Soccer-SCD

Simulatore di calcio - Progetto di Sistemi Concorrenti e Distribuiti

Sebastiano Catellani
Sebastiano Gottardo
Alessandro Sforzin

Universita' degli Studi di Padova
July 6, 2014

Indice

1 Introduzione

Il seguente progetto e' il frutto del lavoro svolto nell'ambito del corso di *Sistemi Concorrenti e Distribuiti*. Il progetto proposto è stato tratto da un concorso annuale chiamato *The Ada Way* ed organizzato da Ada-Europe. In particolare, viene fatto riferimento all'edizione del concorso 2010/2011, anno in cui il tema proposto è stato lo sviluppo di un simulatore di una partita di calcio in cui gli attori della partita, ovvero giocatori e arbitri, sono interamente controllati dal computer e devono agire in maniera indipendente e al tempo stesso concorrente. L'utente ha la possibilità di interagire con la partita in corso, assumendo il ruolo di allenatore delle squadre.

Questo documento ha lo scopo di descrivere l'approccio utilizzato per risolvere il problema e si concentra prevalentemente sulle problematiche di concorrenza e distribuzione incontrate durante lo sviluppo del software.

1.1 Scopo del progetto

Lo scopo del progetto consiste nel realizzare un software che simuli lo svolgimento di una partita di calcio secondo il relativo regolamento. Il software è presenta una componente principale, nella quale avvengono le dinamiche di concorrenza che vedranno le entità in gioco interagire tra di loro, ed alcune componenti di distribuzione, che interagiscono con la componente principale e che vengono pilotate dall'utente.

1.2 Struttura del documento

Il documento assume la seguente struttura:

- il Capitolo 1 consiste in una breve introduzione al progetto e al suo contesto
- nel Capitolo 2 viene proposta l'analisi delle problema e la successiva definizione di un modello per la soluzione
- nel Capitolo 3 si effettua un'analisi architetturale approfondita della soluzione, volta ad evidenziare le scelte relative agli aspetti di concorrenza e distribuzione
- il Capitolo 4 introduce brevemente l'intelligenza artificiale dei giocatori
- nel Capitolo 5 vengono descritte le scelte implementative adottate
- il Capitolo 6 serve come manuale utente e dettaglia le informazioni relative alla configurazione e all'esecuzione del software
- nel Capitolo 7 sono presentate le conclusioni

2 Analisi

Il software che questo progetto mira a realizzare ha lo scopo di creare una simulazione di una partita di calcio. Nella fase di analisi e' stata analizzata una partita di gioco reale, al fine di definire tutte le funzionalita' del software di simulazione, individuando cosi' con precisione il problema da risolvere. Verra' quindi delineato un modello per la soluzione, mettendo in luce le entita' coinvolte e il modo in cui esse interagiscono tra loro.

2.1 Struttura del software

Il software di simulazione e' costituito da tre diverse componenti, che prendono il nome (simbolico) di *Core*, *Field* e *Manager*.

Core Il *Core* e' il modulo centrale del software. Esso contiene tutta la logica di gioco e regola l'interazione tra le diverse entita che lo compongono. E' inoltre responsabile di gestire la comunicazione con i moduli esterni, ovvero *Field* e i due *Manager*, in quanto non dispone di una propria interfaccia grafica.

Field Questa componente rappresenta l'interfaccia grafica della partita. Attraverso di essa l'utente puo' iniziare una nuova partita, mettere in pausa quella corrente oppure chiudere del tutto la simulazione; in quest'ultimo caso, anche le componenti *Core* e le due istanze di *Manager* vengono terminate. Inoltre vi e' una rappresentazione grafica molto semplice del campo di gioco, della panchina, dei giocatori e della palla. Viene inoltre riportato il tempo trascorso dall'inizio della partita, unitamente al punteggio corrente delle due squadre. Infine, e' presente un registro di eventi che permette di ripercorrere gli eventi salienti della partita.

Manager L'ultima componente e' il *Manager*, che permette di controllare la rispettiva squadra, decidendo eventuali cambi di formazione e sostituzioni da effettuare. E' possibile sostituire un giocatore alla volta, fino ad un massimo di tre, mentre non c'e' limite ai i cambi di formazione che e' possibile fare. Le modifiche apportate dall'allenatore vengono applicate alla squadra in occasione della prima interruzione di partita dovuta ad un evento di gioco (ovvero, la messa in pausa della partita da parte dell'utente non conta).

All'avvio del software, prima dell'inizio della partita, viene data la possibilità di configurare la propria squadra e i propri giocatori tramite un'interfaccia grafica dedicata. In essa i giocatori sono ordinati per ruolo, in base alla formazione scelta, e ciascuno di loro ha una scheda a lui dedicata in

cui sono riportate le caratteristiche fisiche configurabili. L'interfaccia da la possibilità di generarle casualmente per il giocatore selezionato, di generarle casualmente per tutti i giocatori della squadra oppure di inserire manualmente un valore per ciascuna di esse. Questa scelta deve essere ponderata perche' le statistiche non sono piu' modificabili a partita iniziata. Anche la formazione iniziale della squadra e' configurabile e puo' essere scelta fra tre possibili formazioni standard utilizzate nel mondo del calcio. La partita inizia non appena entrambe le squadre sono state configurate.

2.2 Funzionalita' del software

In questa sezione vengono introdotte le funzionalità del software di simulazione, che verranno poi dettagliate nei capitoli successivi.

2.2.1 La partita

Il software di simulazione permette di giocare una o piu' partite, ciascuna divisa in due tempi, la cui durata è prefissata e non modificabile dall'utente. L'inizio del secondo tempo e' dettato dalla scelta dell'utente, cosi' da permettergli eventuali modifiche all'assetto delle squadre. Lo svolgimento di una partita segue le regole ufficiali di base del calcio, eccezione fatta per il fuorigioco, che non viene preso in considerazione. Vi sono due squadre, identificate dai colori rosso e blu. Ciascuna squadra è composta da 11 giocatori in campo e 7 riserve in panchina, sostituibili attraverso una delle interfacce grafiche offerte all'utente.

2.2.2 I giocatori

I giocatori sono gli attori principali all'interno della simulazione di una partita. Essi devono essere liberi di agire sul campo di gioco ed avere quindi a disposizione una serie di azioni possibili, al fine di portare la propria squadra alla vittoria. A ciascun giocatore sono assegnate alcune caratteristiche di gioco che ne determinano la "bravura", intesa come la buona riuscita o meno delle azioni che vuole effettuare: tanto piu' alto il valore, tanto piu' "bravo" il giocatore in quel particolare aspetto di gioco. Si consideri come esempio un difensore che decide di rubare palla all'attaccante della squadra avversaria: la buona riuscita dell'intervento sara' quindi determinata dalla bravura (un valore alto) nel contrasto del primo e dalla capacita' di scartare del secondo. Tali carattestiche sono riassumibili come segue e possono essere configurati prima dell'inizio della partita (Sezione **??**):

- attacco

- difesa
- parata (valido solo per il portiere)
- velocità
- precisione
- potenza
- contrasto

Ogni giocatore in campo deve essere in grado di decidere autonomamente, in base alla situazione di gioco ed alle direttive dell'allenatore, le azioni da effettuare. Queste si possono suddividere in tre gruppi: movimento nel campo, in ogni direzione, per raggiungere la posizione desiderata; interagire con la palla, quindi prenderla, spostarla con se', passarla e tirarla in porta; infine, operazioni che comprendono un altro giocatore, per contrastare o scartare un avversario.

2.2.3 L'arbitro

Il corretto svolgimento della partita viene garantito dalla presenza di un arbitro, che provvede ad interrompere il gioco non appena si verifica un'infrazione (ad esempio un fallo) e a farlo ripartire non appena le condizioni lo permettono. Inoltre, l'arbitro ha il compito di sancire l'inizio e la fine di ciascun tempo, facendo opportunamente entrare in campo ed uscire in panchina tutti i giocatori. Infine, egli monitora la posizione della palla per fermare il gioco quando esce o se finisce in porta, aggiornando cosi' il risultato della partita e riportando le squadre al centro del campo.

2.2.4 Il campo

Il campo da gioco deve essere suddiviso in celle, cosi' da permettere una migliore gestione del movimento dei giocatori ed evitare che due di essi si trovino nella stessa posizione contemporaneamente. Le dimensioni del campo rispettano le proporzioni dettate dalle regole ufficiali del gioco, e contano XXX celle in lunghezza e YY in larghezza. Le panchine di entrambe le squadre sono adiacenti al campo, dove sono posizionati tutti i giocatori o solamente quelli di riserva a seconda del caso in cui ci sia una partita in corso o meno. E' possibile assistere allo svolgimento della partita attraverso l'interfaccia grafica del campo di gioco, nella quale vengono visualizzate tutte le informazioni piu' importanti relativi alla partita (2.1). Lo stato della partita puo' inoltre essere modificato, mettendola in pausa e facendola

riprendere in un secondo momento, oppure terminarla prima della fine del tempo regolamentare. A partita terminata, è possibile iniziarne una nuova.

2.2.5 Allenatori e squadra

Gli allenatori hanno il compito di gestire le squadre ed i relativi giocatori in campo. Una decisione presa dall'allenatore si ripercuote sul gioco, andando a modificare le posizioni in campo dei giocatori ed il loro atteggiamento in fase offensiva e difensiva. Inoltre, entro i limiti imposti dalle regole di gioco, puo' effettuare delle sostituzioni tra un giocatore in campo ed uno in panchina. E' l'utente ad agire come allenatore, ogni qual volta lo ritiene opportuno, per effettuare le operazioni appena citate. In questo caso si necessita di una visione generale delle statistiche di ogni giocatore per rendere piu' facile prendere decisioni su quali sostituzioni attuare.

2.3 Il modello

Una volta definite la struttura generale e tutte le funzionalita' del software, e' possibile procedere ad identificare le entita' e la loro tipologia. Per ciascuna di esse verra' quindi stabilito se rappresenta un'entita' attiva, un'entita' reattiva oppure una risorsa protetta, ed il loro ruolo all'interno dello scenario in cui si svolge la partita. Si procedera' quindi a definire le interazioni che legano le suddette entita' l'una con l'altra, cosi' da poterne derivare un modello per il sistema.

