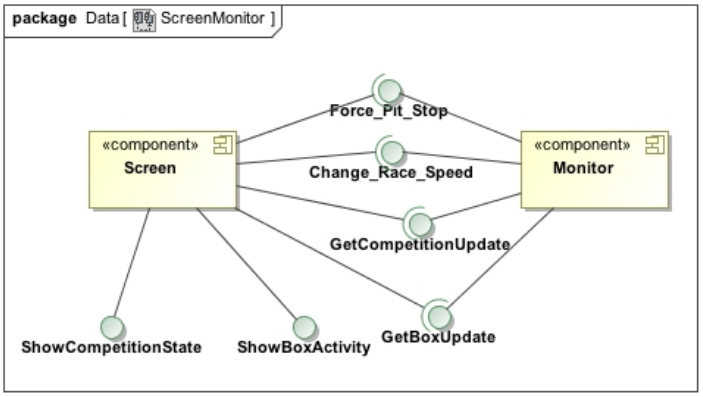

Figura 13: Protocol / Interface diagram - Screen/Monitor

La componente *Screen* comunica con il *Monitor* per ottenere informazioni utili da esporre graficamente all'utente. Tali informazioni possono riguardare la competizione in senso globale, oppure essere legate ai singoli concorrenti.
Inoltre dagli *Screen* è possibile intervenire sulla gara, accelerando o decelerando la velocità di simulazione e forzando un concorrente a rientrare ai box.

A.2.12 Competitor-Box

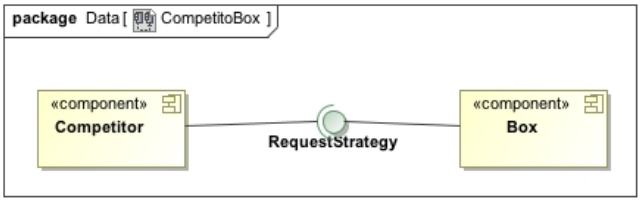

Figura 14: Protocol / Interface diagram - Competitor/Box

Man mano che la competizione procede, il *Box* colleziona i dati di gara del rispettivo concorrente per calcolarne medie e statistiche. A partire da queste informazioni produce una nuova strategia ogni giro. Per questo offre un'interfaccia **RequestStrategy** che il concorrente utilizza nel corso della gara per ottenere suggerimenti utili per proseguire. La strategia fornita dal *Box* potrebbe anche richiedere un pitstop per un rifornimento benzina e cambio gomme.

B Architettura in dettaglio

Si spiegherà ora con maggior dettaglio l'architettura di sistema, esplicando come le principali classi implementate svolgano le funzionalità esposte dalle interfacce illustrate nel capitolo precedente.

B.1 Diagrammi delle classi

B.1.1 Competition

La *Competition* è, come già accennato, una componente di init.
Il tutto ha inizio a partire da **Main_Competition** che, come dice il nome, è l'unità di avvio.
Main_Competition si occupa di istanziare gli oggetti necessari all'avvio e configurazione della competizione. Per la configurazione vengono istanziati **Registration_Handler** e **Competition_Configurator** (della componente *Configurator*). Per l'avvio invece **Starter**.
Ad orchestrare la scena, un unica istanza del **Synch_Competition** condivisa fra le altre 3 unità. Il loro scopo è:

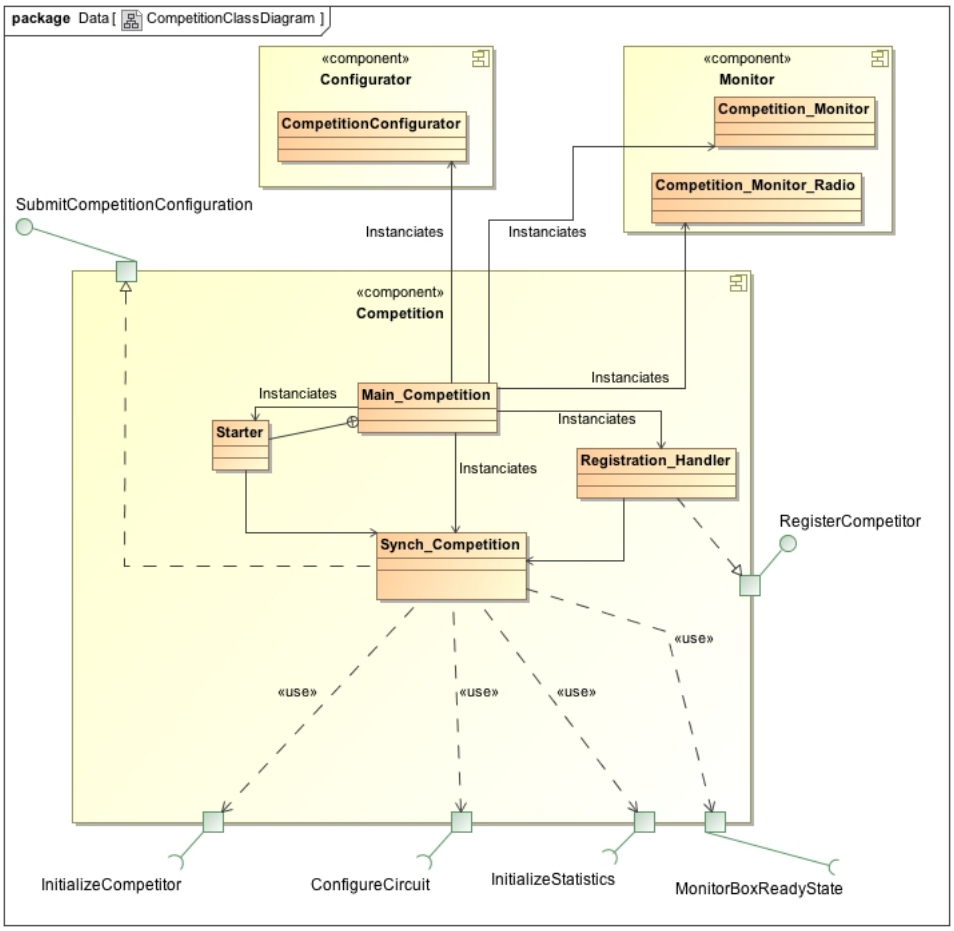

Figura 15: Class diagram - Competition

- **Synch_Competition**, è una risorsa protetta che gestisce l'accesso in mutua esclusione alla configurazione della competizione e dell'inizializzazione dei concorrenti. La risorsa assicura tramite *entry* a guardia booleana che avvengano in ordine prima la configurazione della competizione (da parte del **Competition_Configurator**) e poi la registrazione dei concorrenti (per opera del **Registration_Handler**). Questo vincolo è dato dal fatto che alcune impostazioni di competizione devono essere fornite ai box dopo la configurazione del concorrente, come ad esempio il numero di lap. Inoltre fra i parametri configurabili vi è anche il numero massimo di partecipanti alla gara. Di conseguenza è prima necessario conoscere il limite per poi poter regolare il flusso di registrazioni.
  Durante la fase di configurazione (metodo Configure), vengono inizializzati *Circuit* e *Stats* tramite le interfacce illustrate nel diagramma.

A configurazione di competizione avvenuta, verrà aperta le entry Register_New_Competitor, utilizzata dal **Registration_Handler** per registrare i vari concorrenti. Ad ogni invocazione verrà inizializzato un concorrente (**Competitor_Task**) con un iteratore al circuito e le impostazioni passate in input. Il task del concorrente così istanziato rimarrà in attesa dello "start".
Oltre alla funzionalità di configurazione, questa classe offre anche la funzionalità di avvio (metodo Start). Tale entry si aprirà solo quando tutti i concorrenti previsti sono stati iscritti. Viene invocata dall'unità **Starter**. Il metodo mette il task richiedente in attesa sulla risorsa **Start_Handler** (componente *Monitor*) sull'entry Wait_To_Be_First. Quando tutti i box avranno dato il loro ok (maggiori dettagli a seguire), il thread potrà continuare la sua esecuzione e passare allo Start di tutti i concorrenti in attesa di partire.

- **Starter** è il task finalizzato a gestire l'avvio della competizione. Una volta avviato dal main, il task utilizza il metodo Ready offerto dal package *Competition* per manovrare l'avvio. All'interno del metodo, prima si mette in attesa che tutti i concorrenti si siano iscritti (utilizzando il metodo Wait del singleton di **Synch_Competition**) per poi invocare il metodo Start dello stesso **Synch_Competition**, descritto poche righe più sopra.

- **Registration_Handler** è l'oggetto che rimane in attesa di concorrenti. Più precisamente, è un server dedicato ad accogliere le richieste di registrazione dei concorrenti. Con il supporto di Polyorb è possibile invocare questo oggetto da remoto. Ad ogni richiesta, vengono salvati i parametri di configurazione in un file xml il cui nome e locazione verranno passati a **Synch_Competition** per effettuare il resto delle procedure di inizializzazione del concorrente. Di ritorno ci saranno l'ID del concorrente, il numero di lap, la lunghezza del circuito e il corbaloc del **Competition_Monitor** che il *Box* dovrà utilizzare per rimanere sincronizzato sugli sviluppi del rispettivo concorrente.

B.1.2 Competitor

L'unità principale del *Competitor* è il **Competitor_Task**. È la risorsa attiva che si occupa di percorrere la pista e valutare la traiettoria ottima in base alle condizioni del tratto e dei competitor presenti nelle vicinanze. Le informazioni statiche del concorrente sono contenute in **Driver**. **Car** mantiene alcune carat-

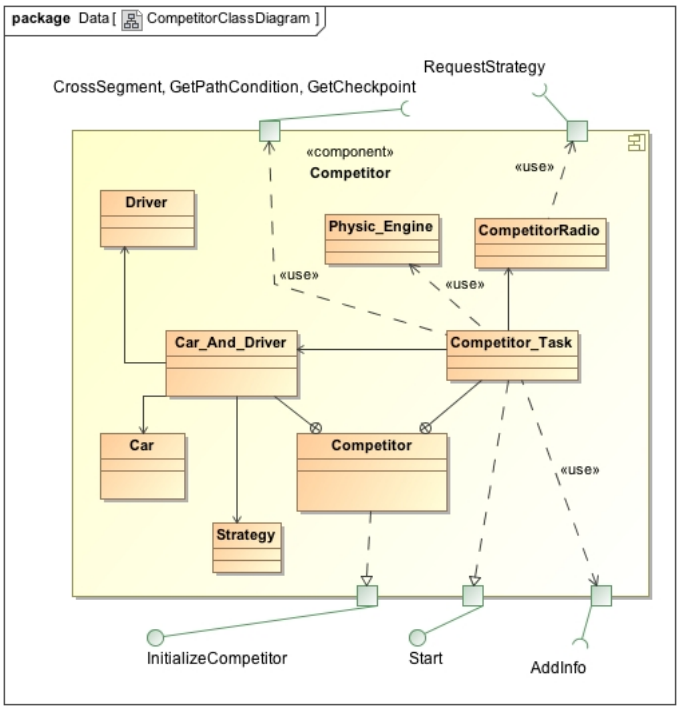

Figura 16: Class diagram - Competitor

teristiche statiche sulla macchina e altre dinamiche, come il livello di benzina e l'usura delle gomme. **Strategy** mantiene le seguenti informazioni:

- Stile di guida
- Numero di giri al pitstop (se 0, bisogna rientrare al pitstop)
- Durata del pitstop (nel caso il pitstop sia avvenuto)
- Livello di Gas (dopo il pitstop in caso sia avvenuto)

STILE DI GUIDA e NUMERO DI GIRI AL PITSTOP vengono aggiornate ad ogni giro, DURATA DEL PITSTOP e LIVELLO DI GAS vengono tenute in considerazione solo in caso di pitstop. Il livello di gas viene usato per aggiornare le impostazioni di **Car** (le gomme vengono automaticamente messe a nuovo). La durata del pitstop viene usata per aggiornare correttamente il tempo di arrivo segnato nel checkpoint di uscita dai box.
Quando un concorrente si trova sul checkpoint prima della corsia dei box, effettua una chiamata al *Box* tramite la **Competitor_Radio** per richiedere una strategia aggiornata. Se la strategia tornata contiene un numero di giri al pitstop pari a 0, il **Competitor_Task** valuterà e percorrerà la corsia dei box

invece che quella del circuito, per poter effettuare il pitstop.

