Soccer-SCD

Simulatore di calcio - Progetto di Sistemi Concorrenti e Distribuiti

Sebastiano Catellani
Sebastiano Gottardo
Alessandro Sforzin

Università degli Studi di Padova
4 agosto 2014

Indice

Storico delle revisioni

Revisione	Data	Autore(i)	Descrizione
1.0	21.07.2014		versione iniziale
1.1	04.08.2014		aggiunto paragrafo iniziale in §2.3; aggiunto paragrafo in §3.3.2;

1 Introduzione

Il seguente progetto è 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 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 dettaglia le informazioni relative alla configurazione e all'esecuzione del software
- nel Capitolo 7 sono presentate le conclusioni

- l'appendice presenta un piccolo manuale utente che spiega come utilizzare il software

2 Funzionalità e modello

Il software che questo progetto mira a realizzare ha lo scopo di creare una simulazione di una partita di calcio. Nella fase di analisi è stata presa in considerazione una partita di calcio reale, al fine di definire tutte le funzionalità del software di simulazione, individuando così con precisione il problema da risolvere. Verrà quindi delineato un modello per la soluzione, mettendo in luce le entità coinvolte e il modo in cui esse interagiscono tra loro.

2.1 Struttura del software

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

Core Il *Core* è il modulo centrale del software. Esso contiene tutta la logica di gioco e regola l'interazione tra le diverse entità che lo compongono. È 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 può 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 è 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, è presente un registro di eventi che permette di ripercorrere testualmente gli eventi salienti della partita.

Manager L'ultima componente è il *Manager*, che permette di controllare una data squadra decidendo eventuali cambi di formazione e sostituzioni da effettuare. È possibile sostituire un giocatore alla volta, fino ad un massimo di tre, mentre non c'è limite ai cambi di formazione che è possibile fare. Le modifiche apportate dall'allenatore vengono applicate alla squadra in occasione della prima interruzione di partita dovuta ad un evento di gioco (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 dedicata in cui sono riportate alcune 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 perché le statistiche non sono più modificabili a partita iniziata. Anche la formazione iniziale della squadra è configurabile e può 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 Funzionalità 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 più partite, ciascuna divisa in due tempi, la cui durata è prefissata e non modificabile dall'utente. L'inizio del secondo tempo è dettato dalla scelta dell'utente, così 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 più alto il valore, tanto più "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 sarà quindi determinata dalla bravura (un valore alto) nel contrasto del primo e dalla capacità di scartare del secondo. Tali caratteristiche sono riassumibili come segue e possono essere configurate prima dell'inizio della partita:

- 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 sé, 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 così 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, così 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 contano 51 celle sul lato lungo e 33 celle sul lato corto; volendo rispettare le proporzioni dettate dalle regole ufficiali del gioco, ogni cella corrisponde a 2 metri quadrati. 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. È possibile assistere allo svolgimento della partita attraverso l'interfaccia grafica del campo di gioco, nella quale vengono visualizzate tutte le informazioni più importanti relativi

alla partita (Sezione 2.1). Lo stato della partita può 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, può effettuare alcune sostituzioni tra un giocatore in campo ed uno in panchina. È 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 più facile prendere decisioni su quali sostituzioni attuare.

2.3 Il modello

La definizione di un modello per il software si suddivide in due fasi, la prima riguardante la distribuzione e la seconda riguardante la concorrenza. Inizialmente, è necessario individuare quali componenti del sistema si prestano in maniera ottimale ad essere partizionate e, successivamente, distribuite. La definizione delle macro-componenti in Sezione 2.1 suggerisce di adottare la stessa separazione anche nell'ambito della distribuzione. Si ha quindi che *Core* viene identificata come componente "centrale", in quanto detiene la logica del gioco (senza la quale le altre due componenti si rivelano inutili): *Field* e le due istanze di *Manager*, anch'esse distribuite, vi interagiscono, al fine di mostrare l'evolversi della partita all'utente e permettendogli al tempo stesso di controllare tutto il sistema. Ne consegue che la comunicazione tra *Core* e le altre due macro-componenti, così come il flusso di dati scambiati, è bidirezionale; non si ha tuttavia alcuna interazione tra la componente *Field* e i due *Manager*, i quali fanno sempre riferimento a *Core*. Maggiori dettagli sulla comunicazione tra le componenti distribuite verranno messi in evidenza nel Capitolo 3.

Una volta definite la struttura generale, la distribuzione delle componenti e tutte le funzionalità del software, è possibile procedere ad identificare le entità e la loro tipologia. Per ciascuna di esse verrà quindi stabilito se rappresenta un'entità attiva, un'entità reattiva oppure una risorsa protetta, ed il loro ruolo all'interno dello scenario in cui si svolge la partita. Si procederà quindi

a definire le interazioni che legano le suddette entità l'una con l'altra, così da poter 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 ultime due categorie è tipicamente dettata dalla mole di lavoro che deve essere svolta in risposta ad una richiesta da parte di un'altra entità.

2.3.1 I giocatori

I giocatori vengono identificati come entità attive, quindi dotate di un proprio flusso di controllo che interagisce con le altre entità 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 partita. Le decisioni che dovranno prendere saranno basate sullo stato circostante e sull'andamento della 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 più fasi, ma questo aspetto verrà ripreso in seguito nei prossimi capitoli.

2.3.2 Lo stato

In fase di analisi è stato messo in evidenza il fatto che un giocatore ha bisogno di apprendere lo stato di gioco per poter decidere la sua mossa successiva. Questo porta alla luce la necessità di avere un luogo unico in cui detenere lo stato di gioco, consultabile dai giocatori ogni qual volta ne abbiano bisogno. Tale stato consiste principalmente nelle posizioni degli altri giocatori in campo e l'andamento del gioco, come per esempio il fatto che la sua squadra sia in fase offensiva o difensiva.