Per entità attive si considerano componenti dello scenario che necessitano di un proprio flusso di controllo, eseguendo ciclicamente le proprie azioni in modo autonomo. Le entità reattive non possiedono un proprio flusso di esecuzione, ma sono in attesa su canali opportunamente esposti per reagire alle richieste da parte di altre entità. Infine, la risorsa protetta differisce dall'entità reattiva in quanto risulta meno complessa, ma come essa mette a disposizione meccanismi per garantire mutua esclusione ed accodamento condizionale. La scelta di utilizzo tra le utlime due categorie è tipicamente dettata dalla mole di lavoro che deve essere svolta in risposta ad una richiesta da parte di un'altra entita'.

2.3.1 I giocatori

I giocatori vengono identificati come entita' attive, quindi dotate di un proprio flusso di controllo che interagisce con le altre entita' in gioco. Essi consisteranno in un numero di task pari al quantità di giocatori in campo, e ripeteranno ciclicamente il proprio corpo di esecuzione fino al termine

della partia. Le decisioni che dovranno prendere saranno basate sullo stato circostante e sull'andamento dell partita. Nella restante parte di questo documento viene fatto riferimento al concetto di turno, inteso come singolo ciclo di esecuzione che viene ripetuto fino alla fine della partita. Tale turno si divide inoltre in piu' fasi, ma questo aspetto verra' ripreso in seguito nei prossimi capitoli.

2.3.2 Lo stato

In fase di analisi si e' evidenziato il fatto di come un giocatore abbia bisogno di apprendere lo stato di gioco per poter decidere la sua mossa successiva. Questo porta alla luce la necessita' di avere un luogo unico in cui detenere lo stato di gioco, ed i giocatori lo consultano ogni qual volta ne abbiano bisogno. Tale stato consiste principalmente nelle posizioni degli altri giocatori in campo e l'andamento del gioco, per esempio se la mia squadra e' in fase offensiva o difensiva.

Quello che un giocatore e' chiamato a fare e' quindi a grandi linee ottenere lo stato di gioco in un determinato istante, calcolare la propria prossima mossa ed andare ad aggiornare lo stato in base a quest'ultima decisione.

Queste considerazioni evidenziano la necessita' di aggiungere una nuova entita' che governi lo stato, che ne garantisca l'integrita' nel proseguire del gioco e garantisca ai giocatori di poter leggere lo stato e modificarlo secondo le mosse scelte.

2.3.3 Interazione tra giocatori: il controllore

I giocatori sono i principali attori del sistema. Un giocatore si sposta dentro e fuori dal campo, e' in grado di compiere diversi tipi di azioni (Sezione **??**), che decide sia sulla base del proprio stato che dello stato della propria squadra e della partita. Un aspetto fondamentale vede i giocatori, cosi' come avviene nel mondo reale, operare in maniera parallela e concorrente tra di loro: questo implica che i risultati delle loro azioni debbano riflettersi sullo stato di gioco, che come descritto in Sezione **??**, detiene tutte le informazioni che definiscono la partita in un dato istante. Si pone pero' un problema: i giocatori, per poter conoscere le informazioni che permetteranno loro di decidere la prossima azione da compiere, i giocatori devono accedere allo stato, che tuttavia e' unico. E' dunque necessario regolamentare l'accesso a tale risorsa, in maniera tale da preservarne la consistenza e la correttezza.

L'organo che si occupa di permettere l'interazione tra diversi giocatori e' il *controllore*. In quanto entita' centrale di controllo, i suoi molteplici compiti possono essere raggruppati come segue:

1. Permettere ai giocatori di accedere allo stato, sia per leggerne le informazioni sia per poterle modificare
2. Sequenzializzare gli accessi allo stato da parte dei giocatori, al fine di non creare inconsistenze sulle informazioni in esso contenute
3. Ricoprire il ruolo di arbitro di gioco "onnisciente" (verra' dettagliato in Sezione 2.3.4)

Come gia' accennato precedentemente, un giocatore decide la sua prossima mossa sulla base dello stato corrente della partita: ad esempio, trovandosi in possesso della palla nell'area avversaria e senza nessun giocatore a marcarlo, e' ragionevole che il giocatore decida di tirare per provare a realizzare un goal. Al tempo stesso, l'azione che egli compie si ripercuote sullo stato della partita, andandone a modificare una parte: di nuovo, se il giocatore segna, il punteggio della partita cambia e i giocatori ritornano in posizione di partenza per ricominciare il gioco. Questo meccanismo mette in luce una necessita' tale per cui le operazioni che alterano lo stato siano quanto piu' possibile sequenziali ed atomiche. Sebbene questo argomento verra' trattato ampiamente nel Capitolo 3, e' importante capire il motivo dell'importanza di queste due caratteristiche.

L'assunzione di atomicita' si rivela particolamente critica quando si assume di operare in condizioni di prerilascio dei processi. Sotto questa condizione, un processo correntemente in esecuzione puo' essere "temporaneamente fermato" (prerilasciato) in favore di un altro processo, che entra

quindi in esecuzione al suo posto; una simile condizione si verifica, ad esempio, se il processo corrente ha una priorita' inferiore di quello che vuole subentrare. Per spiegare come il prerilascio minacci la consistenza dello stato della partita, si consideri una situazione dove due giocatori avversari si contendono il possesso della palla, che giace inerte in mezzo a loro. Entrambi i giocatori leggono lo stato e avviano il processo di decisione della prossima azione. Tuttavia, il processo relativo al primo giocatore viene prerilasciato. Nel frattempo, il secondo giocatore conquista la palla e si muove verso la porta avversaria. Quando il primo giocatore torna in esecuzione si trova in uno stato inconsistente, in quanto lo stato e' cambiato senza che lui lo sappia. Ancora peggio, se il primo giocatore procede con la scrittura della sua azione, lo stato verra' modificato erroneamente, potenzialmente togliendo il possesso di palla al secondo giocatore.

Arrivati a questo punto, serve quindi definire come i giocatori vanno effettivamente a modificare lo stato di gioco. Il modello prevede che sia solamente il controllore ad effetuare le scritture vere e proprie sulla base delle azioni decise e sottoposte da parte dei giocatori. Tali richieste vengono valutate in modo sequenziale, andando cosi' a modificare lo stato con una azione alla volta evitando cosi' condizioni di inconsistenza sullo stato.

Come detto in precedenza, all'inizio di ogni suo turno un giocatore deve prendere coscienza di quale sia lo stato attuale di gioco, decidere la prossima mossa sulla base di quest'ultimo e sottoporla al controllore per modificare lo stato. Questo meccanismo, abbinato al fatto che i giocatore eseguono in modo potenzialmente parallelo e che ogni richiesta viene elaborata in maniera sequenziale, farebbe pensare ad una soluzione che prevede un accesso esclusivo al controllore da parte di un giocatore per tutta la durata del suo ciclo di esecuzione. L'idea alla base della soluzione e' quindi quella di scomporre la routine del turno come segue:

1. il giocatore richiede al controllore lo stato di gioco
2. viene effettuata una fase di computazione con la quale decide la prossima mossa
3. richiede al controllore di applicare l'azione scelta allo stato.

Si ha quindi che la fase di lettura e la fase di scrittura (primo e terzo punto) vengono effettuate "online", ovvero tramite accesso esclusivo al controllore; la parte di scelta della prossima azione avviene invece "offline", quindi senza la necessita' di interagire con esso. In questo modo si guadagna un parallelismo

potenziale nella seconda fase, permettendo cosi' ai giocatori di non interferire tra di loro nella decisione della prossima mossa.

2.3.4 Verifica sullo stato di gioco: l'arbitro

Affiche' una partita si svolga secondo le regole e le modalita' stabilite dal gioco del calcio c'e' bisogno di un arbitro che regoli l'andamento del gioco. Le mansioni dell'arbitro sono molteplici:

- Sancire l'inizio e la fine dei tempi di gioco (inizio primo tempo - fine primo tempo - inizio secondo tempo - fine partita)
- Fermare il gioco e farlo riprendere a seguito (e.g. una rimessa laterale)
- Segnalare eventuali irregolarita' da parte dei giocatori (e.g. un fallo)
- Tenere il conto dei gol segnati da entrambe le squadre, cosi' da decretare il vincitore alla fine della partita
- Gestire le richieste di sostituzione e di cambio di formazione da parte degli allenatori

L'arbitro deve quindi essere in grado di controllare tutte le mosse dei giocatori, cosi' come lo stato e la posizione della palla e la durata della partita fino a quel momento. Nella realta', il ruolo dell'arbitro e' assegnato ad un essere umano, che quindi non e' infallibile: si pensi ad esempio ad un fallo che viene commesso irregolarmente alle sue spalle mentre lui e' impegnato ad assegnare un calcio d'angolo. In questa simulazione si assume piu' semplicemente che l'arbitro sia "onnisciente", ovvero abbia la facolta' di analizzare ogni singola mossa di ciascun giocatore e della palla, in maniera da poter segnalare immediatamente ogni irregolarita' oppure fermare il gioco all'occorrenza.

Si ha cosi' che il controllore, descritto nel paragrafo precedente, ricopre anche il ruolo di arbitro. Questa decisione ha delle ripercussioni non solo nello svolgersi del gioco (l'arbitro e' onnisciente), ma anche nell'assegnazione delle risorse di calcolo. Infatti, se l'arbitro fosse soggetto agli stessi vincoli di esecuzione dei giocatori, andrebbe a concorrere assieme a loro per l'esecuzione sulla CPU come task a se stante; questa situazione non si verifica invece nel caso in cui sia il controllore ad essere anche arbitro, essendo l'entita' centrale che si occupa di eseguire a tutti gli effetti le mosse dei giocatori. Maggiori dettagli sull'implementazione dell'arbitro verranno esposti in Sezione 4.3.