Il **Competitor_Task** e tutte le unità associate vengono inizializzate in fase di init tramite i seguenti metodi offerti dal package **Competitor**:

- Configure_Driver
- Configure_Car
- Costruttore **Competitor_Task**

Il **Competitor** si appoggia al motore fisico gestito da **Physic_Engine** per calcolare i tempi e i consumi.

B.1.3 Circuit

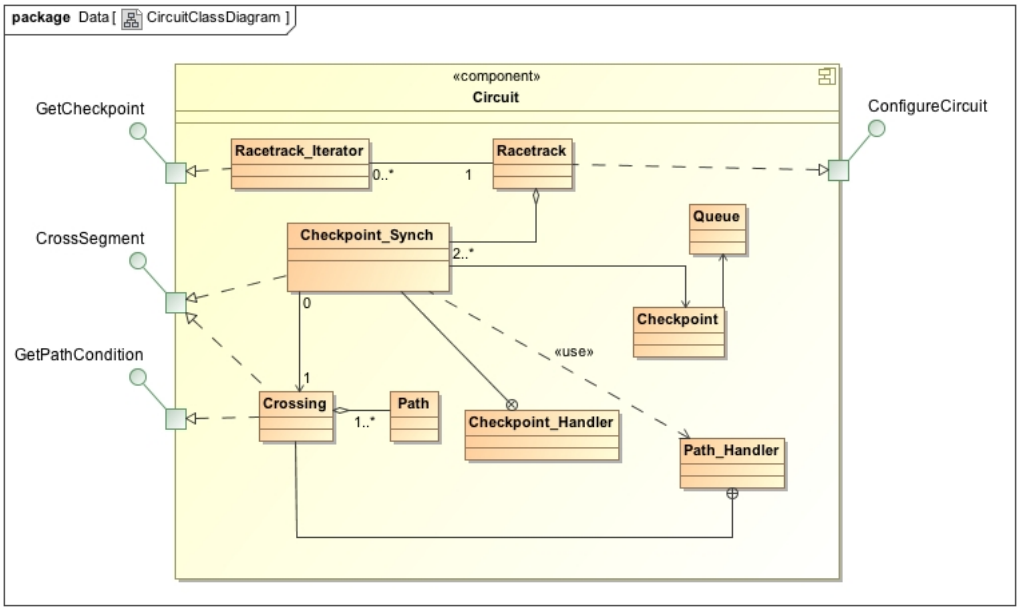

Figura 17: Class diagram - Circuit

Il circuito prende forma a partire da **Racetrack**. Questa unità si occupa di leggere e parsare il file xml dato in input e ricavarne i parametri per la creazione del circuito. Il file XML descrittore del circuito elenca i checkpoint presenti in ogni settore (che sono 3 costanti) e per ognuno specifica le caratteristiche del tratto di pista a seguire:

- lunghezza
- angolo
- numero di traiettorie percorribili
- grip, ovvero aderenza sul tratto

Vi sono inoltre 3 checkpoint che dovranno avere uno dei seguenti attributi booleani:

- goal, ovvero il checkpoint è il primo della pista
- prebox, ovvero dal checkpoint è possibile raggiungere i box
- exitbox, ovvero il checkpoint di arrivo una volta fuori dalla corsia dei box

Dati questi parametri, vengono inizializzati tutti i checkpoint stabiliti ed inseriti in un array chiamato **Racetrack**. Questo sarà l'array iterato dal **Race_Iterator**.

Ogni checkpoint è del tipo **Checkpoint**, il quale contiene tutte le informazioni elencate in precedenza oltre ad una coda per gestire i concorrenti che tentano di accedere al tratto. Il **Checkpoint** è poi inserito in una struttura protetta denominata **Checkpoint_Synch** finalizzata a regolare l'accesso in mutua esclusione. La risorsa inoltre incapsula la lista di **Path** che costituiscono il tratto associato al checkpoint. Infine offre una serie di metodi da utilizzarsi per interagire con le risorse sottostanti.
I **Path** appena accennati vengono generati in fase di creazione del **Checkpoint** a partire dalle informazioni di base reperite dal file di configurazione. Viene generator un numero di **Path** pari alle traiettorie percorribili. Ogni **Path** riceve valori di lunghezza che possono variare. Bisogna assicurare 1.5 m di larghezza per ogni macchina (approssimativamente). La prima riceve i valori di base e viene virtualmente posizionata su un bordo del tratto. Le altre vengono posizionate man mano una di fianco all'altra, quindi la lunghezza della traiettoria cresce in rapporto alla distanza dalla prima e all'angolo del tratto. Infine viene definito quante macchine possono restare in fila dentro ogni **Path** in base alla sua lunghezza e all'assunzione che una macchina necessiti di 4.5 m di spazio.
Tutti i **Path** di un tratto vengono incapsulati in una risorsa protetta denominata **Crossing** che ne regola l'accesso in mutua esclusione e offre i metodi di accesso pubblici.
Infine viene creata una corsia dei box coerente con la distanza fra i checkpoint "prebox" e l'"exitbox". I tratti prima e dopo il checkpoint dei box sono a molteplicità uguale al numero di concorrenti. Questo perchè potenzialmente ai box potrebbero esserci contemporaneamente tutte le auto. Inoltre si è deciso di posizionare il checkpoint del box in coincidenza con il traguardo. Quindi ogni box è anche un goal.

I checkpoint vengono gestiti dal package **Checkpoint_Handler**, mentre i path da **Path_Handler**. Verranno ora elencati metodi più rilevanti esposti dal **Checkpoint_Synch**:

procedure Signal_Arrival(CompetitorID_In : INTEGER)
Il metodo marca nella coda del checkpoint l'arrivo effettivo del concorrente;

procedure Signal_Leaving(CompetitorID_In : INTEGER)
Il metodo marca nella coda del checkpoint l'uscita del concorrente;

procedure Set_Lower_Bound_Arrival_Instant(CompetitorID_In : INTEGER; Time_In : FLOAT)
Il metodo segna il tempo previsto di arrivo del concorrente nella coda del checkpoint;

procedure Remove_Competitor(CompetitorID_In : INTEGER)
Il metodo rimuove il competitor dalla coda. Ciò significa che l'id e il tempo del competitor non apparirà più nella coda. Questo metodo è utilizzato, per esempio, quando un concorrente finisce la gara prima di altri e deve quindi liberare i checkpoint per evitare di creare starvation;

function Get_Time(CompetitorID_In : INTEGER) return FLOAT
La funzione ritorna il tempo segnato sulla coda del concorrente con ID dato in input;

entry Wait_To_Be_First(Competitor_ID : INTEGER) Nel momento in cui un concorrente arriva fisicamente su un checkpoint, dopo aver marcato il suo arrivo utilizza questo metodo per sapere quando arriva il suo turno per attraversare (viene cioè posizionato nella prima posizione della coda);

procedure Get_Paths(Paths2Cross : out CROSSING_POINT; Go2Box : BOOLEAN) Quando il concorrente sa di essere primo sulla coda del **Checkpoint** (in seguito all'invocazione del metodo Wait_To_Be_First), potrà invocare questa procedura per ottenere l'insieme di **Path** che costituiscono il tratto e procedere alla valutazione. Il booleano "Go2Box", se valorizzato a "true", impone al **Checkpoint_Synch** di tornare (se presente) l'insieme di **Path** relativi alla corsia dei box. Si è sicuri che nessuno starà effettuando operazioni sul tratto nel frattempo perchè tale azione da parte degli altri concorrenti non è ammissibile fino a che il **Competitor** corrente non abbia segnalato la sua partenza dal checkpoint tramite Signal_Leaving.

I metodi della risorsa **Crossing** invece servono a ritornare le caratteristiche di ogni **Path** (a partire dall'indice) e a aggiornare l'istante temporale segnato nel path.

Infine sono offerti un insieme di metodi da utilizzare insieme al **RaceTrack_Iterator** per navigare iterare il circuito. I metodi sono più rilevanti sono:

procedure Get_CurrentCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
CurrentCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint correntemente "puntato" dall'iteratore;

procedure Get_NextCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
NextCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint successivo. Nota Bene: il checkpoint dei box non è previsto essere ritornato da questo metodo. Per ottenere il checkpoint dei box è necessario utilizzare Get_BoxCheckpoint;

procedure Get_PreviousCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
PreviousCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint precedente, con le stesse regole del metodo precedente;

procedure Get_ExitBoxCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
ExitBoxCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint all'uscita della corsia dei box;

procedure Get_BoxCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
BoxCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint del box;

function Get_Position
(RaceIterator : RACETRACK_ITERATOR) return INTEGER
Per tornare la posizione corrente dell'iteratore.

Non è stato necessario inserire **Racetrack** o il suo iteratore in una risorsa protetta perchè ogni concorrente dispone della sua copia dell'**Racetrack_Iterator**.