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

Queste considerazioni evidenziano la necessità di aggiungere una nuova entità che governi lo stato, che ne garantisca l'integrità nel proseguire del gioco e garantisca ai giocatori di poter leggere lo stato e modificarlo secondo le mosse scelte: il controllore.

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 ed è in grado di compiere diversi tipi di azioni (Sezione 2.2.2), che decide sia sulla base del proprio stato che dello stato della propria squadra e della partita. Un aspetto fondamentale vede i giocatori, così come avviene nel mondo reale, operare in maniera parallela e concorrente tra di loro; inoltre, i risultati delle loro azioni debbano riflettersi sullo stato di gioco che, come descritto in Sezione 2.3.2, detiene tutte le informazioni che definiscono la partita in un dato istante. Si pone però un problema: i giocatori, per poter conoscere le informazioni che permetteranno loro di decidere la prossima azione da compiere, devono accedere allo stato, che tuttavia è unico. È dunque necessario regolamentare l'accesso a tale risorsa, in maniera tale da preservarne la consistenza e la correttezza.

La componente che si occupa di permettere l'interazione tra diversi giocatori è il *controllore*. In quanto entità 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" (verrà dettagliato in Sezione 2.3.4)

Come già 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, è 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 necessità tale per cui le operazioni che alterano lo stato siano quanto più possibile sequenziali ed atomiche. Sebbene questo argomento verrà trattato ampiamente nel Capitolo 3, è importante capire il motivo dell'importanza di queste due caratteristiche.

Il requisito di atomicità si rivela particolarmente critico quando si assume di operare in condizioni di prerilascio dei processi. Sotto questa condizione, un processo in esecuzione può 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 priorità 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 decidono di provare a conquistarne il possesso. Si supponga che il processo relativo al primo giocatore venga prerilasciato prima di aver potuto applicare la sua mossa sullo stato. 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 nel quale la mossa precedentemente decisa risulta inconsistente, in quanto lo stato è cambiato senza che lui lo sappia. Se si permettesse al primo giocatore di applicare senza alcun controllo la sua mossa si darebbe origine ad una situazione in cui entrambi i giocatori hanno il controllo della palla.

Questo esempio evidenzia come non sia possibile garantire atomicità per l'intero turno di un giocatore senza inficiare il parallelismo, rischiando di ottenere un sistema puramente sequenziale. Più in generale, 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. Tuttavia, un approccio di questo tipo porterebbe ad un'esecuzione strettamente sequenziale del gioco, quindi atomico, e andrebbe a minare il parallelismo potenziale tra i giocatori. Dato che una soluzione di questo tipo non risulta essere desiderabile, l'approccio adottato consiste nello 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.

Il modello che ne consegue prevede che sia solamente il controllore ad effettuare le scritture vere e proprie, sulla base delle azioni decise e sottoposte da parte dei giocatori. Tali richieste vengono valutate in modo sequenziale, andando a modificare lo stato con una azione alla volta ed evitandone così potenziali inconsistenza. 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 necessità di interagire con esso. In questo modo si guadagna un parallelismo potenziale nella seconda fase, permettendo così ai giocatori di non interferire tra di loro nella decisione della prossima mossa.

2.3.4 Verifica sullo stato di gioco: l'arbitro

Affinché una partita si svolga secondo le regole e le modalità stabilite dal gioco del calcio c'è 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 in seguito (e.g. una rimessa laterale)
- Segnalare eventuali irregolarità da parte dei giocatori (e.g. un fallo)
- Tenere il conto dei gol segnati da entrambe le squadre, così 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, così come lo stato e la posizione della palla e la durata della partita fino a quel momento. Nella realtà, il ruolo dell'arbitro è assegnato ad un essere umano, che quindi non è infallibile: si pensi ad esempio ad un fallo che viene commesso irregolarmente alle sue spalle mentre lui è impegnato ad assegnare un calcio d'angolo. In questa simulazione si assume più semplicemente che l'arbitro sia "onnisciente", ovvero abbia la facoltà di analizzare ogni singola mossa di ciascun giocatore e della palla, in maniera da poter segnalare immediatamente ogni irregolarità oppure fermare il gioco all'occorrenza.

Si ha così 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 è 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 sé stante; questa situazione non si verifica invece nel caso in cui sia il controllore ad essere anche arbitro, essendo l'entità centrale che si occupa di eseguire a tutti gli effetti le mosse dei giocatori. Maggiori dettagli sull'implementazione dell'arbitro verranno esposti in Sezione 5.1.

L'elenco delle funzioni che l'arbitro deve espletare nasconde tuttavia un punto critico su cui è 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 verrà 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 è 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'è 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 è quindi il seguente: l'arbitro dovrà prima processare l'evento singolo della componente concorrente (che può causare il gioco fermo, nel caso non lo fosse già) e solo poi processare gli eventi della componente distribuita, se le condizioni sono opportune.

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

La palla è 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 può essere definito come una macchina a stati, schematizzato in Figura 1. 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 può trovarsi solamente in due stati: quiete e moto. Una palla in movimento si può 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.