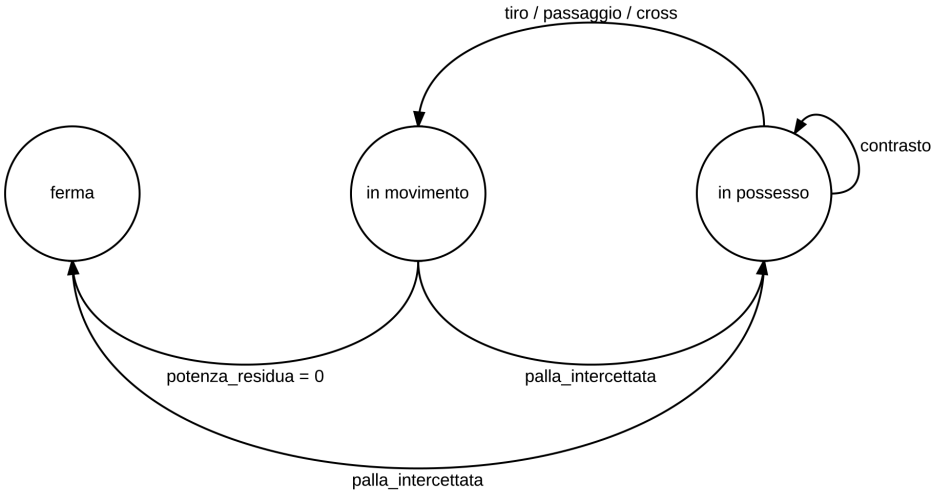
L'elenco delle funzioni che l'arbitro deve espletare nasconde tuttavia un punto critico su cui e' opportuno soffermarsi. I primi quattro punti

sono strettamente legati alla componente concorrente della simulazione, ovvero quella che vede l'interazione dei giocatori con il controllore e, in alcuni casi, l'agente di movimento. L'ultimo punto fa invece riferimento alla componente distribuita della simulazione, ovvero quella che si occupa di ricevere e gestire i comandi impartiti dagli allenatori ed eventualmente dalla finestra principale di controllo (che, come verra' esposto successivamente, coincide con la componente che mostra il campo di gioco e lo svolgersi della partita). La differenza sostanziale tra queste due tipologie di eventi e' racchiusa nel fatto che gli eventi provenienti dalla componente distribuita non sono deterministici e non seguono nessuna regola di generazione, a differenza degli eventi relativi alla parte concorrente. La presenza di due sorgenti di eventi introduce un problema significativo, ovvero l'ordine in cui l'arbitro deve processare/consumare quegli eventi. Ad esempio, si consideri una situazione dove si ha una richiesta di sostituzione per il giocatore 1 in favore del giocatore 3 e, allo stesso tempo, sia stato commesso un fallo commesso dal giocatore 1 sul giocatore 2. L'ordine in cui vengono processati questi eventi determina lo stato successivo, che diverge a seconda che si consideri prima uno oppure prima l'altro. Bisogna tenere conto, ad ogni modo, che gli eventi che vengono generati dalla componente distribuita hanno come precondizione il gioco fermo: quindi c'e' una stretta dipendenza unidirezionale tra un evento singolo della componente concorrente e gli eventi della componente distribuita. L'approccio da seguire per garantire il massimo livello di correttezza temporale e' quindi il seguente: l'arbitro dovra' prima processare l'evento singolo della componente concorrente (che puo' causare il gioco fermo, nel caso non lo fosse gia') e solo poi processare gli eventi della componente distribuita, se le condizioni sono opportune.

2.3.5 La palla in movimento: l'entita' e l'agente di movimento

La palla e' una parte fondamentale del modello della simulazione, in quanto il suo possesso viene conteso dai giocatori che devono tirarla in porta, segnando un gol per la loro squadra. Il suo comportamento puo' essere definito come una macchina a stati, schematizzato nel seguente diagramma:

In ogni momento della partita, la palla occupa una delle celle del campo. Nel caso sia posseduta da un giocatore, essi condividono la stessa cella; in caso contrario, la palla occupa la cella in cui si trova. Inoltre, la palla puo' trovarsi solamente in due stati: quiete e moto. Una palla in movimento si puo' avere quando il giocatore che la controlla si sposta con essa; inoltre, si ha una palla in movimento anche quando un giocatore la passa verso un altro giocatore oppure effettua un tiro verso la porta avversaria. Al contrario, una palla e' inerte se non e' controllata da nessun giocatore, solitamente quando un passaggio o un tiro mancano il bersaglio (e.g. un passaggio troppo debole).



La palla e' quindi una risorsa protetta, che non compie azioni proprie ma che subisce azioni di altre entita' attive (i giocatori) e salva di volta in volta la sua posizione aggiornata.

Di conseguenza, quando un giocatore effettua un passaggio oppure un tiro, la palla deve essere spostata da un'entita' che pero' non puo' essere il giocatore stesso: in altre parole, si tratta di simulare l'impressione di un moto alla palla a seguito di un'azione del giocatore che la controlla. Questa mansione e' ricoperta dal cosiddetto *agente di movimento*. L'agente di movimento e' un'entita' reattiva concorrente agli altri giocatori, il cui unico compito e' quello di spostare la palla in una determinata direzione fino a che la potenza ad essa impressa e' sufficiente a farla avanzare alla cella successiva. Una volta completato il suo compito, smette di eseguire in attesa del prossimo spostamento da effettuare. In alcuni casi, l'agente di movimento viene volutamente bloccato attraverso l'arbitro, ad esempio nel caso in cui la palla esca dal campo e sia necessario assegnare una rimessa oppure un calcio d'angolo.

2.3.6 Modifiche sulle squadre: gli allenatori

Nel gioco del calcio i giocatori di ciascuna squadra vengono coordinati dal proprio allenatore, che ha il compito di scegliere come disporre i giocatori in campo (la formazione) e di effettuare delle sostituzioni, come conseguenza di un infortunio o piu' semplicemente per una scelta tattica. In questa simulazione i due allenatori rappresentano due componenti distribuite separate,

che comunicano con l'unita' centrale di controllo. Ciascun allenatore ha la facolta' di prendere le seguenti decisioni:

- effettuare un cambio di formazione per la propria squadra
- effettuare una sostituzione di un giocatore con un suo compagno presente in panchina

Le decisioni prese da un allenatore hanno come precondizione necessaria il gioco fermo: sara' pertanto l'arbitro a dover esaminare e successivamente accettare le richieste di un allenatore solo quando le condizioni lo permettono. Nel caso di un semplice cambio di formazione, la decisione dell'allenatore viene applicata sulla squadra, cosicche' i giocatori al proprio turno successivo sappiano che la loro posizione di riferimento e' cambiata. Diverso e piu' complesso e' invece il caso della sostituzione. La sequenza di operazioni che seguono una richiesta di sostituzione si possono schematizzare come segue:

1. L'arbitro riceve la richiesta di sostituzione e, non appena il gioco e' fermo, procede a notificarlo ai giocatori
2. Ciascun giocatore, prima di decidere la propria mossa, controlla se l'allenatore ha deciso di sostituirlo con un compagno
3. Nel caso del giocatore interessato alla sostituzione, esso si dirige verso la panchina (eventualmente lasciando la palla dove si trova)
4. Una volta giunto in panchina, il suo compagno prende il suo posto in campo e si dirige verso la propria posizione di riferimento
5. L'arbitro, ad ogni turno, controlla se il giocatore entrante ha raggiunto la posizione di riferimento e, in quel caso, sancisce la ripresa del gioco

Il comportamento sopra descritto vale anche nel caso di piu' sostituzioni simultanee, che vedono i giocatori uscire ed entrare nel campo allo stesso tempo.

Ciascun allenatore ha inoltre una visione meno precisa del gioco e riceve pertanto un sottoinsieme degli eventi che caratterizzano una partita: dal punto di vista decisionale e' infatti poco interessante per un allenatore conoscere ogni singolo movimento di ogni giocatore. Si ha quindi che l'allenatore viene notificato solo in caso di eventi "salienti", che come descritto nel Capitolo 3, sono stati denominati *Game Events*.

3 Analisi architetturale

In questo capitolo verranno presentate le decisioni architetturali inerenti la struttura e l'organizzazione del software di simulazione. L'analisi comincia con una visione generale della gerarchia di eventi, per poi suddividersi nelle scelte che riguardano la parte di concorrenza e le scelte che riguardano la parte di distribuzione.

3.1 Gli eventi

Gli eventi sono l'elemento costituente di una partita. Essi possono essere generati dalle diverse entita' e possono essere raggruppati in diverse categorie. Inoltre, alcuni di questi eventi sono interessano la sola parte concorrente, mentre altri eventi possono viaggiare dalla parte concorrente a quella distribuita e viceversa.

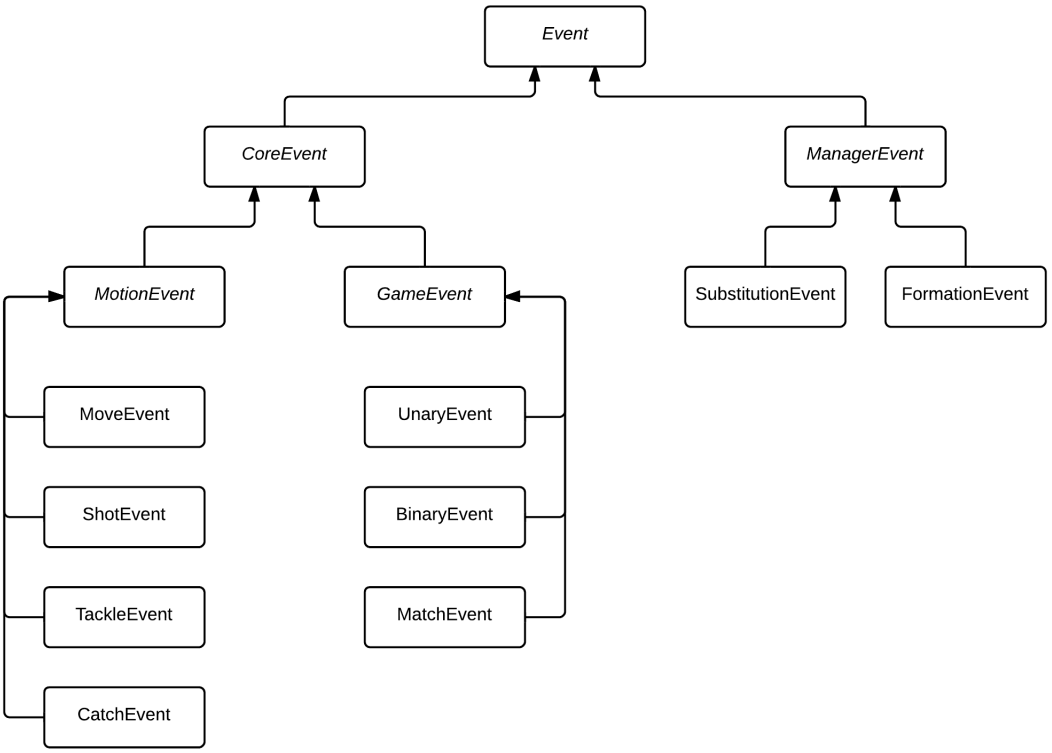


Figure 1: La gerarchia degli eventi che costituiscono una partita.