B.1.4 Stats

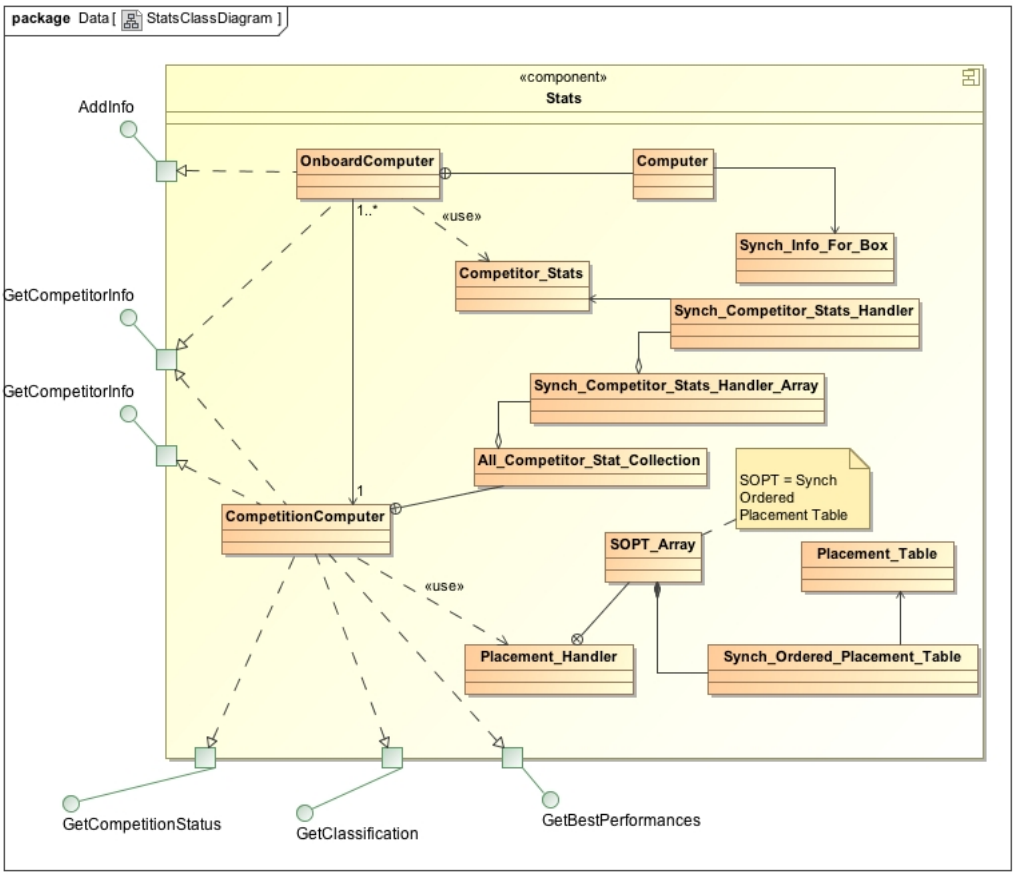

Figura 18: Class diagram - Stats

La componente *Stats* è implicitamente suddivisa in 2 sotto-componenti: **OnboardComputer** e **Competition_Computer**. Prima di discutere dei dettagli delle due, è necessario introdurre la struttura di una risorsa che viene utilizzata come pacchetto per il trasporto degli aggiornamenti tra *Competitor*, **OnboardComputer** e **Competition_Computer**. La risorsa è un tipo record denominato **Competitor_Stats**, contenente i seguenti dati:

- **Time : FLOAT;**
  l'istante a cui fa riferimento l'aggiornamento
- **Checkpoint : INTEGER;**
  il checkpoint che introduceva il tratto che è stato attraversato (completato) all'istante TIME

- **LastCheckInSect : BOOLEAN;**
  se true, il checkpoint è l'ultimo di un settore
- **FirstCheckInSect : BOOLEAN;**
  su true, il checkpoint è il primo del settore
- **Sector : INTEGER;**
  il settore a cui appartiene il checkpoint
- **Lap : INTEGER;**
  la lap in cui a cui l'aggiornamento fa riferimento
- **GasLevel : FLOAT;**
  il livello di gas presente nel serbatoio al momento in cui il tratto è stato attraversato (completato)
- **TyreUsury : PERCENTAGE;**
  la percentuale di usura gomme al momento in cui il tratto è stato attraversato (completato)
- **BestLapNum : INTEGER;**
  la miglior lap fatta dal concorrente dall'inizio della gara all'istante TIME
- **BestLaptime : FLOAT;**
  il tempo della miglior lap fatta dal concorrente dall'inizio della gara all'istante TIME
- **BestSectorTimes : FLOAT_ARRAY(1..3);**
  ogni indice dell'array indica un settore (quindi indice 1 indica il settore 1). Detto ciò, l'array contiene il miglior tempo fatto dal concorrente per ogni settore dall'inizio della gara all'istante TIME
- **MaxSpeed : FLOAT;**
  la massima velocità raggiunta dall'inizio della gara all'istante TIME
- **PathLength : FLOAT;**
  la lunghezza della traiettoria scelta per attraversare il tratto

Ora vediamo più dettagliatamente le due sotto-componenti accennate precedentemente:

- **OnboardComputer**:
  è un computer dedicato ad ogni singolo concorrente. Ogni concorrente ne mantiene un'istanza che utilizza per aggiornare le statistiche di checkpoint in checkpoint. Più precisamente, ogni concorrente possiede un'istanza di **Computer**, un record che colleziona le statistiche del singolo concorrente

e le informazioni statiche sulla configurazione della competizione.
Ogni **Computer** mantiene inoltre un riferimento ad una risorsa che nel diagramma è **Synch_Info_For_Box**. Tale risorsa viene utilizzata per mantenere la lista delle informazioni necessarie al box. È più che altro un meccanismo di ottimizzazione per avere gli aggiornamenti sul singolo competitor immediatamente disponibili quando il *Box* ne richiede.

Il *Competitor* utilizza il metodo AddInfo di **OnBoardComputer** per inviare le informazioni relative al tratto appena attraversato. Da quando tali informazioni sono sottomesse, vengono effettuati i seguenti passaggi:

- se la fine del tratto coincide con la fine del settore, viene verificato il tempo impiegato per attraversare tale settore (facendo riferimento anche ai dati passati) e se migliore di quello precedentemente salvato (in **Computer**), viene aggiornato quello vecchio.
- se la fine del tratto coincide con la fine del settore vengono anche aggiornate le informazioni in **Synch_Info_For_Box**. Si ricorda infatti che il *Box* riceve le informazioni aggiornate alla fine di ogni settore.
- se la fine del tratto corrisponde con la fine del giro, viene aggiornato il miglior tempo di giro come fatto per i settori.
- una volta effettuati tutti i controlli, le informazioni vengono impacchettate e inviate a **Competition_Computer** per ulteriori controlli e per essere salvate.

- **Competition_Computer**:
  è invece il computer dedicato al calcolo delle statistiche globali, ovvero riguardanti tutti i partecipanti. Qualunque informazione che riguardi uno o più concorrenti è da richiedere a questa entità. Il **Competition_Computer** si appoggia a due risorse per l'archiviazione e l'organizzazione dei dati:

  All_Competitor_Stats_Collection: la risorsa è destinata a mantenere la storia degli aggiornamenti di tutti i concorrenti. Ad ogni competitor è dedicato un array di **Synch_Competitor_Stats_Handler**. Questa entità serve ad racchiudere un **Competitor_Stats** nel corpo di una risorsa protetta. L'array è inizializzato con capacità pari a N° Lap * N° Checkpoint, poichè le informazioni vengono aggiunte ad ogni checkpoint. Ogni posizione dell'array quindi fa riferimento ad un checkpoint e l'array è intrinsecamente ordinato per tempo crescente.

Synch_Competitor_Stats_Handler mette a disposizione un entry di get (Get_All) che si apre solo nel momento in cui la risorsa è inizializzata. Questo permette di mettere in attesa i client che richiederanno informazioni non ancora disponibili.

Placement_Handler: come dice il nome, questo package è pensato per gestire la classifica. Al suo interno è presente **SOCT_Array** che è la parte più alta di una struttura utilizzata per il supporto alla creazione della classifica. Nel punto più basso c'è **Placement_Table**, unità finalizzata a raccogliere i tempi di lap dei concorrenti. A gestire questa tabella virtuale c'è
Synch_Ordered_Placement_Table, che permette di mantenere la tabella ordinata in base ai tempi e offre un set di metodi per il reperimento dei dati. Infine **SOCT_Array** (SOCT sta per Synch Ordered Placement Table) è un array in cui ogni posizione rimanda ad alla tabella della classifica della lap corrispondente all'indice.

Tramite il metodo Add_Stat, il **Computer** di **OnboardComputer** invia i pacchetti con gli aggiornamenti. Prima di salvare definitivamente un aggiornamento, viene verificato se fa riferimento ad un checkpoint di fine lap. In tal caso viene prima aggiornata la tabella della classifica corrispondente alla lap appena percorsa con l'istante di tempo segnato. Successivamente l'aggiornamento viene salvato nell'array di riferimento del concorrente, aprendo così la risorsa a chiunque la richieda o la stesse già richiedendo.

Quanto descritto costituisce le fondamenta di **Competition_Computer**. Il metodi offerti navigano queste strutture per ottenere i dati richiesti:

procedure Get_StatsByTime
(Competitor_ID : INTEGER;
Time : FLOAT;
Stats_In : out COMPETITOR_STATS_POINT);
fornisce il primo aggiornamento con tempo maggiore o uguale all'istante dato;

procedure Get_StatsBySect(Competitor_ID : INTEGER;
Sector : INTEGER; Lap : INTEGER;
Stats_In : out COMPETITOR_STATS_POINT);
fornisce le statistiche del concorrente COMPETITOR_ID inerenti alla fine del settore richiesto nella lap richiesta;

procedure Get_StatsByCheck(Competitor_ID : INTEGER;
Checkpoint : INTEGER; Lap : INTEGER;

Stats_In : out COMPETITOR_STATS_POINT);
fornisce le statistiche del concorrente COMPETITOR_ID inerenti al checkpoint e lap richiesti;

procedure Get_BestLap(TimeInstant : FLOAT;
LapTime : out FLOAT; LapNum : out INTEGER;
Competitor_ID : out INTEGER);
fornisce il miglior giro, il tempo di tale giro e il concorrente che fatto il record;

procedure Get_BestSectorTimes(TimeInstant : FLOAT;
Times : out FLOAT_ARRAY;
Competitor_IDs : out INTEGER_ARRAY;
Laps : out INTEGER_ARRAY);
fornisce il miglior tempo per ogni settore con i concorrenti che hanno fatto il record.

procedure Get_Lap_Placement(Lap : INTEGER;
TimeInstant : FLOAT;
CompetitorID_InClassific : out INTEGER_ARRAY_POINT;
Times_InClassific : out FLOAT_ARRAY_POINT;
LappedCompetitors_ID : out INTEGER_ARRAY_POINT;
LappedCompetitor_CurrentLap : out INTEGER_ARRAY_POINT);
dato l'istante di tempo in input, il metodo fornisce la classifica più aggiornata con i tempi per quell'istante, compresi i concorrenti doppiati in ordine di posizione e la lap che stanno percorrendo. I concorrenti che invece non sono doppiati ma devono ancora finire la lap a cui la classifica si riferisce non vengono inclusi nella lista.

Il **Competition_Monitor**, (come vedremo poi più in dettaglio è una sotto-componente di *Monitor*) fa riferimento al **Competition_Computer** per ottenere le informazioni di competizione.