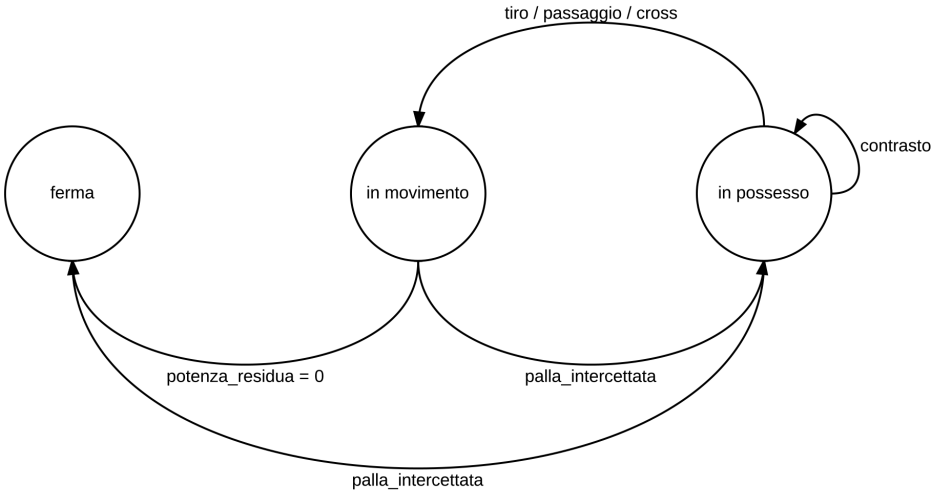

Figura 1: Diagramma dei possibili stati della palla.

Al contrario, una palla è inerte se non è controllata da nessun giocatore, solitamente quando un passaggio o un tiro mancano il bersaglio (e.g. un passaggio troppo debole). La palla è quindi una risorsa protetta, che non compie azioni proprie ma che subisce azioni di altre entità 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'entità che però non può 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 è ricoperta dal cosiddetto *agente di movimento*. L'agente di movimento è un'entità reattiva concorrente agli altri giocatori, il cui unico compito è quello di spostare la palla in una determinata direzione fino a che la potenza ad essa impressa è 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 più semplicemente per una scelta tattica. In questa simulazione i due allenatori rappresentano due componenti distribuite separate, che comunicano con l'unità centrale di controllo. Ciascun allenatore ha la facoltà 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: sarà 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, cosicché i giocatori al proprio turno successivo sappiano che la loro posizione di riferimento è cambiata. Diverso e più complesso è 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 è 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 più 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 è 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 vengono generati dalle diverse entità e possono essere raggruppati in diverse categorie. Inoltre, alcuni di questi eventi interessano la sola parte concorrente, mentre altri eventi possono viaggiare dalla parte concorrente a quella distribuita e viceversa.

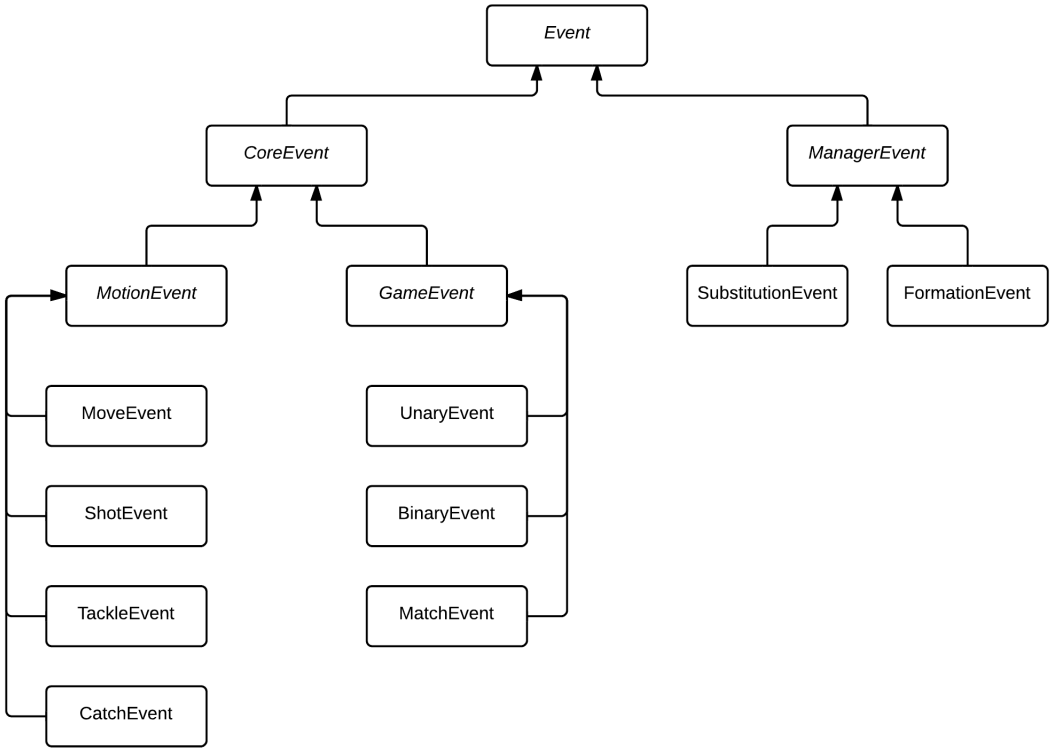

Figura 2: La gerarchia degli eventi che costituiscono una partita.

La struttura globale degli eventi è schematizzata in Figura 2. Data la sua complessità 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 è 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 *Core*) e gli eventi che vengono generati nella parte distribuita (fondamentalmente, dagli allenatori). *Event* può essere considerata come un'entità 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. 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) aggiornamenti 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 quegli 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 può 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 altresì 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 già anticipato, questi eventi vengono anche inviati alla parte distribuita, così 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 così 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 può 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 irregolarità.

BinaryEvent Questa particolare categoria vanta un solo evento possibile: il fallo. Questo evento è 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 modalità 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 è quella di notificare all'arbitro una richiesta di sostituzione per un giocatore oppure di cambio di formazione. L'arbitro, una volta ricevuto l'evento, provvederà 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 dovrà 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 necessità di arricchirlo con meccanismi e componenti dedicate.