La struttura globale degli eventi e' schematizzata in Figura 1. Data la sua complessita' ed estensione, verranno ora trattati singolarmente i diversi

tipi di eventi al fine di spiegarne il loro significato e il contesto in cui vengono utilizzati.

Event *Event* rappresenta l'evento generico ed e' alla base della gerarchia. Da esso si diramano due macro-categorie di eventi: i *CoreEvent* e i *ManagerEvent*. Esso sono rispettivamente gli eventi che vengono generati nella parte concorrente (il cosiddetto "Core") e gli eventi che vengono generati nella parte distribuita (fondamentalmente, dagli allenatori). *Event* puo' essere considerata come un'entita' astratta, che non viene concretizzata se non da uno dei suoi derivati.

CoreEvent Gli eventi di tipo *CoreEvent* sono un insieme di eventi che vengono generati dalla parte concorrente del sistema, ovvero il Core. La loro generazione tuttavia non vincola il loro utilizzo nella sola parte concorrente: essi vengono infatti inviati alla parte distribuita per notificare gli allenatori (e l'interfaccia grafica del campo) sullo svolgersi della partita. Vi sono due tipologie di *CoreEvent*: i *MotionEvent*, che rappresentano le possibili azioni dei giocatori, e i *GameEvent*, che invece rappresentano tutti quelli eventi che influiscono sullo stato di gioco. Anche in questo caso, i *CoreEvent* sono astratti e trovano una concretizzazione nei loro discendenti.

MotionEvent Tutte le azioni che un giocatore puo' compiere sono definite dai *MotionEvent*. Sono stati definiti quattro tipi di *MotionEvent*, elencati di seguito.

- *MoveEvent* - Descrivono i movimenti di un giocatore, dal punto in cui si trova al punto in cui si vuole spostare. Questi eventi vengono altresi' usati per descrivere gli spostamenti della palla.
- *ShotEvent* - Rappresentano il tiro/passaggio effettuato da un giocatore, e sono caratterizzati da alcune informazioni quali la posizione del giocatore e la potenza impressa alla palla.
- *TackleEvent* - Corrisponde al tentativo di contrasto verso un altro giocatore.
- *CatchEvent* - Questo evento descrive il gesto di prendere possesso di una palla, sia essa inerte sul campo (non in possesso) oppure come intercettazione di una palla in movimento.

Ciascuno di questi eventi viene sottoposto all'attenzione dell'arbitro, che ne valida la correttezza nel rispetto delle regole del gioco. Inoltre, come gia' anticipato, questi eventi vengono anche inviati alla parte distribuita, cosi' da poter aggiornare la visualizzazione della partita e per permettere agli allenatori di prendere decisioni tattiche.

GameEvent Gli eventi che regolano lo svolgimento del gioco rientrano nella categoria di *GameEvent*. Essendo una tipologia di evento molto vasta, esso si suddivide ulteriormente in tre specializzazioni: gli *UnaryEvent*, i *BinaryEvent* ed i *MatchEvent*. Le prime due tipologie si riferiscono al fatto che l'evento coinvolge solamente un giocatore (e.g., una rimessa laterale: da qui, unario) oppure due giocatori (e.g. un fallo), mentre la terza tipologia raccoglie tutti gli eventi come l'inizio della partita, la fine del primo tempo, e cosi' via.

UnaryEvent Tutti gli eventi di gioco che sono raggruppati in questa categoria vanno a coinvolgere un solo giocatore per la ripresa del gioco. Essi rappresentano le seguenti situazioni:

- Rimessa laterale
- Rimessa dal fondo
- Calcio d'angolo
- Calcio di punizione
- Calcio di rigore

Ciascuno di questi eventi prevede che il gioco possa essere sbloccato da un particolare giocatore della squadra interessata e che la palla sia posizionata nella posizione stabilita dall'arbitro. Il gioco puo' riprendere solo se i giocatori sono nella loro posizione di riferimento per quella particolare situazione, prima che la palla venga rimessa in gioco. Inoltre, questi eventi sono generati esclusivamente dall'arbitro, che per ogni azione effettuata controlla lo stato di gioco globale alla ricerca di irregolarita'.

BinaryEvent Questa particolare categoria vanta un solo evento possibile: il fallo. Questo evento e' infatti l'unico a coinvolgere simultaneamente due giocatori di squadre opposte, e si origina quando il contrasto avviene in maniera irregolare (infortuna il giocatore, oppure compie gesti pericolosi). In questo caso l'arbitro, dopo aver segnalato del fallo, decide le modalita' di ripresa del gioco, ad esempio un calcio di punizione, e genera il rispettivo evento.

MatchEvent Gli eventi che regolano l'inizio e la fine di ciascun tempo di gioco fanno parte dei *MatchEvent*. Di conseguenza, vi sono quattro possibili varianti: inizio del primo tempo, fine del primo tempo, inizio del secondo tempo, fine del secondo tempo. Questi eventi sono usati sia dai giocatori per capire quale azione dovranno compiere (e.g., ingresso in campo), ma

vengono anche inviati alle componenti distribuite per notificare lo svolgersi della partita.

ManagerEvent Questa classe astratta di eventi, al contrario dei *CoreEvent* che sono generati dalla componente *Core*, vengono generati dalle componenti distribuite, in particolar modo dai due *Manager*. La loro funzione e' quella di notificare all'arbitro una richiesta di sostituzione per un giocatore oppure di cambio di formazione. L'arbitro, una volta ricevuto l'evento, provvedera' ad elaborarlo alla prima occasione utile di gioco fermo.

Un *SubstitutionEvent* contiene informazioni quali la squadra interessata, il numero di maglia del giocatore uscente e il numero di maglia del giocatore entrante; un *FormationEvent* invece, oltre alla squadra interessata, fornisce anche una sigla identificativa del nuovo assetto che la squadra dovra' assumere.

3.2 Concorrenza

Questa sezione propone inizialmente una panoramica sulla soluzione scelta per garantire l'assenza di deadlock e starvation in fase di design. Successivamente viene fatta luce su casi particolari che il modello finora descritto non gestisce, mettendo in evidenza la necessita' di arricchirlo con meccanismi e componenti dedicate.

La scelta del linguaggio per lo sviluppo della parte principale del software e' ricaduta su Ada, il quale ha un modello di concorrenza che mette a disposizione una semantica molto ricca. Un aspetto molto interessante riguarda i canali di comunicazione per le richieste: questi permettono accessi potenzialmente paralleli in fase di lettura, mentre garantiscono mutua esclusione e, in alcuni casi, anche accodamento condizionale in scrittura. Quest'ultimo costrutto è molto utile quando si vuole soddisfare una richiesta solamente al verificarsi di alcune condizioni, come per esempio una risorsa che si libera o una variazione dello stato di gioco.

3.2.1 Deadlock, starvation e correttezza

In un sistema concorrente è necessario tenere conto di problematiche come la starvation ed il deadlock, specialmente quando si è in presenza di entità che condividono risorse. Nel caso di questo progetto, i giocatori devono avere accesso ad uno stato condiviso che gli permetta di prendere decisioni, le quali andranno poi a modificare lo stato stesso.

L'analisi del problema ha evidenziato la necessità di avere un unico luogo in cui mantenere lo stato. In fase di modellazione, tale problematica è stata risolta ponendo un'unica entità che garantisce mutua esclusione al controllore dei dati condivisi.

Questa decisione rende più semplice garantire l'assenza di deadlock e di starvation. Nel secondo caso, ogni job ha la garanzia di accedere alla risorsa in un tempo finito, in quanto le richieste vengono processate con ordinamento FIFO basato sul tempo di arrivo. Per quanto riguarda invece il primo caso, e' presente solamente uno dei quattro requisiti che stanno alla base di questa pericolosa condizione. I giocatori eseguono la propria sequenza di operazioni (che caratterizza un turno) dividendole tra "online" ed "offline": questo vuol dire che solamente le fasi di lettura e di scrittura necessitano dell'utilizzo della risorsa, mentre la parte di intelligenza artificiale viene effettuata in modo autonomo sui dati recuperati; inoltre, solamente la parte di scrittura richiede mutua esclusione. Il pre-rilascio non è inibito ed il suo comportamento non è controllato dal software, ma dal sistema operativo operativo sottostante: un job pre-rilasciato che sta eseguendo la sezione critica di una richiesta in mutua esclusione deterrà ancora, al suo risveglio, l privilegio sulla risorsa, mentre gli altri job che la richiedono saranno in attesa sul rispettivo canale esposto.

Un caso più particolare si ha invece per quanto riguarda attesa circolare ed accumulo di risorse, per cui entra in gioco un meccanismo di riaccodamento e rivalutazione delle richieste che verra' discusso in seguito. Per ora, basti sapere il concetto alla base di tale meccanismo: una richiesta che non viene soddisfatta a causa di una risorsa occupata può fallire o venire riaccodata, cosi' da poter effettuare un nuovo tentativo di esecuzione in un secondo momento. In questo caso, dopo un certo numero di tentativi, l'operazione viene rivalutata e modificata in una simile, che vada comunque a soddisfare le esigenze del giocatore.