B.1.5 Monitor

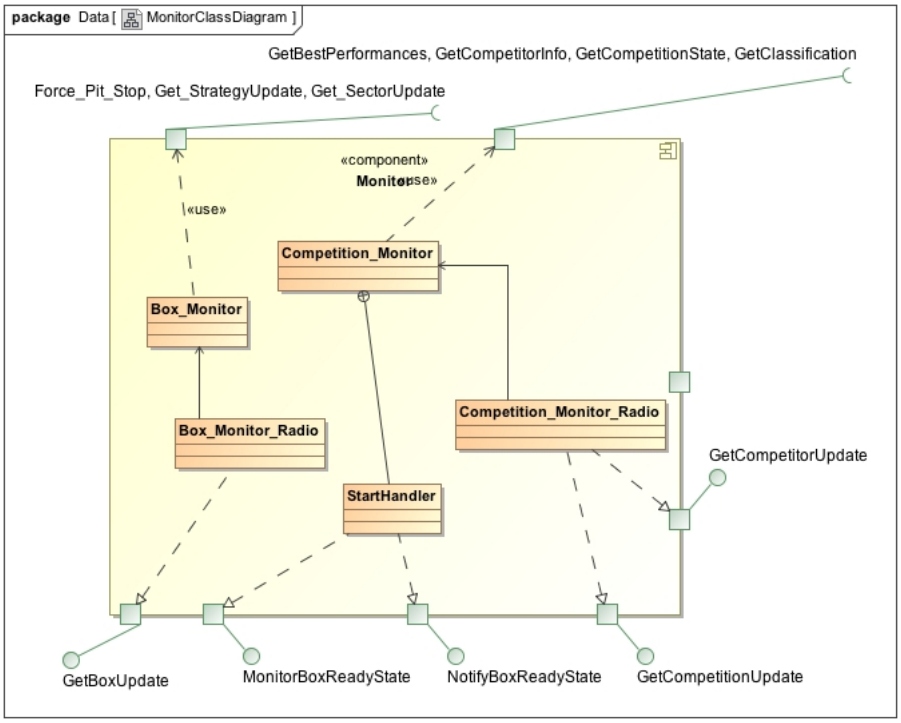

Figura 19: Class diagram - Monitor

Monitor si suddivide in due sotto-componenti: **Box_Monitor** e **Competition_Monitor**. La prima è dedicata alle informazioni esposte dal *Box*, l'altra a quelle esposte dalla *Competition*.
Box_Monitor produce i dati interrogando la componente *Box*, precisamente **Box_Data**, unità che vedremo in seguito, tramite il metodo Get_Info, grazie al quale si ottiene un oggetto che contiene le informazioni inerenti all'ultimo settore completato dal concorrente (comprese le medie di consumo) e, se disponibile, la strategia calcolata dal *Box* per la lap successiva.

Competition_Monitor si appoggia a *Stats* per reperire le informazioni richieste. I metodi che espone sono:

procedure Get_CompetitionInfo(TimeInstant : FLOAT; PlacementTimes : out Common.FLOAT_ARRAY_POINT; XMLInfo : out Unbounded_String.Unbounded_String);
la procedura inizializza la stringa con le informazioni di competizione in

formato XML. Tali informazioni riguardano il posizionamento dei concorrenti all'istante dato (quale tratto stanno percorrendo e in che lap), le migliori performance fino all'istante dato e la classifica aggiornata all'ultima lap in corso, con tempi e concorrenti doppiati. Il metodo si appoggia ai metodi forniti da **Competition_Stats** per reperire le informazioni necessarie.

procedure Get_CompetitorInfo(lap : INTEGER; sector : INTEGER ; id : INTEGER; time : out FLOAT; updString : out Unbounded_String.Unbounded_String);
la procedura fornisce le informazioni di un concorrente aggiornate al settore e lap richieste. Il metodo si appoggia a **OnBoardComputer** per reperire le informazioni date, precisamente utilizzando il metodo Get_BoxInfo utilizzando come parametro un riferimento al **Computer** del concorrente di cui si richiedono le informazioni.

Queste classi non comunicano direttamente con i loro clienti, ma comunicano tramite un intermediario: **Box_Monitor_Radio** e **Competition_Monitor_Radio**, ai quali viene demandato il compito di gestire le comunicazione distribuita tramite Polyorb (usando il protocollo CORBA).

B.1.6 Box

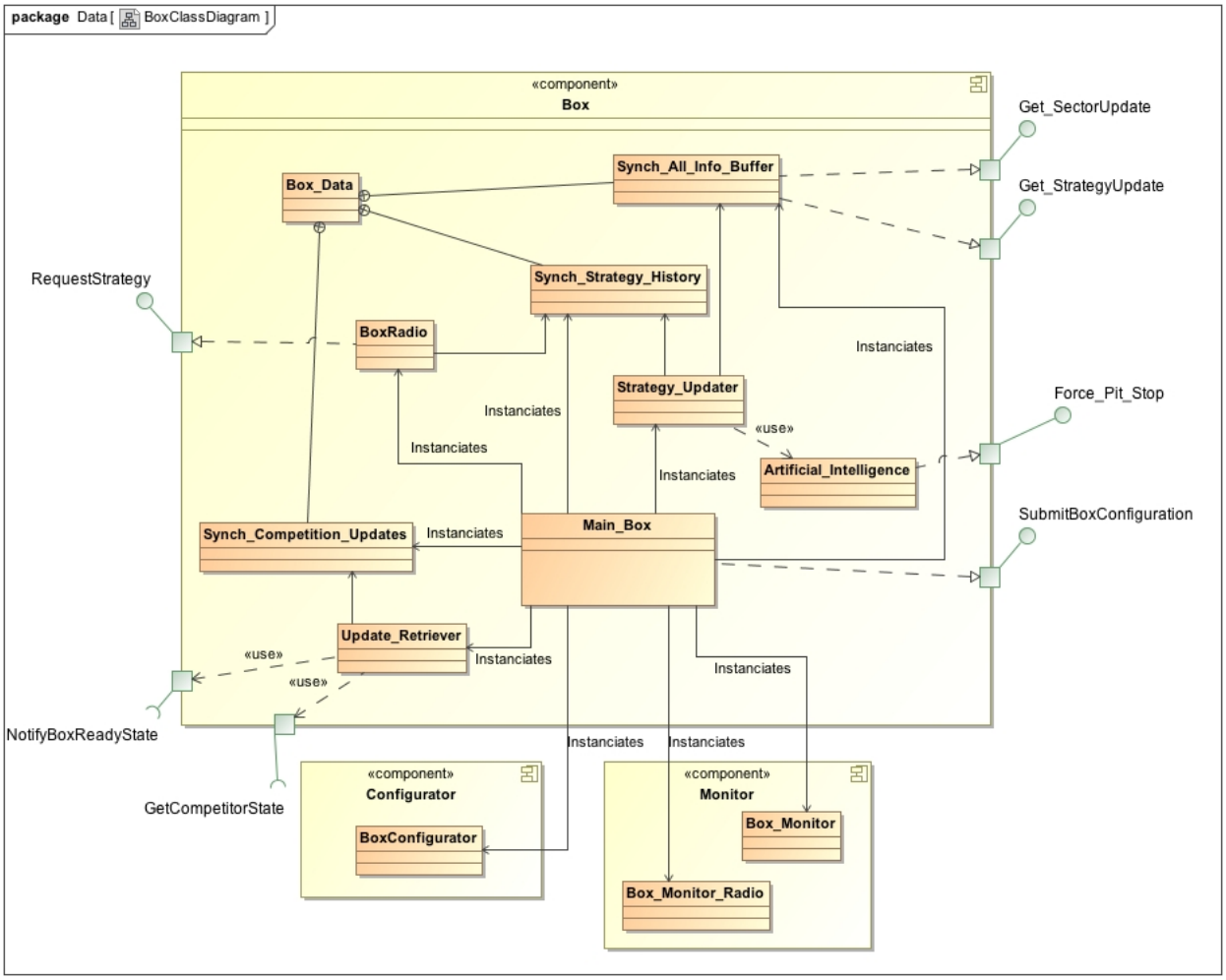

Figura 20: Class diagram - Box

Il *Box* è una componente che si occupa di assistere la gara del concorrete. A tale scopo sono stati previsti due task con le seguenti caratteristiche:

Update_Retriever:

Come il nome suggerisce, questa unità è finalizzata a recuperare gli aggiornamenti di gara relativi al concorrente associato al box. A tale scopo, il task si connette al **Competition_Monitor** per ottenere un pacchetto di informazioni aggiornato alla fine di ogni settore. Tali informazioni vengono convertite in un formato compatibile con il sistema (vengono ricevute

in XML e convertite in un oggetto **Competition_Update**). Fatto ciò, gli aggiornamenti vengono inseriti nel buffer **Synch_Competition_Updates**. Il task **Strategy_Updater** potrà così leggere gli aggiornamenti non appena disponibili (essendo tale buffer condiviso fra i due task).
Il task richiede le informazioni per lap e settore. Se un'informazione non è ancora stata prodotta, il thread verrà messo in attesa.

Strategy_Updater:
L'entità rappresenta una parte dell'"intelligenza artificiale" del sistema. Raccoglie gli aggiornamenti di gara dal buffer **Synch_Competition_Updates** man mano che essi vengono aggiunti. Tali aggiornamenti vengono di volta in volta usati per aggiornare le medie sui consumi e aggiornare la strategia di gara. L'aggiornamento della strategia avviene alla fine di ogni secondo settore appoggiandosi al package **Artificial_Intelligence**, disaccoppiato dal resto per poterlo modificare con facilità. Una volta ottenuto l'aggiornamento del secondo settore, vengono usate le informazioni relative al 3° settore del giro precedente e quelle relative al 1° e 2° settore del giro corrente per computare una nuova strategia. Una volta che la strategia è stata calcolata viene inserita nel buffer **Strategy_History**, da dove diventa disponibile nel caso il concorrente la richieda.

A supportare la comunicazione dei dati fra entità attive vi sono 3 risorse protette contenute nel package **Box_Data**:

Synch_Competition_Updates:
Supporta la comunicazione fra **Update_Retriever** e **Strategy_History**. Contiene le informazioni di settore del concorrente raccolte dal task **Update_Retriever**. **Strategy_History** reperisce le informazioni in ordine di inserimento.

Synch_Strategy_History:
Supporta la comunicazione fra **Strategy_History** e **BoxRadio**. Il primo aggiunge una nuova strategia alla fine di ogni secondo settore al buffer. Il secondo è un tramite fra *Competitor* e *Box* e permette al **Competitor_Task** di richiedere una strategia nuova alla fine di ogni lap tramite il metodo Get_Strategy. Se la strategia è presente, verrà tornata. Altrimenti il thread che si occupa di gestire la richiesta viene messo in attesa.

Synch_All_Info_Buffer:
Supporta la comunicazione fra *Box* e **Box_Monitor**.Questa risorse è destinata a contenere tutte le informazioni legate ai box. Oltre alle informazioni grezze di settore sul concorrente, la risorsa colleziona anche le

medie calcolate dai box e le strategie ogni qualvolta ne siano disponibili di nuove. Le informazioni sono ordinate cronologicamente e associate ad un indice. L'indice va da 1 a $3 * LapTotali$, poichè i settori sono tre. Se un'informazione relativa ad un settore e ad una lap non è disponibile, il richiedente viene messo in attesa fino a che l'aggiornamento non risulti disponibile.