La scelta del linguaggio per lo sviluppo della parte principale del software è 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, è 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, il 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 verrà 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, così 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 della palla 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 la palla è slegata dalla velocità di gioco dei giocatori in campo. Essa deve invece essere libera 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 riposizio-

narla, ad esempio, in un punto opportuno a seconda dell'irregolarità. 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 è 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 palla in movimento. È ragionevole pensare che la sua azione possa fallire perché 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 è spostata.

Sulla base delle considerazioni fatte finora, all'interno dell'architettura è 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 è stata quindi studiata in modo tale da permettere ad un entità alla volta di poterla controllare, sia essa uno dei giocatori (attraverso il controllore) oppure l'agente di movimento. Quest'ultimo si attiverà quando risvegliato da un giocatore e sarà fermato quando un altro giocatore conquista il pallone o se il moto che gli era stato impresso si è 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 entità in gioco sarebbe più 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 può cambiare solamente in caso di sostituzione tra un giocatore in campo ed uno in panchina; così facendo non vi è necessità di creare o risvegliare un secondo task.

Area di azione

Ogni giocatore è caratterizzato da alcune statistiche, impostate in fase di inizializzazione, che lo differenziano dagli altri: esse hanno come scopo quello di rendere lo svolgimento del gioco più realistico (Sezione 2.2.2). Abbinate allo stato di gioco del giocatore stesso, rendono inoltre possibile effettuare alcune assunzioni che influiscono nello svolgimento del resto del turno.

Se un giocatore è in possesso della palla e tra le proprie statistiche ha un'alta precisione e potenza, esso potrà effettuare un lancio lungo e raggiungere compagni più distanti; in caso contrario, potrà sempre optare per un passaggio corto ad un compagno vicino. Viene definita in questo modo un'area di interesse del giocatore, entro la quale esso potrà agire a prescindere dall'azione che sceglierà di fare. Un giocatore che chiede al controllore lo stato di gioco della partita non necessita di avere una visione globale dell'intero campo, bensì 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 suddivisa come segue:

1. richiesta al controllore del proprio stato di gioco;
2. individuazione della posizione della palla;
3. richiesta al controllore dello stato della propria area di interesse.

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

La seconda fase consiste nella parte di intelligenza artificiale, nella quale viene decisa la prossima mossa da effettuare. Una volta decisa, viene creato il relativo evento a cui viene automaticamente abbinato un certo livello di utilità: questo valore determina quanto sia importante per il giocatore l'azione appena decisa (ad esempio, l'azione di tiro nell'area di porta avversaria assume un'importanza maggiore rispetto ad un passaggio in fase di impostazione

della prossima azione). L'azione composta da evento e utilità viene sottoposta al controllore tramite una chiamata in mutua esclusione con accodamento.

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

Ci sono altri casi in cui vi è la necessità di fermare (per poi far successivamente 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, così come l'uscita, è 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: al fine di ottenere questo comportamento è stata inserita una cella di campo esattamente adiacente al punto desiderato, 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 anche per effettuare le sostituzioni. Un giocatore che deve essere sostituito esce fino a raggiungere tale cella e successivamente si sposta nel suo posto in panchina: lì il suo identificativo viene cambiato con quello del giocatore entrante. Dato che i giocatori non hanno stato, se non l'identificativo con il quale recuperano ad ogni turno le proprie informazioni dal controllore, modificare l'identificativo equivale ad aver effettuato una sostituzione. In questo modo vengono creati solamente 22 task in ogni partita, che vengono opportunamente riutilizzati per simulare quelli in panchina.

3.2.4 Stato, controllore ed arbitro onnisciente

Lo stato consiste in una serie di dati che descrivono ogni giocatore in campo: un identificativo, il numero di maglia, la posizione attuale e quella di riferimento in campo (cioè la posizione dettata dalla formazione), oltre ad

informazioni generali sullo stato nel suo complesso. Lo scopo delle azioni da parte dei giocatori è quella di andare a modificare la propria posizione all'interno del campo ed effettuare altre operazioni ai fini del gioco.

Il controllore, avvalendosi del suo duplice ruolo di arbitro, è in grado di gestire l'andamento del gioco bloccando, quando necessario, la ricezione di azioni da parte dei giocatori.

Arbitro

Riprendendo quanto detto in Sezione 2.3.4, l'arbitro non è altro che un insieme di funzioni che il controllore svolge per garantire il rispetto delle regole del calcio.

In particolare, si consideri il corpo di esecuzione del controllore all'arrivo di una nuova richiesta da parte di un giocatore. In questo flusso, l'arbitro viene chiamato a seguito dell'esecuzione della mossa del giocatore ed effettua due tipologie di controlli, chiamati simbolicamente *PreCheck* e *PostCheck*.

I controlli effettuati nella fase di PreCheck sono volti principalmente a verificare, in presenza di gioco fermo, che le condizioni per la ripartenza siano soddisfatte (e.g., che tutti i giocatori siano nella loro posizione di riferimento), andando conseguentemente ad aggiornare lo stato di gioco. Inoltre, questa fase viene utilizzata anche per verificare che la sostituzione tra due o più giocatori sia stata terminata con successo.

La fase di PostCheck mira invece a verificare se le condizioni attuali, a seguito della mossa del giocatore, hanno i presupposti per fermare il gioco. Nell'ordine, questa fase effettua i seguenti controlli:

1. verifica se l'ultima azione ha causato un fallo su un altro giocatore;
2. controlla se ci sono eventi provenienti dalle componenti distribuite (e.g., una sostituzione) e, in caso di gioco fermo, le prende in carico;
3. controlla se la palla è uscita dal campo e determina l'azione conseguente.

Al termine di questa seconda fase viene impostato il nuovo stato di gioco, sulla base del quale i giocatori sceglieranno l'azione nel loro prossimo turno. Inoltre tale canale viene sfruttato dal sistema per fermare il gioco e farlo riprendere, per esempio quando l'utente mette in pausa il gioco.