3.2.2 Palla

La palla consiste di una risorsa protetta che ne detiene la posizione e ne permette l'accesso in mutua esclusione. Si supponga di tenere tale risorsa all'interno del controllore, permettendo solo ad esso di accedervi: in questo caso, sia i giocatori che l'agente di movimento dovrebbero concorrere per andare a modificare lo stato e, di conseguenza, anche la posizione della palla: il moto del pallone sarebbe quindi posto allo stesso livello delle decisioni e degli spostamenti dei giocatori. In queste circostanze verrebbe meno un fattore di realismo molto importante, tale per cui il pallone è slegato dalla

velocità di gioco dei giocatori in campo. Esso deve invece essere libero di muoversi a velocità molto più alte rispetto ai giocatori e deve poter continuare il proprio moto anche con gioco fermo: in quest'ultimo caso sarà l'arbitro a riposizionarla, ad esempio, in un punto opportuno a seconda dell'irregolarita'. Inoltre, accodando una richiesta di movimento della palla insieme ad altre richieste dei giocatori, si rendono le operazioni strettamente sequenziali, eliminando un livello di indeterminismo desiderabile che e' proprio del parallelismo e che lo avvicina al gioco reale.

Un chiaro esempio di questo aspetto si trova nel tentativo di un giocatore di prendere la pallla in movimento. E' ragionevole pensare che la sua azione possa fallire perche' il tiro è troppo forte. Se invece si pongono la palla e lo stato in due luoghi differenti, diventa possibile uno scenario in cui il giocatore tenta di prendere la palla nell'ultima posizione a lui nota ma quest'ultima nel frattempo si e' spostata.

Sulla base delle considerazioni fatte finora, all'interno dell'architettura e' stato deciso di tenere le informazioni riguardanti la palla slegate rispetto al controllore. In questo modo si permette sia ad un giocatore che all'agente di movimento di contendere la palla in modo potenzialmente parallelo, demandando alla risorsa che identifica la palla gestirne gli accessi in mutua esclusione e sotto determinate circostanze.

La struttura della palla e' stata quindi studiata in modo tale da permettere ad un entita' alla volta di poterla controllare, sia essa uno dei giocatori (attraverso il controllore) oppure l'agente di movimento. Quest'ultimo si attivera' quando risvegliato da un giocatore e sara' fermato quando un altro giocatore conquista il pallone o se il moto che gli era stato impresso si e' esaurito. In particolare, l'attivazione e disattivazione del task sfrutta la potenza dei canali che permettono accodamento con condizione.

3.2.3 Giocatori

I giocatori, dopo una prima fase di inizializzazione, eseguono ad ogni turno un'operazione che va a modificare sia lo stato di gioco che il proprio. Se ogni giocatore avesse la propria parte di stato, la sua condivisione con le altre entita' in gioco sarebbe piu' complicata di quanto necessario; inoltre, la necessità di un unico stato centrale rende tali informazioni ridondanti oltre che difficilmente consistenti durante l'esecuzione della simulazione.

Task stateless

Utilizzando un approccio in cui i giocatori sono stateless si permette una migliore gestione dei dati condivisi, che vengono mantenuti in un unico luogo all'interno del sistema. Ad ognuno dei 22 task in campo viene assegnato un identificativo con il quale, ad ogni turno, il giocatore ottiene le informazioni che lo riguardano e di cui ha bisogno. Tale identificativo viene ottenuto in fase di inizializzazione e puo' cambiare solamente in caso di sostituzione tra un giocatore in campo ed uno in panchina; cosi' facendo non vi e' necessita' di creare o risvegliare un secondo task.

Area di azione

Ogni giocatore e' caratterizzato da delle statistiche impostate in fase di inizializzazione che lo differenziano dagli altri, ed ha come scopo quello di rendere lo svolgimento del gioco piu' realistico. Tali statistiche comprendono potenza, velocita', contrasto, difesa, attacco, precisione e bravura in porta. Abbianate allo stato di gioco dello stesso giocatore, rendono possibile effettuare alcune assunzioni che influiscono nello svolgimento del resto del turno.

Se un giocatore possiede la palla e tra le proprie statistiche ha un'alta precisione e potenza, esso potra' effettuare un lancio lungo, in caso contrario potra' optare un passaggio corto ad un compagno vicino. Viene definita in questo modo un'area di interesse del giocatore, entro la quale esso potra' agire a prescindere dall'azione che scegliera' di fare. Un giocatore che chiede al controllore lo stato di gioco della partita non necessita di avere una visione globale dell'intero campo, bensi' solamente di una sua sotto parte, definita dalla sua posizione ed un raggio determinato in base alle caratteristiche del giocatore ed alle informazioni preliminari che ha ottenuto riguardo allo stato globale.

Il turno

La parte iniziale di ogni turno del giocatore risulta' cosi' suddivisa:

1. richiesta al controllore del proprio stato di gioco
2. dove si trova la palla
3. richiedere al controllore lo stato della propria area di interesse.

Queste operazioni, in quanto letture, avvengono effettuate potenzialmente in modo parallelo rispetto agli altri task.

La seconda fase consiste nella parte di intelligenza artificiale nella quale viene decisa la . Una volta decisa la mossa da effettuare, viene creato il

relativo evento a cui abbina un certo livello di priorita', cioe' quanto e' importante per il giocatore l'azione appena decisa. L'azione composta da evento e priorita' vengono sottoposte al controllore tramite una chiamata in mutua esclusione con accodamento. Tale canale viene sfruttato dal sistema per fermare il gioco e farlo riprendere, per esempio quando l'utente mette in pausa il gioco.

Antecedente al ciclo di turni che caratterizzano la normale esecuzione di un giocatore e' presente una fase di inizializzazione nella quale il giocatore attende che tutti i giocatori siano attivi, recupera il proprio id ed attende l'inizio della partita.

Ci sono altri casi in cui vi e' la necessita' di fermare per poi far ripartire i giocatori, in questi casi viene fatto uso di una risorsa protetta che mette a disposizione dei canali per accodare i giocatori per poi sboccarli al verificarsi di determinate condizioni.

Entrata in campo

L'entrata in campo come l'uscita e' un altro degli aspetti critici della concorrenza del progetto. I giocatori sono creati con posizione di partenza sulla panchina della propria squadra, che consistono in una serie di celle esterne al campo, ed all'avvio della partita si spostano verso la propria posizione. In una partita reale i giocatori entrano ed escono dal terreno di gioco dal centro del lato lungo, per avere questo comportamento e' stata inserita una cella di campo esattamente adiacente al punto desiderata attraverso la quale i giocatori passano dalla panchina al campo e viceversa.

Sostituzione

La cella utilizzata nel meccanismo di entrata ed uscita dal campo viene sfruttata per effettuare le sostituzioni, un giocatore che deve essere sostituito dopo essersi accorto tramite lettura dello statao esce fino a raggiungere tale cella, in essa il suo id viene cambiato con quello del nuovo giocatore. Dato che i giocatori non hanno stato, se non l'id che li identifica e con il quale recuperano ad ogni turno le proprie informazioni dal cotroller, modificando il suo valore e' equivalente ad averlo sostituito. In questo modo vengono creati solamente 22 giocatori in ogni partita, riutilizzandoli per simulare quelli in panchina.

3.2.4 Stato, controllor ed arbitro onniscente

Lo stato consiste in una serie di dati che descrivono ogni giocatore in campo, quindi un identificativo, il numero di maglia, la posizione attuale e quella di riferimento in campo (cioe' la posizione dettata dalla formazione), oltre ad informazioni generali sullo stato nel suo complesso. Lo scopo delle azioni da parte dei giocatori e' quella di andare a modificare la propria posizione all'interno del campo ed effettuare altre operazioni ai fini del gioco.

Il controllore gestisce gioco tramite la sua componente di arbitro influenzando l'andamento del gioco, quando necessario blocca la ricezione di richieste scrittura di eventi.

Arbitro

Ogni richiesta viene presa in cosiderazione dalla componente che rappresenta l'arbitro, il quale valuta la correttezza o meno dell'azione. In caso di fallo l'arbitro ferma il gioco dal punto di vista logico e sancisce la squadra per il prossimo possesso palla. A questo punto prima che il gioco possa riprendere l'arbitro attende che le squadre si riposizionino correttamente per garantire le giuste distanze dal punto di battuta, poi il gioco riprende al momento di battuta da parte del giocatore designato.

Processare eventi

Nel sistema esistono diversi tipi di eventi e di azioni che un giocatore puo' efffettuare, dal punto di vista del controllore vengono gestite in modo differente: una volta accettata la richiesta ne determina il tipo e cerca di soddisfare il giocatore permettendogli di aggiornare lo stato con l'azione richiesta. Il fatto che il giocatore divida il suo turno in fasi, fa in modo che i presupposti secondi cui ha scelto una determinata mossa al momento non vi siano piu'.

Per esempio, un giocatore tenta di prendere la palla che e' in una certe posizione, ma tra quando se ne e' accorto e quando tenta effettivamente di prenderne il possesso, la palla si e' spostata. Questo, come anche in altri casi in cui ha senso applicare questo tipo di comportamento, e' dettata dal tentativo di creare una simulazione realistica, se un giocatore non e' abbastanza veloce o si accorge tardi di una certa circostanza e' giusto che veda la propria azione fallire.

Se l'operazione fallisce, cioe' non puo' essere applicata allo stato, il

controllore si comporta in modo differente in base al tipo di operazione richiesta:

- operazioni come il passaggio, il tiro, il tentativo di prendere la palla ad un avversario o mentre non e' controllata da nessun giocatore se non vanno a buon fine falliscono, il giocatore al turno successivo si accorge che lo stato non e' stato modificato come aveva richiesto
- le operazioni di movimento possono fallire, la cella di destinazione potrebbe essere stata occupata, sotto certe condizioni tale mossa puo' essere ritentata in seguito

L'utilita' che un giocatore assegna alla propria azione determina il comportamento nel secondo caso, la richiesta quando accettata dal controllore ha una certa utilita', se non va a buon fine questo valore viene decrementato e la mossa messa in attesa di una nuova occasione di esecuzione. Ad ogni tentativo il valore decresce fino a diventare minore di una certa soglia fissata priori, sotto la quale l'azione richiesta viene rivalutata dal controllore. Questo consiste nel soddisfare la richiesta con un'operazione il piu' simile possibile a quella decisa dal giocatore.