Si può constatare che a concorrere per le informazioni collezionate nel *Box* non vi sono solo unità interne alla componente, ma anche thread che agiscono da altre componenti. Tali thread sono il **Competitor_Task** lato *Competition* e l'interfaccia di visualizzazione delle informazioni del box lato *Screen*. Il primo richiede una nuova strategia alla fine di ogni lap tramite **CompetitorRadio** (lato *Competition*) che si mette in contatto con **BoxRadio** che comunica direttamente con **Synch_Strategy_History** per recuperare la strategia richiesta o essere messo in attesa.
Il secondo (*Screen*) richiede tutte le informazioni dall'indice 1 all'indice massimo contenute in **Synch_All_Info_Buffer** tramite la parte del *Monitor* destinata ai box. Le informazioni tornate possono riguardare solo un aggiornamento di settore o anche di strategia, in base a quanto disponibile per quell'indice.

Infine il **Main_Box** è destinato ad inizializzare e configurare le risorse protette condivise e i task e istanziare gli oggetti ORB necessari alla comunicazione fra componenti.

B.1.7 Configurator

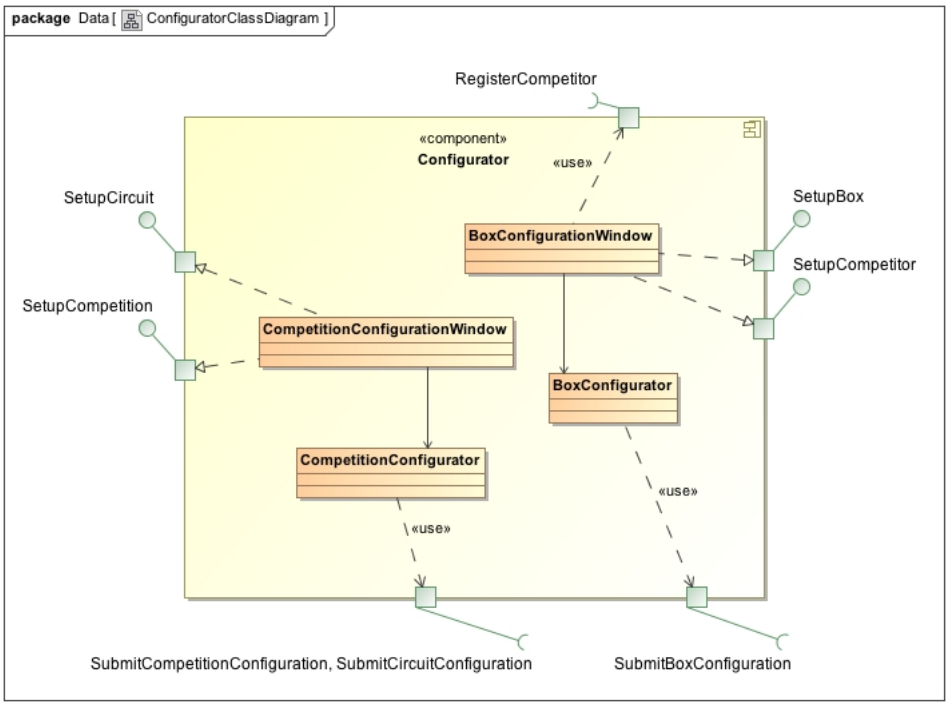

Figura 21: Class diagram - Configurator

La componente, come già accennato, gestisce la configurazione delle componenti *Box* e *Competition*. Può essere vista come suddivisa a sua volta in due ulteriori sotto-componenti, una destinata all'inserimento dati e l'altra destinata alla comunicazione dei dati alla logica sottostante. Vediamo infatti nel diagramma due classi **CompetitionAdminWindow** e **BoxAdminWindow** che offrono all'utente un'interfaccia di configurazione (scritta in Java) per *Competition* e *Box*; queste due classi sono legate ai rispettivi "configurator", ovvero classi che permettono l'intercomunicazione fra linguaggi diversi, di modo quindi che i dati inseriti nelle interfacce utente possano essere trasferiti alle componenti sottostanti per la configurazione e inizializzazione. In dettaglio:

CompetitionConfigurationWindow:

Utilizzata per impostare i parametri di gara, ovvero:

- numero lap
- numero concorrenti

- file da cui leggere il circuito

Una volta impostati i parametri essi verranno sottomessi via **CompetitionConfigurator** alla *Competition* per la configurazione e l'inizializzazione.

BoxConfigurationWindow:

Utilizzata per impostare le caratteristiche statiche del concorrente associato al box (fare riferimento al capitolo A.1.2 per informazioni su tali parametri), dettagli riguardanti il box stesso, ovvero la strategia (fare riferimento al capitolo A.1.5 per ulteriori dettagli) e la configurazione iniziale dell'auto, ovvero tipo di gomme e quantità di benzina iniziale. Queste informazioni vengono poi utilizzate per registrare il concorrente alla competizione (tramite il **RegistrationHandler** della componente *Competition*). Una volta registrato il concorrente si potranno ottenere il resto delle informazioni necessarie ad inizializzare il box, ovvero numero di giri e lunghezza circuito (per esempio). È quindi ora possibile inviare i parametri di configurazione al *Box* tramite il **BoxConfigurator**.

B.1.8 Screen

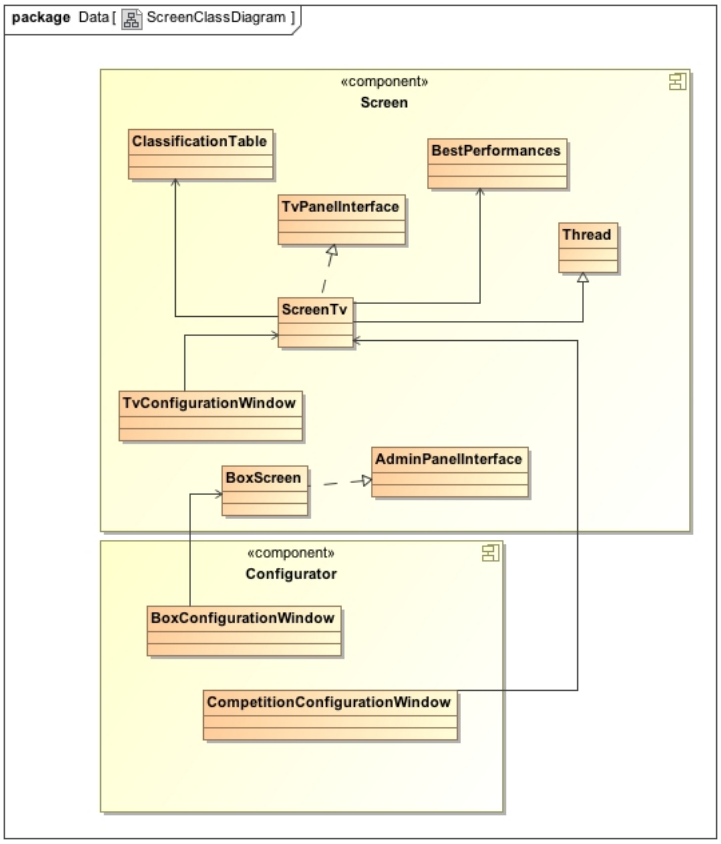

Figura 22: Class diagram - Screen

BoxScreen:

La classe **BoxScreen** visualizza l'andamento della competizione da parte del concorrente a cui si riferisce.Questa finestra viene avviata dopo che il concorrente si è registrato alla competizione. Una volta avviato si mette in ascolto sul *Box_Monitor_Radio* che ritorna (quando disponibili) le informazioni relative all'andamento del proprio concorrente, visualizzando tempi, posizione sulla pista, usura delle gomme e livello della benzina.

ScreenTv:

La classe **ScreenTv** è l'implementazione dell'interfaccia **TvPanelInterface** e si occupa di offrire una visualizzazione dei dati di gara. Quando viene creato lo ScreenTv la connessione con il monitor di sistema è già stata effettuata ed è quindi funzionante. Tramite il riferimento al monitor lo ScreenTv è in grado di ottenere inizialmente le informazioni statiche della

gara tramite il metodo Get_CompetitionConfiguration e utilizzarle per costruire la GUI. Successivamente si mette in ascolto sul monitor aspettando l'iscrizione dei concorrenti.Ad iscrizione avvenuta vengono stampate le informazioni relative ai concorrenti. Successivamente, quando parte la gara, lo screenTv comincia ad acquisire informazioni dal monitor. La richiesta viene effettuata ogni intervallo di tempo (fissato per lo screen della competizione, configurabile per lo screen di una tv generica). Ogni volta che viene richiesto un aggiornamento possono presentarsi due situazioni. La prima è la disponibilità delle informazioni che quindi vengono lette e la gui viene così aggiornata con i valori opportuni, la seconda è la non disponibilità di aggiornamenti, in tal caso lo screenTv rimane in attesa.

B.2 Analisi della concorrenza

B.2.1 Interazione Competitor - Circuit

La strategia adottata per permettere ai concorrenti di percorrere la pista è stata descritta ad alto livello nel corso della relazione. Tale strategia è indipendente dal linguaggio o dall'implementazione, quindi il codice scritto ha semplicemente tradotto la teoria in pratica. Gli aspetti legati all'implementazione a cui si è dovuto prestare attenzione invece sono descritti di seguito:

Accesso simultaneo ai checkpoint
: Per garantire che il checkpoint sia acceduto in modo mutamente esclusivo fra task, è stato sufficiente inserirlo in una risorsa protetta: **Synch_Checkpoint**. Tale risorsa offre i metodi necessari ai **Competitor_Task** per segnalare il proprio arrivo sulla coda, "scrivere" il tempo di arrivo limite ecc. in modo da evitare race condition.

Prima posizione raggiunta
: Come si è visto, quando un concorrente arriva ad un checkpoint deve segnalare il suo arrivo effettivo e, una volta raggiunta la prima posizione nella coda del checkpoint, iniziare a valutare il path da percorrere. Per mettere in pratica il sistema progettato, è stato necessario avvalersi di un'ulteriore risorsa protetta chiamata **Waiting_Block**. Ne viene istanziata una lista in ogni **Synch_Checkpoint** (una lista lunga quanto il numero di concorrenti). Ogni elemento della lista è associato ad un concorrente. Quando un concorrente necessita di attendere su una qualunque condizione, viene messo in attesa sul suo rispettivo **Waiting_Block**. Quando la condizione si verifica, il thread che ha causato il cambio di condizione invoca il metodo Notify nel **Waitin_Block**, che risveglia il thread che era in attesa. La condizione per cui viene utilizzata questa struttura è il raggiungimento

della prima posizione nella coda del checkpoint. Il seguente diagramma descrive la sequenza di eventi che portano il **Competitor_Task** da segnalare il suo arrivo al checkpoint all'ottenere i path tra cui scegliere per effettuare l'attraversamento del tratto.