Processare eventi

Nel sistema esistono diversi tipi di eventi e di azioni che un giocatore può effettuare, che dal punto di vista del controllore vengono gestiti in maniera differente: una volta accettata la richiesta del giocatore, il controllore ne determina il tipo e cerca di soddisfarla, così da aggiornare lo stato con l'azione richiesta. D'altro canto, il fatto che il giocatore divida il suo turno in fasi fa sì che i presupposti secondo cui ha scelto una determinata mossa al momento della scrittura non siano ancora validi.

Si consideri la situazione in cui un giocatore tenta di prendere la palla che si trova in una certa posizione, ma dal momento in cui riceve le informazioni sulla posizione e quando tenta effettivamente di prenderne il possesso, la palla si è spostata. Questo comportamento, come anche in altri casi in cui ha senso applicarlo, è dettato dal desiderio di creare una simulazione realistica, in cui un giocatore non sufficientemente veloce o tardivo nell'accorgersi di una certa circostanza veda la propria azione fallire.

In caso di fallimento dell'operazione, ovvero a seguito di un'azione che non può più essere applicata allo stato senza creare inconsistenze, il controllore si comporta in modo differente in base al tipo di richiesta:

- azioni come il passaggio, il tiro, il tentativo di prendere la palla (sia essa controllata da un avversario o meno) se non vanno a buon fine falliscono semplicemente; al turno successivo il giocatore, essendo privo di un proprio stato locale, elaborerà una nuova azione basandosi sulle informazioni fornite dal controllore;
- le azioni di movimento possono fallire per diverse ragioni (e.g., la cella di destinazione potrebbe essere stata occupata), ma possono essere riprovate in un secondo momento, o rivalutate se si verificano determinate condizioni

L'utilità che un giocatore assegna alla propria azione determina il comportamento del controllore nel secondo caso. Al momento dell'accettazione, la richiesta ha una certo valore di utilità (lo stesso assegnatogli dal giocatore): tuttavia, se non dovesse andare a buon fine, questo valore viene decrementato e la mossa viene messa in attesa di una nuova occasione di esecuzione. Ad ogni tentativo di esecuzione il valore di utilità decresce fino a che non diventa minore di una certa soglia fissata a priori, sotto la quale l'azione richiesta viene rivalutata dal controllore.

La rivalutazione viene definita come il soddisfacimento della richiesta originale attraverso un'operazione il più simile possibile a quella decisa dal giocatore.

Tale meccanismo ha come scopo quello di riflettere ciò che succede nelle dinamiche di una partita reale: infatti, se un giocatore si trova improvvisamente un altro giocatore nel percorso da lui deciso, temporeggerà per un attimo (inteso come il decremento dell'utilità) nell'attesa di un suo spostamento, infine cambierà leggermente la propria traiettoria per evitare l'ostacolo.

Un ultimo particolare che coinvolge la rivalutazione di una mossa è il seguente. Un'azione di movimento, per poter essere di nuovo sottoposta al controllore, necessita che il giocatore che la impedisce si sia spostato. Se tale approccio fosse puntuale sulla cella del campo potrebbe risultare troppo oneroso, sia dal punto di vista implementativo che dal punto di vista dell'esecuzione. La soluzione adottata consiste nel permettere la rivalutazione delle mosse fallite di un giocatore ogni qual volta un altro giocatore lasci la propria posizione. Per rendere questo meccanismo più efficiente e scalabile è stato suddiviso il campo di gioco in 6 "zone" (o settori): ogni richiesta che non è andata a buon fine e deve essere ritentata viene accodata sulla zona di appartenenza della cella in cui desidera spostarsi, in attesa che un altro giocatore in quella stessa 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.

Riassumendo, la condivisione di risorse tra processi, se non gestita in modo appropriato, può portare a deadlock; inoltre, dal punto di vista dell'accesso allo stato, la mutua esclusione e l'approccio sequenziale garantiscono la sua assenza. Diversa invece la questione dal punto di vista delle risorse logiche: se due giocatori desiderano l'uno la cella dell'altro vi potrebbe essere una possibile situazione di stallo, in quanto entrambi detengono una risorse e non la cedono fino a che non ottengono quella desiderata. Grazie ai meccanismi descritti in questa sezione, si permette alle azioni di essere rivalutate, portando così i giocatori a modificare le proprie richieste e garantendo quindi l'assenza di deadlock anche in questo caso.

3.2.5 La velocità

Nello svolgimento del gioco finora descritto le caratteristiche fisiche entrano in gioco in diversi frangenti. Quando due giocatori entrano in contatto tra di loro per contendersi la palla, ad esempio, conta soprattutto il contrasto; contano invece attacco e difesa quando un difensore tenta di rubare il pallone all'attaccante e così via. In tutti i casi è il controllore ad esaminare la richiesta di una determinata azione e a considerare le caratteristiche fisiche dei vari giocatori, decidendo quindi chi vincerà il contrasto, quanto preciso

sarà il passaggio, quanto forte applicare al tiro, ...

Diverso è il discorso della velocità, per la quale è necessario un approccio differente. La velocità può essere espressa come la quantità di mosse che un giocatore è in grado di fare in un determinato lasso di tempo, che si traduce nel numero di richieste sottoponibili al controllore in questo frangente. Per semplicità di esposizione, tale quantità di tempo è identificata dal simbolo T, mentre con t_0, t_1, t_2, ... si indicano gli istanti a distanza T durante lo svolgimento del gioco.

Ad ogni giocatore viene abbinato un valore tra 1 e 5 (calcolato automaticamente in base alle sue caratteristiche) per indicare quante mosse sono permesse durante T: di conseguenza, il giocatore con più mosse a disposizione nell'unità di tempo sarà più veloce rispetto ad un giocatore che ha un numero minore; questo numero corrisponde quindi al numero di turni che deve poter eseguire.