Tale meccanismo ha come scopo quello di riflettere cio' che succede nelle dinamiche di una partita reale, se un giocatore vuole raggiungere una certa posizione, ma nel percorso che vuole fare si trova un altro giocatore, per un po' temporeggiera' nell'attesa di un suo spostamento, poi cambiera' leggermente la propria corsa per evitare l'ostacolo.

Nella descrizione del meccanismo manca un particolare, cioe' quando un azione fallita viene ritentata. Un azione di movimento per poter essere di nuovo sottoposta al controllore necessita che il giocatore che la impedisce si sia spostato, ma tale approccio potrebbe risultare troppo oneroso dal punto di vista implementativo e poi a tempo di esecuzione. La soluzione scelta consiste di permettere la rivalutazione delle mosse fallite ogni qual volta che un giocatore lascia la propria posizione, anche se non e' detto che sia quella desiderata da un giocatore. Per rendere questo meccanismo piu' efficiente e' stato suddiviso il campo di gioco in 6 settori, quindi ogni richiesta che non e' andata a buon fine e deve essere ritentata viene accodata sulla zona in cui si trova il giocatore, in attesa che un altro giocatore di quella zona effettui un movimento. In questo modo non vengono rivalutate tutte le richieste accodate all'interno del controllore, ma solamente quelle che potenzialmente ora sono applicabili.

Questi meccanismi nel loro insieme permettono di non avere dead-lock, cioe' che un giocatore non resto bloccato in attesa di una risorsa che al momento della richiesta non era disponibile.

3.2.5 La velocita'

Nello svolgimento del gioco descritto finora le caratteristiche entrano in gioco quando i giocatori entrano in contatto tra di loro, per esempio contrasto ed attacco quando il difensore tenta di rubare il pallone all'attaccante, la precisione e la potenza quando due compagni vogliono passarsi il pallone o tirare in porta, la parata per evitare che la propria squadra prenda gol, e cosi' via. In tutti questi casi e' il controllore che quando esamina la richiesta di una determinata azione considera i valori dei vari giocatori, decidendo quindi chi vincera' il contrasto, quanto preciso sara' il passaggio, quanto forte applicare al tiro.

Per quanto riguarda la velocita' e' necessario un approccio differente, essa si esprime come la quatita' di mosse che e' in grado di fare in un lasso di tempo, quindi quante richieste puo' sottoporre al controllore in questo frangete. Per semplicita' di esposizione tale quantita' di tempo e' identificata dal simbolo T, mentre con t_0, t_1, t_2,... indichiamo gli istanti a distanza T durante lo svolgimento del gioco.

Ad ogni giocatore e' abbinato un valore tra 1 e 5 per indicare quante mosse sono permesse durante T, quindi il giocatore con piu' mosse a disposizione sara' piu' veloce rispetto ad un numero minore. Questo rispecchia quanti turni deve poter eseguire, diventa quindi cruciale come viene calcolato T, esso deve permettere ad ogni giocatore di eseguire un numero di volte pari al suo valore che rappresenta la velocita'. Dato un campionamento del tempo di esecuzione pessimo di un turno di un giocatore, che chiameremo C, vogliamo che un giocatore lento esegua C una volta all'interno di T, uno un po' piu' veloce due volte, e cosi' via fino a 5. Inoltre vogliamo che all'inizio di ogni T, al tempo t_i, tutti i giocatori siano pronti per eseguire il proprio turno, questo aggiunge al gioco un ulteriore livello di indeterminismo, in quanto non e' possibile sapere quali saranno i giocatori selezionati per eseguire per primi.

Da questo ragionamento si ottiene che T e' pari alla somma di tutti questi tempi, quindi del tempo di tutte le esecuzioni di C da parte di ogni giocatore in campo. In questo modo si garantisce ad ogni giocatore un tempo di esecuzione sufficiente per portare a termine i propri turni, spetta poi ad ognuno di essi distribuirli in modo uniforme nell'arco di T, tramite un

sistema di delay che scandisce il tempo ripartendolo. Un esempio, poniamo T pari a 40 unita' di tempo ed iniziato al tempo t_i, C uguale a 2 e la velocita' pari a 5: il giocatore tra un turno e l'altro tentera' di eseguire il suo turno entro le prime 8, 40 diviso 5, unita' di tempo, per poi attendere fino a $t_i + 8$ per il prossimo turno. Non e' detto che esso riesca ad eseguire per 2 unita' di tempo nel periodo di 8, dato che i giocatori iniziano ad eseguire tutti insieme, ma di sicuro eseguira' 5 volte il tempo C (2) all'interno dell'iperperiodo T (40).

Dato C come stima fatta a priori, in questo meccanismo il controller ha il compito di calcolare T e tenere aggiornato il tempo t_i di riferimento. T e' il pari alla somma di tutti i turni di ogni giocatore, il controllore deve calcolare tale valore ad inizio del gioco, per poi aggiornarlo ogni qual volta ce ne sia il bisogno, per esempio in caso di sostituzione con un giocatore con un altro, questo perche' non e' detto che abbiano la stessa velocita'. Per quanto riguarda t_i, in un a normale esecuzione della partita tale valore viene calcolato partendo da un tempo di riferimento all'avvio del software, al quale viene aggiunto ogni volta T, in caso invece di interruzione del gioco da parte dell'utente tale valore di riferimento deve aggiornato con l'istante in cui il gioco puo' riprendere.

[TODO esempio di iperperiodo grafico]

3.3 Distribuzione

Dal momento che la componente *Core* non dispone di una propria interfaccia grafica che possa garantire la fruizione e l'interazione dall'esterno, si pone la necessita' di rendere possibile una comunicazione bidirezionale da e per le componenti distribuite, ovvero *Field* e le due istanze di *Manager*. Di seguito verranno quindi analizzate le soluzioni adottate per questo particolare aspetto del software, la cui rispettiva implementazione verra' successivamente trattata nel Capitolo 4.3.

3.3.1 Il bridge

Il modulo che si occupa di consentire alla componente *Core* di accettare comunicazioni dall'esterno, sia in entrata che in uscita, e' chiamato *bridge*. A causa della duplice natura delle comunicazioni possibile, il bridge e' logicamente diviso in due parti: bridge input e bridge output.

Bridge input Il bridge input viene utilizzato dalle due componenti distribuite *Field* e *Manager* per comunicare con la componente *Core*, occupandosi quindi di gestire tutte le comunicazioni in ingresso. Per quanto riguarda

Field, i metodi a sua disposizione permettono di:

- Iniziare una nuova partita
- Mettere in pausa la partita corrente
- Dare il via al secondo tempo, a seguito di un intervallo
- Terminare la simulazione (spegnimento globale del software)

Per quanto riguarda invece la componente *Manager*, e' possibile:

- Recuperare tutte le informazioni relative ad una squadra
- Recuperera tutte le informazioni relative ai giocatori
- Cambiare la formazione della squadra
- Effettuare una sostituzione tra due giocatori

Appare tuttavia evidente che la separazione logica tra le due tipologie di bridge non rispecchia il flusso dei dati trasmessi: infatti, nel caso del bridge input, esso permette alle componenti distribuite sia di richiedere informazioni (e.g. recuperare le statistiche dei giocatori) che di inviare delle informazioni (e.g. cambiare la formazione).

Bridge output Il bridge output viene invece utilizzato da *Core* per notificare gli avvenimenti della partita, permettendo cosi' a *Field* e *Manager* di disegnare l'interfaccia grafica e di mostrare le relative informazioni. Questo modulo gestisce quindi tutte le comunicazioni in uscita; inoltre, il flusso di dati trasmessi e' solo uscente, a differenza della sua controparte.

Alla base del bridge output e' posto un buffer, il cui compito consiste nel raccogliere tutti gli eventi di gioco che vengono generati durante una partita ed inviarli periodicamente alle componenti distribuite. Il contenuto del buffer viene inviato se si verifica una tra tre particolari condizioni. Banalmente, il fatto che il buffer si riempia completamente causa l'invio di tutti gli eventi e un conseguente svuotamento dello stesso. Se invece il buffer non riceve piu' eventi entro un certo periodo di tempo noto a priori, viene inviato il tutto contenuto del buffer (e viene svuotato). Infine, vi sono eventi piu' importanti di altri che devono essere notificati immediatamente (e.g. una fallo): anche in questo caso, il sopraggiungere di uno di questi eventi scaturisce l'invio di tutti gli eventi e lo svuotamento conseguente del buffer.

Ad ogni modo, la scelta della dimensione del buffer e' molto importante e determina il throughtput degli eventi e la conseguente "fluidita'" della

rappresentazione grafica della partita. Tuttavia, un invio piu' frequente di eventi puo' essere causa di una congestione di rete, quindi e' opportuno trovare un buon bilanciamento tra quantita' di eventi inviati e frequenza di invio. A questo proposito, il bridge output applica una sorta di filtraggio degli eventi nel tentativo di unire piu' eventi in uno unico. Questo avviene molto spesso negli eventi di movimento dei giocatori e della palla, che sono in assoluto i piu' frequenti durante una partita. La tecnica che il bridge output adotta e' quella di unificare delle mosse successive di uno stesso giocatore (o della palla) in un unico evento, che ha come punto di partenza quello del primo evento ricevuto e come punto di destinazione quello dell'ultimo evento ricevuto. In questo modo il numero di eventi e' di gran lunga inferiore e garantisce maggior efficienza senza inficiare sulle performance e sul throughtput.