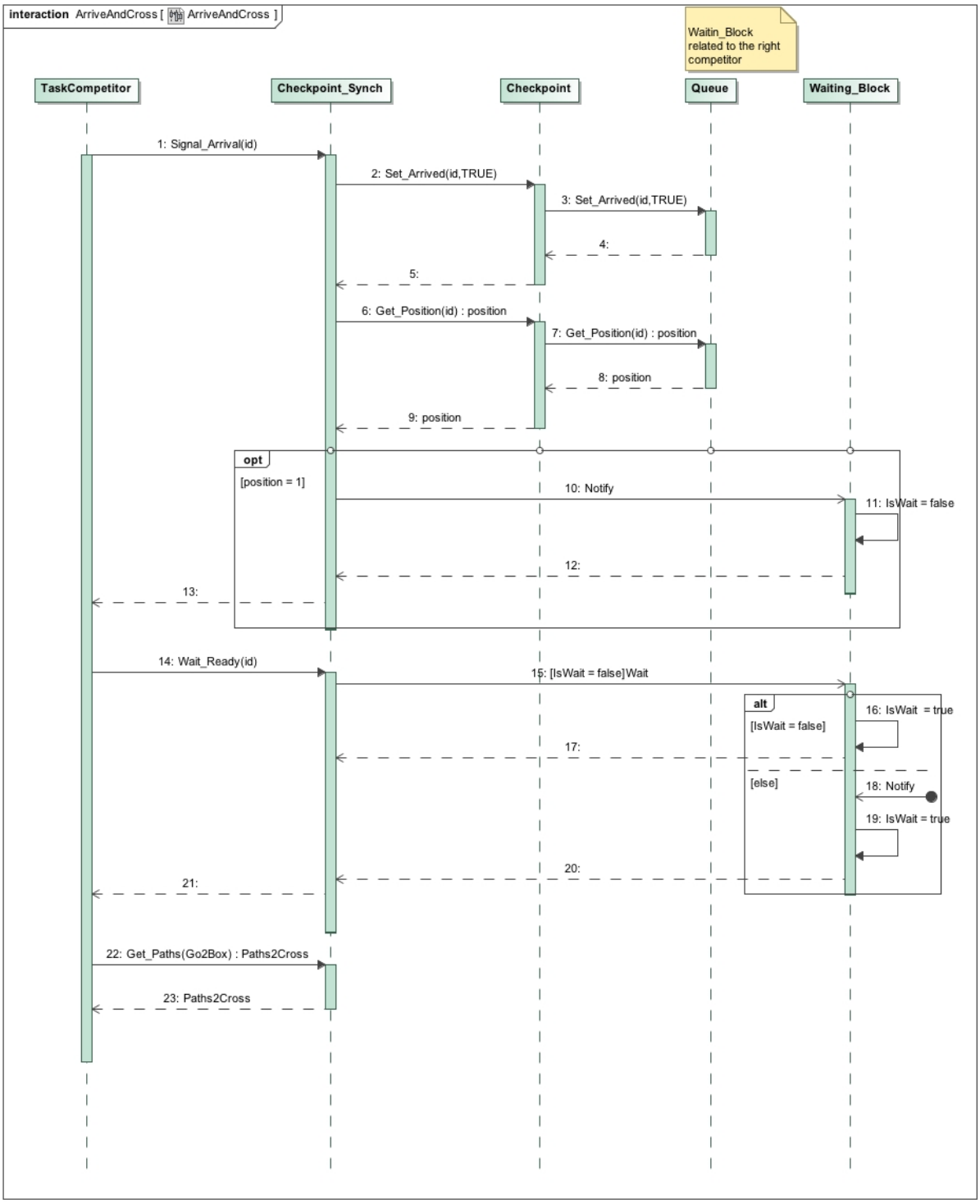

Figura 23: Sequence Diagram - Arrivo al checkpoint e recupero lista traiettorie

Prima di passare alla spiegazione del diagramma è necessario premettere in quali casi un concorrente può cambiare stato nella coda del **Synch_Checkpoint**:

- CAMBIO POSIZIONE: può avvenire quando un qualunque concorrente segna un nuovo tempo minimo (o effettivo) di arrivo con Set_ArrivalTime o quando un concorrente viene rimosso dal checkpoint con il metodo Remove_Competitor;
- DA "IN ARRIVO" AD "ARRIVATO" E VICEVERSA: avviene su invocazione del metodo Signal_Arrival, quando il concorrente arriva (passando da "in arrivo" ad "arrivato") o del metodo Signal_Leaving, quando il concorrente ha finito ed è pronto a lasciare il checkpoint (passando da "arrivato" a "in arrivo").

Quando il **Competitor_Task** segnala il suo arrivo (a partire dalla chiamata **1**), viene segnato l'arrivo effettivo all'interno del checkpoint. Successivamente viene controllata la posizione del concorrente in coda (chiamate dalla **6** alla **9**). Se il concorrente è primo, è necessario segnalarlo al rispettivo **Waiting_Block**, anche se il thread non si è ancora messo in "wait" (giusto per aprire la guardia). Successivamente il thread del **Competitor_Task** attende il suo momento con Wait_To_Be_First. Se era già primo, il metodo ritorna subito, dando la possibilità al concorrente di ricevere la lista di path (sicuro che nessun altro in quel momento la starà valutando). Altrimenti il thread viene messo in attesa nel **Waiting_Block** relativo al concorrente. Il thread viene risvegliato nel momento in cui un altro **Competitor_Task** modifica il suo istante di arrivo limite (facendo in modo che il concorrente in attesa venga a trovarsi nella prima posizione della coda) o quando viene rimosso dal checkpoint. Eseguendo una qualunque delle azioni accennate nella premessa, il **Competitor_Task** verifica quindi la presenza di un concorrente effettivamente arrivato in prima posizione nella lista ed effettua una Notify sul relativo **Waiting_Block** (nel diagramma viene espresso come un evento esterno al punto **18**). In questo modo il concorrente in attesa può procedere alla valutazione.

B.2.2 Consumatori esterni - Stats

La sotto-componente **Competition_Monitor** espone diverse viste riguardanti i dati relativi alla competizione. Tali dati vengono acceduti in modo "simultaneo" da qualunque entità esterna lo richieda. Viene creato quindi un thread per ogni richiesta proveniente da un nodo distribuito (i *TVScreen* per esempio, di cui ne potrebbero esistere potenzialmente decine o più). La tecnica adottata per affrontare il problema ha alla base l'idea che tutte le statistiche vengono calcolate a partire da dati "grezzi" relativi ai singoli concorrenti. Essendo ogni dato riferito ad un istante e ad una specifica posizione nelle competizione (lap numero 2 e checkpoint 5 per esempio), è possibile calcolare qualunque statistica che sia riferita ad un istante o ad una posizione (es: i tempi di ogni concorrente alla fine della lap 4).
Ad ogni concorrente è assegnato un array (in *Stats*) in cui ogni posizione è destinata a contenere un aggiornamento. Gli aggiornamenti sono cronologicamente ordinati. È quindi possibile richiedere la posizione del concorrente nel circuito all'istante t semplicemente scorrendo l'array. Ogni informazione è contenuta in una risorsa protetta la cui lettura viene aperta solo nel momento in cui l'aggiornamento relativo viene inserito. Quindi se viene richiesta un'informazione non ancora presente, il thread richiedente semplicemente rimarrà in attesa.
Volendo ottenere lo stato di gara all'istante t sarà quindi sufficiente ciclare sui concorrenti chiedendo la loro posizione in tale istante, e quando tutti i concorrenti avranno aggiunto l'aggiornamento richiesto sarà possibile avere uno snapshot della gara in tale istante.
Per la classifica è stata aggiunta una struttura dati di supporto finalizzata a venire popolata con i dati di fine lap di ogni concorrente (**Synch_Placement_Table**).

B.2.3 Risorse a due consumatori

Ci sono infine numerosi casi in cui una risorsa protetta è acceduta dai classici consumatore e un produttore. Accesso simultaneo di due risorse attive al massimo dunque. È il caso, ad esempio, delle informazioni relative al box, **Synch_All_Info_Buffer**. Tale risorsa viene valorizzata da **Strategy_Updater** e letta dallo screen relativo al box (tramite **Box_Monitor**). Come visto nella sezione B.1.6, il consumatore accede alla risorsa per indice tramite Get_Info. Quindi, se l'informazione di indice richiesto non è ancora stata inserita, la sentinella READY verrà impostata a FALSE e il thread **riaccodato** in Wait con la guardia READY=TRUE. Quando il produttore inserisce una nuova risorsa, verrà impostato READY a TRUE, di modo che l'eventuale thread in attesa possa verificare se l'informazione voluta sia stata inserita. In caso affermativo l'informazione viene ritornata. In caso negativo il thread riesegue la stessa procedure

descritta inizialmente. Chiaramente questa tecnica non potrebbe funzionare con più consumatori, poichè la guardia potrebbe venire cambiata più volte ad insaputa di un thread che stia venendo **riaccodato** da Get_Info a Wait (la **requeue** infatti prevede prerilascio). La soluzione però si presta a risolvere il problema per tutte le situazioni in cui non più di un consumatore e un produttore accedano alla stessa risorsa.

B.2.4 Conclusioni

Quanto descritto riguarda i casi più problematici di gestione dell'accesso concorrente a risorse. Il resto dei casi viene trattato con semplici risorse protette e **entry** con guardie. Non si presenta quindi il rischio di racecondition. Nemmeno starvation dovrebbe essere possibile, a meno che per qualche motivo un thread relativo ad un **Competitor_Task** non si interrompa inaspettatamente, bloccando il processo di aggiornamento delle statistiche e quindi la produzione di snapshot di gara.

B.3 Distribuzione

Nella seguente sezione verranno spiegati dettagli fondamentali riguardanti la distribuzione nel sistema.

B.3.1 Componenti distribuite

Si è deciso di permettere la distribuzione delle seguenti componenti:

- ***Box***:
  La componente *Box* è possibile avviarla in un nodo separato rispetto alla competizione. È necessario infatti fornire il **corbaloc** della competizione per poter inizializzare tale componente. La scelta è stata suggerita da due motivazioni. Una, puramente pratica, riguarda un potenziale problema di sovraccarico computazionale. Il *Box* infatti, anche se per ora adotta una *AI* con vincoli abbastanza rilassati, potrebbe richiedere una potenza di calcolo superiore per effettuare dei calcoli estremamente complessi. Si pensi per esempio se venisse adottato un sistema ad apprendimento automatico o un algoritmo di ricerca della soluzione ottima ad albero per ogni scelta che il *Box* deve prendere. Avere la possibilità di poterlo isolare in una macchina dedicata potrebbe significare un notevole miglioramento nelle prestazioni.
  La seconda motivazione invece è legata alla realtà: i box normalmente comunicano via radio con i propri piloti. Inserire lo strato di comunicazione remota fra *Competitor* e *Box* è sembrato quindi un buon inizio per dare

maggiore credibilità alla simulazione. I problemi di comunicazione fra le radio di box e pilota si possono tradurre in problemi di rete fra i due nodi ospitanti *Box* e *Competitor*.

- ***TVScreen***:
  Le TV da cui poter osservare il procedere della gara possono essere inizializzate ovunque, a patto di essere in possesso del **corbaloc** del monitor di competizione. Il fatto che le TV (come nella realtà) potrebbero essere numerose, è parso sensato pensarle eseguite in nodi differenti da quello della competizione. Anche in questo caso per evitare il sovraccarico computazionale della macchina ospitante la competizione. Come per i box, se la TV avviata è una e a solo output testuale (come quella presentata nel progetto) il problema non si presenterebbe comunque. Ma volendone avviare molte a volendo magari in un futuro estendere le funzionalità del sistema aggiungendo un overlay grafico più interessante alle TV, tornerebbe molto utile in termini di efficienza poterle eseguire in nodi dedicati.