Il calcolo di T è di importanza cruciale. Esso deve permettere ad ogni giocatore di eseguire un numero di volte pari al valore che rappresenta la sua velocità. 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, un giocatore un po' più veloce due volte, e così via fino a 5. Vogliamo inoltre 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 desiderabile, in quanto non è possibile sapere quali saranno i giocatori selezionati per eseguire per primi.

Da questo ragionamento si ottiene che T è pari alla somma di tutti questi tempi, ovvero alla somma 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: spetterà poi ad ognuno di essi distribuirli in modo uniforme nell'arco di T, tramite un sistema di delay che scandisce il tempo.

Si consideri il seguente esempio. Posto T pari a 40 unità di tempo, calcolato secondo gli accorgimenti appena discussi, ed iniziato al tempo t_i, C uguale a 2 e la velocità pari a 5: il giocatore (P_i), tra il turno corrente e il successivo, tenterà di eseguire il suo turno entro le prime 8 (40 diviso 5) unità di tempo, per poi attendere fino a $t_i + 8$ il prossimo turno. Tale comportamento è illustrato in Figura 3.

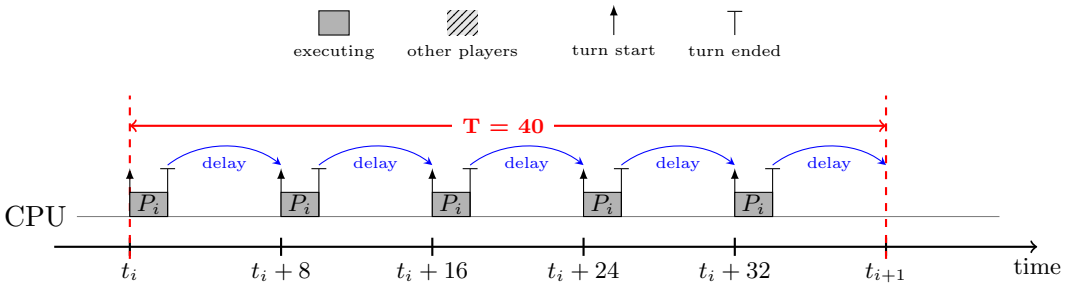

Figura 3: Esecuzione del giocatore con distribuzione dei turni.

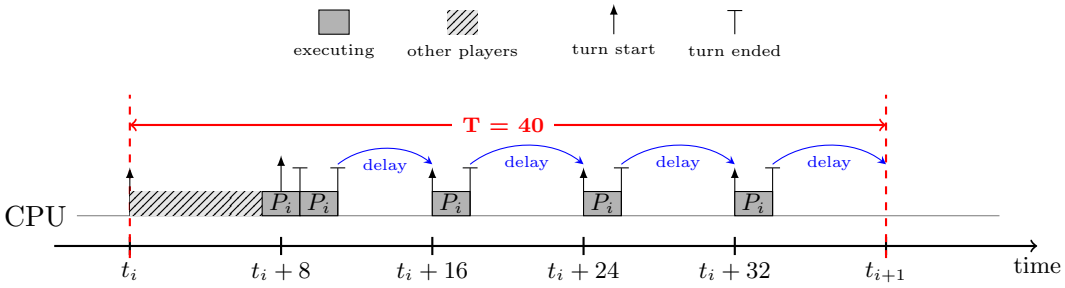

Figura 4: Esecuzione del giocatore con interferenza.

Non è detto che esso riesca ad eseguire per 2 unità di tempo nel periodo di 8, dato che i giocatori iniziano ad eseguire tutti insieme, ma di sicuro eseguirà 5 volte il tempo C (2) all'interno dell'iperperiodo T (40). La Figura 4 porta un esempio di questo tipo, al tempo t_i il giocatore non è il primo ad eseguire, attende il proprio turno e, se già passato, non aspetterà il quanto di tempo successivo.

Dato C come stima fatta a priori, in questo meccanismo il controllore ha il compito di calcolare T e tenere aggiornato il tempo t_i di riferimento. All'inizio del gioco T è pari alla somma di tutti i turni di ogni giocatore, ma sarà il controllore a dover poi mantenerlo aggiornato ogni qual volta ce ne sia bisogno; questo succede, ad esempio, nel caso di una sostituzione tra due giocatori, in quanto non è detto che abbiano la stessa velocità. Per quanto riguarda t_i, in una normale esecuzione della partita viene calcolato partendo da un tempo di riferimento all'avvio del software, al quale viene aggiunto ogni volta T. Nel caso di un'interruzione del gioco da parte dell'utente, invece, tale valore di riferimento viene aggiornato con l'istante in cui il gioco può riprendere.

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 necessità di rendere possibile una comunicazione bidirezionale da e per le componenti distribuite, ovvero *Field* e le due istanze di *Manager*. Di seguito verranno analizzate le soluzioni adottate per questo particolare aspetto del software, la cui rispettiva implementazione verrà trattata nel Capitolo 5.

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, è chiamato *bridge*. A causa della duplice natura delle comunicazioni possibile, il bridge è 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*, è possibile:

- recuperare tutte le informazioni relative ad una squadra;
- recuperare 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 così 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 è solo uscente, a differenza della sua controparte.

Alla base del bridge output è 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 più eventi entro un certo periodo di tempo noto a priori, viene inviato il tutto contenuto del buffer (e viene svuotato). Infine, vi sono eventi più 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 è molto importante e determina il throughtput degli eventi e la conseguente "fluidità" della rappresentazione grafica della partita. Tuttavia, un invio più frequente di eventi può essere causa di una congestione di rete, quindi è opportuno trovare un buon bilanciamento tra quantità di eventi inviati e frequenza di invio. A questo proposito, il bridge output applica una sorta di filtraggio degli eventi nel tentativo di unire più eventi in uno unico. Questo avviene molto spesso negli eventi di movimento dei giocatori e della palla, che sono in assoluto i più frequenti durante una partita. La tecnica che il bridge output adotta è 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 è di gran lunga inferiore e garantisce maggior efficienza senza inficiare sulle performance e sul throughtput.