Utilizzo del bridge Il bridge, poiche' risiede all'interno di *Core*, deve essere necessariamente acceduta ed utilizzata da una delle entita' presenti in quella componente. Inoltre, dal momento che le comunicazioni dall'esterno sono asincrone e non predicibili, che l'accesso avvenga serialmente, sia esso in lettura o in scrittura. Per garantire questa caratteristica l'unica entita' che si occupa di interagire con il bridge e' l'arbitro (ai fini della spiegazione lo si consideri come un sotto-modulo del controllore). Per facilitare la comprensione delle interazioni dell'arbitro con il bridge, si consideri la seguente situazione, dove il controllore ha appena eseguito un'azione proveniente da un giocatore e la sottopone all'attenzione dell'arbitro. Quest'ultimo esegue le seguenti operazioni:

1. Controlla se l'azione puo' causare una situazione di gioco fermo (che, si ricorda, permette ad una sostituzione e ad un cambio di formazione di avvenire)
2. Interroga il bridge input e controlla se ci sono richieste da parte della distribuzione; in caso affermativo, le esegue, se le condizioni lo permettono
3. Valida l'azione del giocatore e aggiunge il rispettivo evento alla coda del buffer di bridge output
4. Se tale azione ne causa un'altra (e.g., un goal), l'evento di quest'ultima viene aggiunto alla coda del buffer di bridge output

L'ordine in cui queste operazioni vengono eseguite rispecchia l'ordine nel quale vengono processate in una vera partita di calcio. Inoltre, essendo l'accesso al bridge riservato al solo arbitro, vengono evitate potenziali inconsistenze sullo stato della partita.

L'unica eccezione a questa regola e' costituita dagli eventi della distribuzione che agiscono sullo stato generale del sistema, ovvero la richiesta di una nuova partita, di mettere in pausa quella corrente oppure di terminare la simulazione ed il sistema. In questo caso e' direttamente il bridge input ad impartire il relativo comando, senza quindi aspettare che sia l'arbitro ad accorgersi della richiesta; tale scelta e' stata dettata dal fatto che questa tipologia di richieste ha la massima priorita' sulle altre, in quanto agisce sul sistema stesso.

3.3.2 Architettura client-server

La comunicazione tra le componenti distribuite *Field* e *Manager* e la componente centrale *Core* puo' essere vista come una comunicazione tipica di un'architettura di tipo client-server. Generalmente, il server si mette in ascolto di eventuali richieste in ingresso: quando il client effettua una richiesta al server, questo la prende in carico, la elabora e restituisce al client la risposta. Tutto questo avviene mantenendo una connessione attiva tra le due parti, che viene terminata non appena il client riceve la risposta del server.

Per poter offrire le funzionalita' descritte in Sezione 3.3.1, *Core* assume il ruolo di server ed ha come client *Field* e le due istanze di *Manager*. All'avvio del software, *Core* si mette a disposizione delle componenti distribuite, rimanendo in ascolto fino al comando di uscita. I client, una volta che *Core* e' avviato, possono aprire una connessione con quest'ultimo, rimanendo in attesa fino a che la loro richiesta non viene soddisfatta ed essi hanno ricevuto la rispettiva risposta.

3.3.3 Architettura publisher-subscriber

L'aspetto che vede *Core* comunicare con le altre componenti e' invece meno comune rispetto alla tipica comunicazione client-server, analizzata nella sezione precedente. Infatti, dal momento che *Core* non offre nessun tipo di interazione diretta con l'utente, e' necessario che le informazioni riguardanti la partita vengano inviate a *Field* (per la visualizzazione e il controllo del software) e ai due *Manager* (per la gestione della squadra).

Nonostante una soluzione possibile sia quella di replicare l'approccio precedente, che vedrebbe quindi ciascuna entita' agire sia da server che da client a seconda delle necessita', non rappresenta un approccio scalabile e porterebbe un notevole aumento della complessita' generale del sistema. La soluzione adottata si basa invece su un'architettura di tipo publisher-subscriber. Il suo funzionamento prevede che vi sia un fornitore di contenuti

(il produttore), solitamente divisi in canali, al quale le entita' interessate si iscrivono (i consumatori): ogni qual volta il produttore dia origine ad un nuovo contenuto, questo viene mandato sul canale corrispondente a tutti coloro che si sono iscritti agli aggiornamenti per quel particolare contenuto. Sara' poi compito dei consumatori quello di processare correttamente il contenuto di volta in volta ricevuto.

Questo particolare approccio risulta particolarmente vantaggioso in quanto mantiene una struttura chiara del sistema e non richiede alcun tipo di meccanismo di interrogazione continua (polling). Inoltre, e' un approccio altamente scalabile e moderatamente trasparente per quel che riguarda il fornitore di contenuti, che ha una lista di iscritti per ciascun canale. E' tuttavia necessario che vi sia una connessione persistente tra produttore e consumatori, pena l'impossibilita' del produttore di contattare direttamente i consumatori all'originarsi di un nuovo contenuto di loro interesse.

4 IA dei giocatori

In questa sezione viene descritta la struttura dell'intelligenza artificiale dei giocatori.

4.1 Programmazione Logica: Prolog

Per creare il sistema decisionale dei giocatori utilizzeremo Prolog, un linguaggio di programmazione che trova le sue radici nella logica del primo ordine. [1]. Essendo un linguaggio di programmazione ispirato alla logica del primo ordine, un programma scritto in Prolog è un insieme di predicati formati da una concatenazione di *precondizioni*, ovvero dei fatti che devono essere tutti veri affinchè la *testa* della clausola sia vera.

Prolog risolve le clausole utilizzando l'inferenza logica in maniera molto efficiente, ma senza nessun controllo su cicli o cammini infiniti. Ciò lo rende molto veloce se gli viene fornito un corretto insieme di clausole, ma incompleto altrimenti. Infatti la politica adottata da Prolog è quella di "scaricare" sul programmatore la responsabilità di scrivere programmi corretti ed efficienti.

4.2 Struttura dell'IA

In questa sezione viene analizzata la struttura dell'intelligenza artificale creata in Prolog.

Il sistema è diviso in due parti principali ciascuna delle quali è composta da una serie di predicati logici scritti in Prolog.

La prima parte è formata da predicati che descrivono quello che il giocatore sa riguardo a ciò che lo circonda, detta anche *base di conoscenza*. Per ottenere tali predicati ciascun giocatore deve procurarsi informazioni riguardanti la situazione in prossimità della sua posizione (quali la sua posizione, quale squadra ha la palla, quali giocatori sono vicini a lui e così via) interrogando lo stato. In seguito, un apposito algoritmo implementato all'interno del giocatore si preoccuperà di convertire tali informazioni in predicati logici scritti secondo i formalismi di Prolog. Così facendo ciascun giocatore creerà una serie di predicati concantenati che formeranno la suddetta base di conoscenza, la quale viene presa in input dalla seconda parte del sistema, al fine di decidere l'azione più adeguata alla situazione corrente.

[1] Durante lo sviluppo del progetto è stato utilizzato SWI-Prolog, un'implementazione open source di Prolog compatibile con ogni piattaforma e che mette a disposizione un ampio set di tools per lo sviluppo. Sito ufficiale: http://www.swi-prolog.org.

La seconda parte del sistema è formata da predicati che rappresentano le azioni che i giocatori possono effettuare. Questa parte è a sua volta suddivisa in tre categorie principali:

- *Actions*: contiene tutte le clausole riguardanti le azioni che un giocatore 'normale' può effettuare. Le categorie di azioni a disposizione di un giocatore sono passaggio (*pass*), tiro (*shot*), movimento (*move*), contrasto (*tackle*) e 'prendi la palla' (*catch*);
- *Keeper*: comprende le azioni che il giocatore può fare se assume il ruolo di portiere. Il motivo dell'introduzione di questa distinzione tra giocatore 'normale' e portiere è che il primo non solo ha a disposizione più azioni possibili rispetto al secondo, ma ha anche un comportamento differente. Ciò non dovrebbe sorprendere visto che durante la partita il portiere sta la maggior parte del tempo fermo nella sua porta, mentre un giocatore normale si sposta nel campo ed interagisce con altri giocatori molto più spesso. Inoltre alcuni eventi di gioco non interessano minimamente il portiere, come ad esempio una rimessa laterale o un semplice calcio di punizione eseguito vicino alla metà campo, mentre il giocatore normale potrebbe dover spostarsi in una posizione particolare a causa di essi. Le azioni contenute in questo file sono quindi in numero minore e, in alcuni casi, eseguite diversamente. Per tutti questi motivi è stato ritenuto opportuno utilizzare un set di azioni 'personalizzato' solo per il portiere, nel qual caso durante l'esecuzione del programma i predicati in 'Keeper' verranno utilizzati al posto dei predicati in 'Actions';
- *Utilities*: contiene clausole ausiliarie utilizzate dai predicati in 'Actions' e 'Keeper'. Alcuni esempi di clausole ausiliarie sono l'aggiunta di un elemento ad una lista, il calcolo della distanza tra due punti e il calcolo della corretta coppia di coordinate in cui spostarsi. Queste clausole sono state inserite in un modulo distinto per facilitare la comprensione del programma finale.

Le azioni che possono essere scelte dal giocatore in un dato momento sono determinate dallo stato della partita, dalla presenza di eventi specifici (rimessa, punizione e così via) e dalla situazione specifica nelle vicinanze del giocatore. Più precisamente, l'azione da compiere in un dato istante è il risultato di un'inferenza logica effettuata sulle clausole incluse nella base di conoscenza del giocatore, ovvero sull'input ricevuto dalla prima parte del sistema.

4.3 Workflow

Il processo decisionale che porta il giocatore ad eseguire un'azione è quindi il seguente:

1. Il giocatore interroga lo stato per ottenere informazioni riguardanti le sue immediate vicinanze e lo stato complessivo della partita;
2. Un algoritmo dedicato interno al giocatore provvede a convertire le informazioni ottenute dallo stato in clausole logiche scritte secondo i formalismi di Prolog;
3. Viene lanciato il programma Prolog descritto in Sezione 4.2 fornendo come input la concatenazione di clausole così ottenuta al passo precedente;
4. Prolog esegue un inferenza logica sull'input ricevuto al passo precedente e fornisce in ouput la mossa ottimale per il giocatore data la situazione corrente;

Implementazione

Paragrafo introduttivo.

Concorrenza

Concorrenza.

Distribuzione

In questa sezione viene discussa l'implementazione delle componenti distribuite del software, ovvero *Field* e *Manager*.