B.3.2 Interazione fra le componenti distribuite

Le componenti comunicano la maggior parte dei dati appoggiandosi a protocolli XML. Ogni set di informazioni scambiato fra componenti ha il suo schema dedicato. In questo modo si può avere un formato di rappresentazione dei dati standard, più facilmente manipolabile con le librerie e le conoscenze disponibili. Dati che richiedono una precisione maggiore, ovvero gli istanti temporali espressi in FLOAT (che possono crescere notevolmente all'aumentare del tempo di gara), vengono trasmessi utilizzando il tipo FLOAT fornito dal linguaggio IDL e, per sequenze di lunghezza non conosciuta a priori, utilizzando il tipo IDL definito come segue

```
typedef sequence<float> float_sequence;
```

Tale tipo viene poi manipolato in modo diverso in base al linguaggio utilizzato per la componente, rimanendo comunque entro un buon margine di precisione.

B.3.3 Misure di fault tolerance

Cavi di rete danneggiati o router intasati potrebbero provocare il malfunzionamento della comunicazione fra le componenti distribuite. Nel caso di malfunzionamento della comunicazione fra **TVScreen** e *Competition*, la competizione non viene compromessa. Nella peggiore delle ipotesi il **TVScreen** non è in grado di ottenere i dati richiesti e ritenterà in seguito. Nel caso uno o più dei nodi ospitanti i *Box* cada o rimanga isolato, la competizione rischierebbe di essere compromessa. In prossimità del checkpoint prima dei box infatti il concorrente

effettua una richiesta remota al *Box* per l'aggiornamento della strategia. Se tale richiesta produce errore, il concorrente rischierebbe di non poter procedere la gara. Si è deciso quindi di gestire l'arrivo di dati corrotti o la scomparsa di connessione semplicemente facendo mantenere al concorrente la stessa strategia adottata al giro precedente.
Se la connessione venisse invece a mancare temporaneamente al *Box*, esso rimarrebbe in attesa che la connessione ritorni per procedere all'aggiornamento delle strategie.

C Glossario

A

AUTO: l'entità utilizzata dal pilota per correre sul circuito.

B

BOX: il box rappresenta il supervisore del concorrente. Calcola nuove strategie in base all'andamento della gara che vengono comunicate periodicamente al competitor. Decide inoltre il momento adatto per un pitstop (e tutti i dettagli legati ad esso).

C

CONCORRENTE: l'entità costituita da pilota e auto che partecipa alla gara correndo sul circuito.
COMPETIZIONE: è l'insieme dei giri, del circuito e del numero di concorrenti che partecipano alla gara.
CIRCUITO: la pista, composta da checkpoint, tratti (e traiettorie) e box.

M

MOLTEPLICITÀ DI UN TRATTO: determina quante auto possano stare contemporaneamente su un tratto.

P

PILOTA: colui che guida la macchina.

S

SETTORE: ve ne sono tre per ogni circuito (come da regolamento) e rappresentano un segmento specifico del circuito. STILE DI GUIDA: caratterizza

il modo di guidare del pilota. Uno stile di guida più aggressivo porterà il pilota a tenere una velocità più alta.
STRATEGIA: viene utilizzata per determinare lo stile di guida del concorrente ad ogni giro e per ordinare un pitstop in caso
di necessità. Addizionalmente, se avviene un pitstop, fornisce informazioni sul tempo di stop e la nuova configurazione dell'auto.

T
TRATTO: un segmento della pista a caratterizzato da angolo, lunghezza e molteplicità.
TRAIETTORIA: rappresenta uno dei possibili percorsi all'interno di un tratto. Ogni tratto ha un numero di traiettorie pari alla molteplicità. Tali traiettorie usualmente differiscono per angolo e lunghezza rimanendo entro i limiti del tratto.

D Manuale utente

D.1 Prerequisiti

Il programma può essere eseguito solo su macchine Unix-like. Inoltre, per la compilazione, è necessario disporre delle seguenti componenti:

- gnatmake 4.4.3
- gcc 4.4.3
- librerie **XmlAda 3.2w** installate con comando xmlada-config eseguibile anche senza privilegi di root.
  In caso contrario, scaricare il pacchetto XMLAda da
  `http://libre.adacore.com/libre/download2/`
  e, dopo la decompressione, seguire le istruzioni incluse per la compilazione ed installazione.
- librerie **PolyORB GPL 2009-20090519 (rev. 144248)** installate con comando polyorb-config eseguibile anche senza privilegi di root
  In caso contrario, scaricare il pacchetto PolyORB da
  `http://libre.adacore.com/libre/download2/`
  e, dopo la decompressione, seguire le istruzioni incluse per la compilazione ed installazione.

La versione indicata delle librerie è quella usata per i test. Non si esclude tuttavia che il programma possa funzionare anche con versioni appena precedenti o successive.

D.2 Installazione

Dalla directory radice, eseguire i seguenti comandi:

- *make competition*
  per compilare la componente destinata a ospitare la competizione
- *make box*
  per compilare la componente destinata a eseguire il box
- *make tv* per compilare la componente che verrà usata per la TV

Le tre componenti hanno make diversi perchè possono essere compilate ed eseguite su macchine separate, comunicando remotamente.

D.3 Avvio

Come per la compilazione, dalla directory radice eseguire il comando desiderato fra i seguenti:

- *./start_competition.sh*
  avvierà la competizione. Apparirà il pannello di configurazione (del quale verranno forniti dettagli in seguito)
- *./start_box.sh n*
  avvierà un numero n box. Per eseguire un test in locale è possibile avviare tanti box quanti quelli stabiliti durante la configurazione della competizione. In tal caso, l'interfaccia di configurazione dei box presenterà nello spazio riservato al corbaloc per la connessione alla competizione, il corbaloc della competizione caricato dal contesto locale. Se invece si volessero avviare i box in altri nodi, il corbaloc per la registrazione può essere trovato in
  competition_registrationhandler_corbaLoc.txt
  a competizione configurata.
- *./start_tv.sh*
  inizializza una tv. Anche in questo caso, se il corbaloc della competizione è presente in locale, verrà impostato di default nel textbox dedicato. È comunque possibile reimpostarlo utilizzando il corbaloc salvato nel file
  competition_monitor_corbaLoc.txt
  a competizione configurata.

L'interfaccia TV è possibile copiarla senza necessità di ricompilazione in tutte le macchine ove la si voglia eseguire. E' sufficiente, una volta compilata, copiare il file *start_tv.sh* e la directory *obj/java/GUI* (mantenendo la gerarchia intatta) in un qualunque nodo distribuito ed eseguire lo starter.

D.4 Terminazione

Per terminare il programma (box, competizione o tv che sia) premere il tasto "x" in alto a destra sulla finestra.

D.5 Interfaccia box

Nella prima schermata che appare si può selezionare un file di configurazione da cui caricare i dati per i concorrenti. Se non si seleziona l'opzione si passa alla seconda schermata dove si possono impostare i parametri a mano, partendo dai valori di default. Le operazioni possibili sono:

1. Impostare il nome del concorrente
2. Impostare il cognome del concorrente
3. Impostare la scuderia del concorrente
4. Impostare il livello di serietà del box
5. Impostare la massima capacità del serbatoio
6. Impostare il livello di benzina iniziale
7. Impostare la mescola delle gomme montate (le gomme a mescola morbida si consumano più rapidamente)
8. Impostare lo stile di guida del concorrente
9. Impostare la massima accelerazione della macchina
10. Impostare la massima velocità raggiungibile
11. Inserire il corbaloc che fa riferimento al Registration Hanlder

Inoltre sono presenti un pulsante per avviare la competizione e uno per riportare la gui allo stato di default,

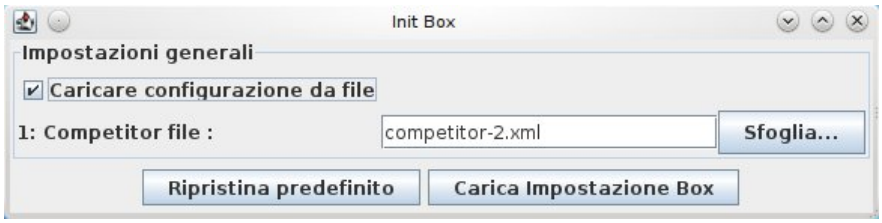

Figura 24: Finestra di pre - configurazione del box e del concorrente

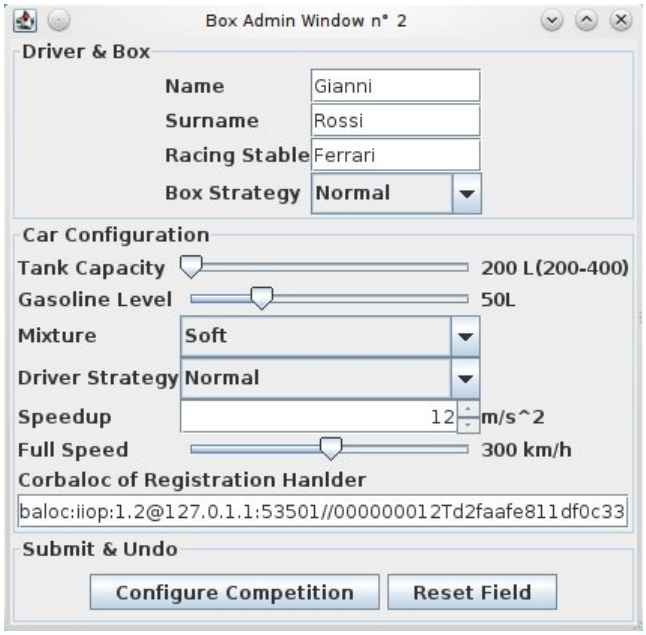

Figura 25: Finestra di configurazione del box e del concorrente

Una volta inseriti i dati e premuto il pulsante di avvio della competizione comparirà una finestra con al suo interno

1. Pannello dei consumi medi del concorrente con dati relativi alla benzina (in litri al kilometro) e dell'usura delle gomme (in percentuale relativo a 1 km)
2. Pannello con il log della gara aggiornato a ogni fine settore con dati di tempo di gara, livello di benzina e usura delle gomme
3. Pannello con altre informazioni statiche sulla gara (configurazione iniziale, stile di guida)