L'approccio appena descritto introduce però un potenziale problema per la rappresentazione grafica della partita. La mossa originata dall'unificazione di più mosse consecutive avrebbe un punto di inizio e un punto di fine molto distanti tra loro: l'effetto che si otterrebbe sarebbe una sorta di "teletrasporto" del giocatore, che passerebbe dalla sua posizione corrente ad una posizione molto più distante. Per prevenire questo inconveniente, ogni evento viene inviato assieme a due timestamp: il primo timestamp corrisponde alla ricezione del primo evento da parte del buffer, mentre il secondo timestamp

corrisponde alla ricezione dell'ultimo evento; questo intervallo corrisponde quindi alla durata della mossa unificata. Così facendo, la rappresentazione grafica può tenere conto dell'effettiva durata di una mossa e disegnarla di conseguenza.

Utilizzo del bridge Il bridge, poiché risiede all'interno di *Core*, deve essere necessariamente acceduto ed utilizzato da una delle entità presenti in quella componente. Inoltre, dal momento che le comunicazioni dall'esterno sono asincrone e non predicibili, è necessario che l'accesso avvenga serialmente, sia esso in lettura o in scrittura. Per garantire questa caratteristica l'unica entità che si occupa di interagire con il bridge è 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 può 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 è 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 è direttamente il bridge input ad impartire il relativo comando, senza quindi aspettare che sia l'arbitro ad accorgersi della richiesta; tale scelta è stata dettata dal fatto che questa tipologia di richieste ha la massima priorità 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* può 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 funzionalità 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* è 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.

A questo punto è opportuno approfondire la scelta di identificare *Core* stesso con il server per le comunicazioni con le altre componenti. Solitamente, un disaccoppiamento tra *Core* e il server di comunicazione sarebbe desiderabile. Tuttavia, la decisione di unificare queste due componenti è stata dettata da un aspetto puramente tecnologico, che verrà trattato approfonditamente in Sezione 5.2 ma che viene qui brevemente anticipato. La scelta di utilizzare i WebSocket, un protocollo di comunicazione che si appoggia a TCP, unitamente ad Ada Web Server, ha introdotto un maggiore accoppiamento tra *Core* e il modulo di comunicazione con le altre componenti, mitigato solo dalla presenza del bridge (che fornisce una separazione logica sufficiente a mantenere una comprensione globale del sistema piuttosto semplice). Si è tuttavia osservato che i potenziali benefici derivati dall'uso di questa tecnologia relativamente nuova avrebbero potuto essere superiori agli svantaggi che un sistema più accoppiato porta intrinsecamente con sé.

3.3.3 Architettura publisher-subscriber

L'aspetto che vede *Core* comunicare con le altre componenti è 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, è 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 entità agire sia da server che da client a seconda delle necessità, non rappresenta un approccio scalabile e porterebbe un notevole aumento della complessità 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 entità interessate si iscrivono (i consumatori): ogni qual volta il produttore da origine ad un nuovo contenuto, questo viene mandato sul canale corrispondente a tutti coloro che si sono iscritti agli aggiornamenti per quel particolare contenuto. Sarà poi compito dei consumatori quello di processare correttamente il contenuto di volta in volta ricevuto.

Questo approccio risulta particolarmente vantaggioso in quanto mantiene una struttura chiara del sistema e non richiede alcun tipo di meccanismo di interrogazione continua (polling). Inoltre, è 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'impossibilità del produttore di contattare direttamente i consumatori all'originarsi di un nuovo contenuto di loro interesse.

3.3.4 Workflow

In Figura 5 viene fornita una schematizzazione del modello di comunicazione tra le componenti distribuite. Si è cercato di enfatizzare l'uso dei due diversi pattern utilizzati (client/server e publisher/subscriber) anche per quanto riguarda il flusso di informazioni: si ha infatti che nel primo caso le componenti *Field* e *Manager* possono sia richiedere informazioni sulla partita che modificarle (e.g., un cambio di formazione); diversamente, nel secondo caso *Core* si occupa semplicemente di notificare le altre componenti sui recenti avvenimenti della partita.