La comunicazione tra la componente *Core* e le componenti *Field* e *Manager* e viceversa avviene facendo agire *Core* come un web server. Ciò viene fatto grazie all'utilizzo di AWS (Ada Web Server). All'avvio del sistema, viene inizializzato un web server all'indirizzo *localhost* alla porta 28000 a cui *Field* e *Manager* si connettono e con il quale successivamente scambiano informazioni.

La comunicazione tra le componenti distribuite *Field* e *Manager* e la componente centrale *Core* è stata implementata con un modello di comunicazione tipico di un'architettura client-server. Ad esempio, ciascuna istanza di *Manager*, una volta avviata, invia una richiesta HTTP GET a *Core* richiedendo le statistiche dei giocatori della squadra. La componente *Field* effettua invece chiamate HTTP GET verso *Core* per richiedere azioni quali l'avvio di una nuova partita, l'avvio del secondo tempo della partita corrente, la messa in pausa della partita corrente, ma anche la terminazione forzata della partita.

Per permettere tale tipo di comunicazione a partire dalle componenti distribuite scritte in Java è stata utilizzata la libreria open source Apache HttpComponents, la quale fornisce una completa implementazione del protocollo HTTP in Java e consente di avere accesso a maggiori funzionalità e flessibilità rispetto al package standard *java.net* di Java.

In generale, le richieste provenienti da *Field* e *Manager* sono inizialmente ricevute dal modulo *Soccer.Server.Callbacks* che si occupa di verificare di quale tipo di richiesta si tratta (i metodi a disposizione di delle componenti distribuite sono consultabili in sezione 3.3.1) per poi inoltrarla correttamente verso la componente *Core* tramite il modulo bridge input. Nel caso in cui il client richiedente attenda una risposta da *Core* (e.g. il client ha chiesto le statistiche dei giocatori) il modulo *Soccer.Server.Callbacks* si occuperà di fornirla al client.

La comunicazione di *Core* con le componenti distribuite *Field* e *Manager* è invece vista come un modello di comunicazione di tipo publisher-subscriber.

Tale server mette infatti a disposizione i seguenti canali di comunicazione (ovvero apre dei websocket) ai quali le componenti interessate si iscrivono per ricevere informazioni:

- */managerVisitors/registerForStatistics* è il websocket su cui una delle istanze di *Manager* rimane in ascolto, in particolare l'istanza che rappresenta la squadra che gioca "fuori casa"
- */managerHome/registerForStatistics* è il websocket su cui una delle istanze di *Manager* rimane in ascolto, in particolare l'istanza che rappresenta la squadra che gioca "in casa"
- */field/registerForEvents* è il websocket a cui si connette *Field*

I websocket vengono utilizzati da *Core* per mandare eventi ed informazioni alle altre componenti. Le istanze di *Manager* riceveranno informazioni riguardanti le statistiche dei giocatori e la formazione della squadra, mentre *Field* riceve tutti gli eventi correlati con la partita in corso.

Tramite l'utilizzo dei websocket viene mantenuto attivo un canale di comunicazione tra *Core* e le componenti distribuite, permettendo così la fruizione di contenuti verso *Field* e *Manager*. Il principale vantaggio nell'utilizzo dei websocket è sta nel fatto che l'invio di nuovi contenuti disponibili da parte dell'entit à produttore verso le entit à consumatori avviene senza alcuna richiesta o sollecitazione da parte dei consumatori: non appena è disponibile un nuovo contenuto, il produttore lo invia autonomamente ai consumatori, i quali sono in ascolto sui corrispondenti websocket aperti.

Ada Web Server mette a disposizione una serie di API per facilitare l'uso dei websocket. Per aprire un nuovo websocket è sufficiente utilizzare la funzione *Register* passando come parametro l'indirizzo su cui renderlo disponibile. Per inviare dei dati vi è la funzione *Send*, alla quale è possibile specificare il websocket sul quale spedire i dati e anche il tipo di dati spedito.

Come spiegato in sezione 3.3.1, non vi è un continuo stream di informazioni verso le componenti distribuite, ma viene utilizzato un buffer all'interno di bridge output il cui scopo è bilanciare l'invio di dati: se da un lato un throughtput troppo basso comprometterebbe la rappresentazione grafica della partita, dall'altro un invio troppo frequente di aggiornamenti potrebbe causare una congestione di rete.

4.3.1 Codifica delle Informazioni

Tutti i messaggi scambiati secondo i modelli appena descritti sono codificati utilizzando il formato JSON, il quale è un formato di testo completamente indipendente dal linguaggio di programmazione, ma utilizza convenzioni

conosciute dai programmatori di linguaggi della famiglia del C, come C, C++, C#, Java, JavaScript, Perl, Python, e molti altri. Questa caratteristica fa di JSON un linguaggio ideale per lo scambio di dati.

La componente *Core* processa i dati ricevuti in formato JSON utilizzando*GNATColl*, o GNAT Component Collection, una libreria che mette a disposizione degli ADA package general purpose aggiuntivi. Tra di essi vi è il package *GNATColl.JSON*, il quale permette sia la creazione di oggetti JSON che il parsing di tale tipo di dati ricevuti da *Core*.

Per gestire le informazioni in formato JSON le componenti distribuite, *Manager* e *Field*, utilizzano *Gson*, una libreria open source inizialmente sviluppata da Google. Questa libreria è stata scelta per la semplicità d'uso e la versatilità delle API messe a disposizione. La conversione di un oggetto Java nella sua rappresentazione in JSON è effettuata utilizzando il methodo *toJson()* messo a disposizione dalla libreria. Similmente, la conversiona di una semplice stringa JSON nel corrispondente oggetto Java è possibile utilizzando il metodo *fromJson()*.

4.3.2 Interfacce Grafiche

Le GUI delle componenti *Field* e *Manager* sono state realizzate utilizzando il linguaggio Java. Questa scelta è stata guidata dal fatto che Java è un linguaggio che garantisce un certo livello di portabilità del sistema. In particolare, la grafica ed il layout sono stati implementati utilizzando il framework Swing di Java.

Compilazione ed esecuzione

Per la compilazione e l'esecuzione del progetto sono necessari i seguenti software:

- GNAT_GPL, ambiente di sviluppo;
- AWS, web server scritto in ADA;
- XMLAda, librerie per il parsing XML;
- GNATColl, una libreria che mette a disposizione degli ADA package aggiuntivi;
- Java utilizzato nello sviluppo delle interfacce grafiche;
- Prolog, in particolare SWI-Prolog, utilizzato nello sviluppo dell'intelligenza artificiale dei giocatori;
- Ada Util, un insieme di packages per Ada che forniscono funzionalità aggiuntive.

Tutti questi software sono gratuiti e liberamente scaricabili. In particolare GNAT_GPL, AWS, XMLAda e GNATColl sono sviluppati e supportati da Ada-Core e sono disponibili al sito http://libre.adacore.com sotto licenza GPL. Bisogna inoltre fare attenzione ad avere una versione di Java non inferiore a Java 6. Come detto in precedenza l'implementazione di Prolog utilizzata è SWI_prolog, scaricabile gratuitamente dal sito http://www.swi-prolog.org/ in licenza LGPL. Infine Ada Util è reperibile al sito http://code.google.com/p/ada-util/ in licenza Apache 2.0.

Il progetto è stato testato nelle distribuzioni Linux Ubuntu e Mint e in Mac OS X.
Di seguito vengono proposti i passaggi da eseguire per avere un ambiente di sviluppo pronto, in grado di compilare il progetto. La distribuzione di riferimento è Mint desktop 15 i386. Per ulteriori dettagli si può far riferimento ai file README o INSTALL presenti negli archivi dei file scaricati.

- Scompattare [GNAT FILE]

  installare l'ambiente con il comando ./doinstall
  esportare le variabili d'ambiente in questo modo
  export PATH=/usr/gnat/bin:$PATH
  export GPR_PROJECT_PATH=/usr/gnat/lib/gnat
  export ADA_PROJECT_PATH=/usr/gnat/lib/gnat

- Scompattare [XML ADA FILE] e lanciare i comandi

  ./configure –prefix=/usr/gnat

make all

make install (eseguire da root)

- Scompattare [AWS FILE] e lanciare i seguenti comandi

  make setup

  make build

  make install (eseguire da root)

- Scompattare [GNATCOLL FILE] e lanciare i seguenti comandi

  ./configure

  make

  make prefix=/usr/gnat install (eseguire da root)

AUNIT

ADA UTIL

- L'installazione di SWI-Prolog è abbastanza semplice, in quanto gli sviluppatori mettono a disposizione un repository comodamente aggiungibile al package manager di Ubuntu e di tutte le distribuzione basate su Ubuntu

  sudo apt-add-repository ppa:swi-prolog/stable

  sudo apt-get update

  sudo apt-get install swi-prolog

A questo punto l'ambiente di sviluppo è pronto. Per compilare ed eseguire il progetto bisogna aprire un terminale, spostarsi nella cartella del progetto ed eseguire il comando

```
sh run\_all.sh
```

Con tale comando viene lanciato lo script che si occupa di compilare tutto il progetto e di avviarlo. Al termine della compilazione compariranno le GUI di configurazione della squadra dalle quali sarà poi possibile iniziare la partita.

Conclusioni

Il progetto ha avuto come obiettivo quello di creare una simulazione di una partita di calcio con componenti di concorrenza e di distribuzione.

Per quanto concerne la concorrenza abbiamo appreso come sia importante creare in fase di desigh un'architettura che sia a prova di deadlock e di starvation, e che allo stesso permetta un certo livello di controllo all'interno del sistema. La scelta di avere un unico punto di sincronizzazione ha ridotto il potenziale parallelismo tra le entita' aumentando il livello di contesa nell'ottenere le risorse. In fase di sviluppo e' stato complicato trovare gli errori quando le dinamiche non rispettavano le attese, il modello di concorrenza di Ada ha permesso di utilizzare meccanismi che rispecchiassero i comportamenti che volevamo ottenere.

Distribuzione, che pacco.

IA in Prolog, croce e delizia, genio e sregolatezza.