4. Pulsante per forzare il pitstop

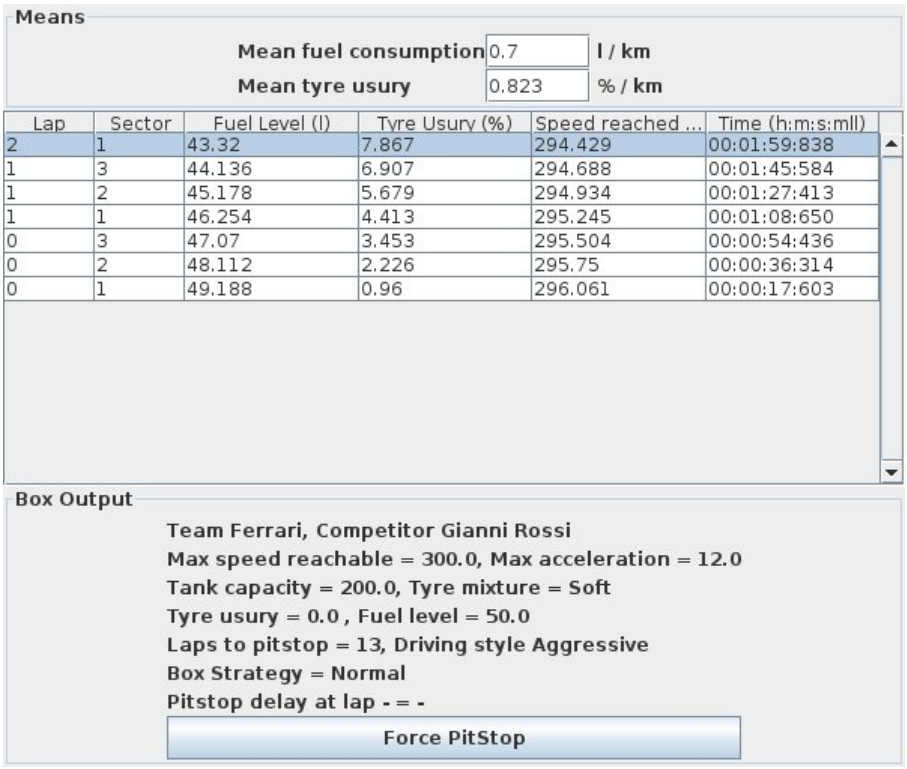

Figura 26: Monitor dei box

D.6 Interfaccia competizione

La prima interfaccia relativa alla competizione serve per impostare la competizione stessa. I parametri da settare sono:

1. File xml con il tracciato
   il formato per il file xml del tracciato è il seguente:

```
<racetrack>
  <sector>
      <checkpoint>
        <length>19.00</length>
          <mult>7</mult>
          <angle>140.00</angle>
          <grip>7.0</grip>
```

```
        </checkpoint>
        <!-- altri checkpoint-->
    </sector>
    <!--altri 3 sector strutturati allo stesso modo-->
</racetrack>
```

Il checkpoint può avere uno dei seguenti attributi:

- **goal="true"**: obbligatorio in massimo 1 checkpoint. Indica il checkpoint di partenza.
- **prebox="true"**: obbligatorio in massimo 1 checkpoint nel settore 3.
- **exitbox="true"**: obbligatorio in massimo 1 checkpoint nel settore 1.

Un requisito per il corretto funzionamento è di avere almeno 2 checkpoint per ogni settore, e i settori devono essere esattamente 3.

2. Numero di concorrenti richiesti
3. Numero di giri previsti
4. Nome del tracciato

Una volta settati i parameteri e schiacciato il pulsante *Start Competition* verrà avviato una tv che presenta le informazioni sulle iscrizioni nella parte bassa della finestra. Ogni volta che un concorrente si iscrive compare nella gui e una volta raggiunto il numero di concorrenti stabiliti la gara si avvia e la gui comincia ad offrire i dati disponibili, come spiegato nel paragrafo D.7

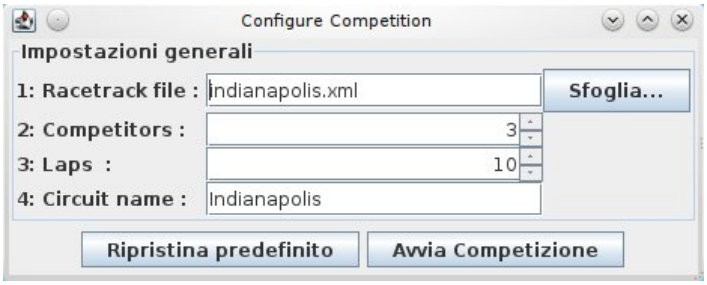

Figura 27: Finestra di configurazione della competizione

D.7 Interfaccia TV

L'interfaccia della tv consiste in un pannello iniziale con al suo interno un cronometro che scandisce il tempo di aggiornamento delle informazioni seguito da un pannello con le informazioni sul miglior giro e sui migliori tempi nei settori. L'intervallo per il tempo che scorre è impostabile se si tratta di una tv configurata tramite il *TvConfigurationWindow*, prestabilito (e molto basso) se si tratta della tv avviata dalla competizione. Il formato della stringa di refresh dev'essere *secondi* . *millisecondi*. Questo tempo che scorre è tempo relativo alla competizione e quindi solidale con il resto del sistema. Il pannello centrale offre la visualizzazione di due tabelle. La tabella a destra si riferisce all'ultima classifica disponibile mentre a sinistra viene visualizzata la classifica del giro precedente (escluso al primo giro di gara dove viene presentata una tabella vuota). Nelle classifiche verranno riportati i distacchi dal primo concorrent (di cui è riportato il tempo completo). Per i doppiaggi invece viene indicato il numero di giri di distacco. Nella classifica dell'ultima lap viene indicato invece il distacco temporale dal primo di tutti i concorrenti. Nella parte bassa della finestra viene presentato un log della competizione rappresentando a ogni istante di tempo posizione nella pista, numero di checkpoint, numero di settore e giro per ogni concorrente.

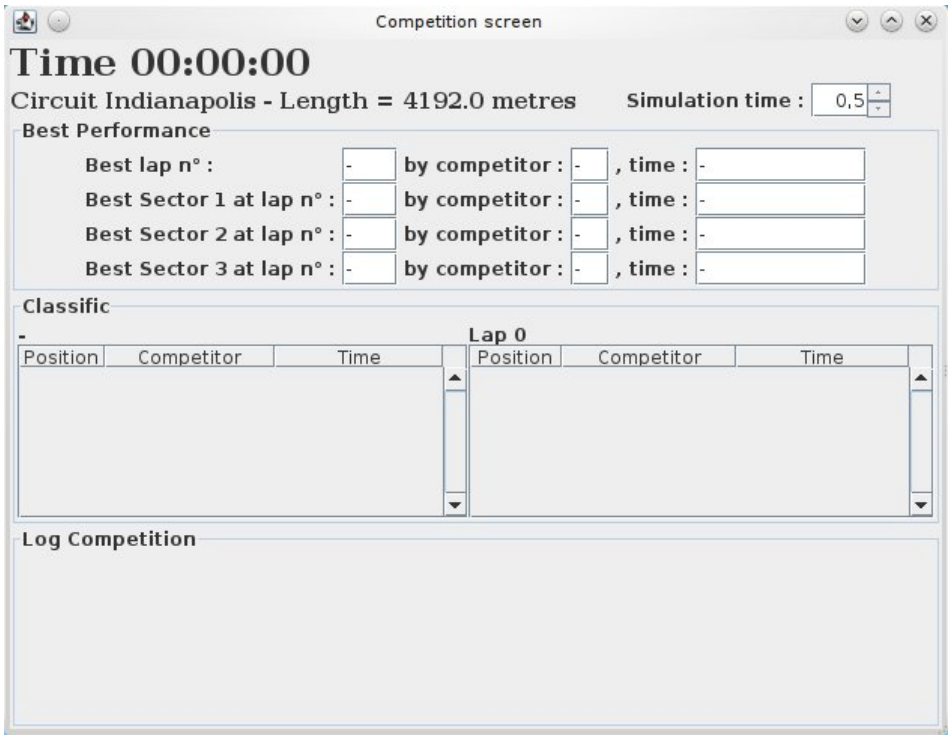

Figura 28: Finestra di visualizzazione dell'andamento della gara

La componente di avvio dell'interfaccia TV può essere l'interfaccia di competizione oppure una schermata di configurazione dove va inserito il corbaloc del Monitor della competizione e impostato il tempo di refresh per il reperimento delle informazioni. Il pulsante (presente solo nel monitor della competizione) presente in alto a destra permette di modificare il tempo della simulazione. Nel pannello con il log della competizione sono presenti tre campi per ogni concorrente che rappresentano rispettivamente i concorrenti davanti a lui, nello stesso tratto e dopo di lui.

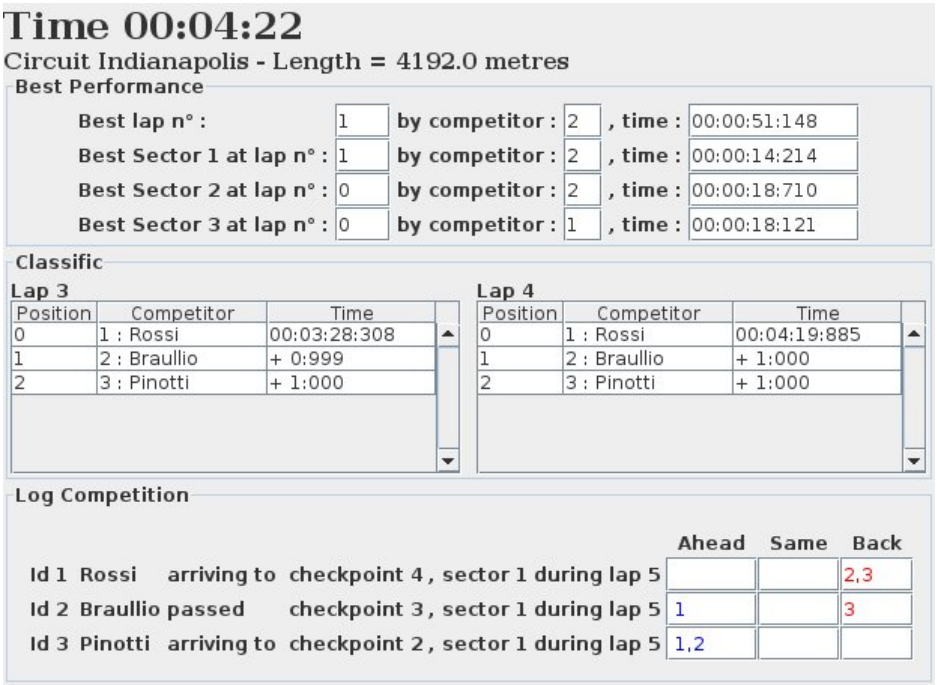

Figura 29: Finestra di visualizzazione dell'andamento della gara - durante la simulazione

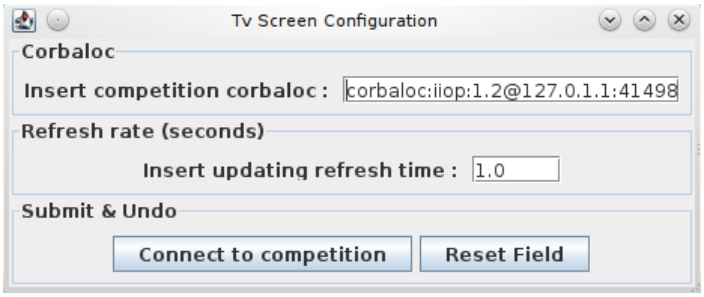

Figura 30: Finestra di configurazione della tv