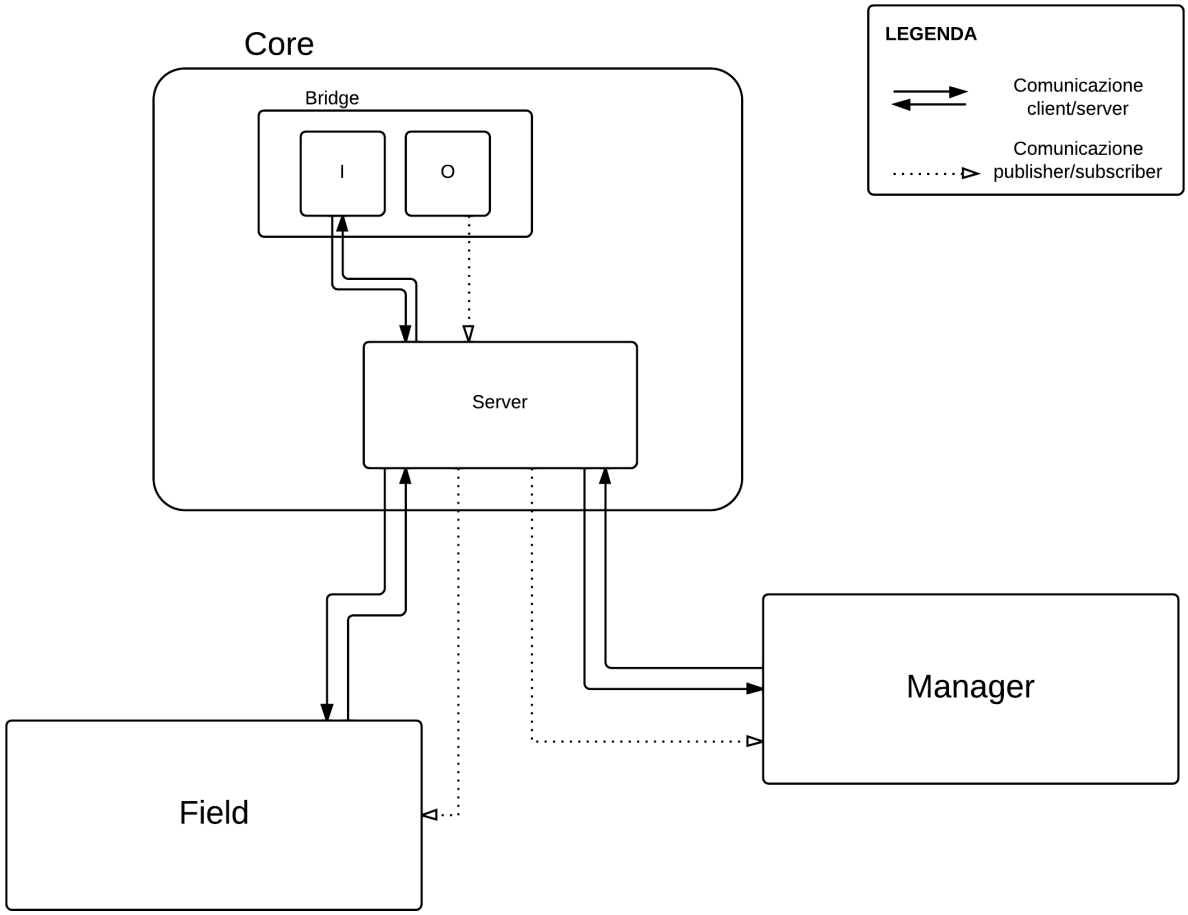

Figura 5: Diagramma della comunicazione tra le componenti distribuite.

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 è stato utilizzato Prolog, un linguaggio di programmazione che trova le sue radici nella logica del primo ordine. [1]. Come tale, 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 artificiale 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 costruirsi tali predicati ciascun giocatore deve procurarsi informazioni riguardanti la situazione in prossimità della sua posizione (la sua posizione, quale squadra ha la palla, quali giocatori sono vicini a lui e così via) interrogando lo stato.
Più precisamente, per ciascuna informazione che il giocatore chiede ed ottiene dallo stato, viene costruito il corrispondente predicato logico scritto in Prolog. Vi sarà quindi, ad esempio, un predicato per la posizione del giocatore, un predicato per la posizione della palla, un predicato per ciascun giocatore presente nelle vicinanze del giocatore corrente e così via. Ovviamente nello stato sono presenti tipi di dato non interpretabili da Prolog. È quindi necessario un passaggio intermedio che ci permetta di trasformare le informazioni contenute nello stato in un formato comprensibile a Prolog.
Questa conversione è molto semplice ed avviene man mano che il giocatore riceve dati dallo stato. Prendiamo come esempio la posizione attuale del

[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.

giocatore. Il giocatore richiede allo stato questa informazione, il quale gli restituisce un oggetto di tipo *Coordinate*, un tipo di dato che rappresenta la posizione con una coppia di interi (x,y). A questo punto avviene la conversione. Dall'oggetto di tipo Coordinate sono estratti i due interi x e y, i quali vengono inseriti in una nuova stringa contenente la posizione del giocatore, scritta sotto forma di predicato logico capibile da Prolog. Se supponiamo che la posizione del giocatore presente nello stato sia la coppia di interi (10,15), la corrispondente stringa generata sarebbe:

```
position(10, 15).
```

Questa operazione viene ripetuta per tutte le richieste che il giocatore fa allo stato al fine di ottenere un quadro generale di ciò che sta succedendo nella partita.
Una volta esaurite tutte le richieste allo stato, le varie stringhe così ottenute vengono concatenate in un'unica nuova stringa, che rappresenterà la base di conoscenza del giocatore. Questa nuova stringa viene passata come parametro, tramite uno script adibito all'esecuzione del motore di Prolog e alla ricezione del risultato dell'inferenza logica effettuata sulla base di conoscenza fornita come parametro, alla seconda parte del sistema.
Questa seconda parte è formata da predicati che rappresentano le azioni che i giocatori possono effettuare. A sua volta, essa si suddivide 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. L'algoritmo di conversione delle informazioni da tipo di dato Ada a stringhe contenenti i predicati scritti secondo i formalismi di Prolog elabora i dati ottenuti dallo stato
3. Viene lanciato, tramite un apposito script, il motore Prolog fornendo come input la concatenazione di predicati ottenuta al passo precedente
4. Prolog esegue un inferenza logica sull'input ricevuto al passo precedente e fornisce in output la mossa ottimale per il giocatore data la situazione corrente

La mossa ottenuta in output è ricevuta sotto forma di stringa Ada contenente un predicato Prolog costituito dalla mossa da effettuare e una coppia di interi (x,y) che rappresentano la coordinata sulla quale il giocatore andrà ad effettuare la mossa. Questa stringa viene scomposta in tre sottostringhe: una contenente la mossa da effettuare (ad esempio *shot*, il che sta ad indicare che il giocatore "ha deciso" di tirare), una stringa per la componente x della coordinata e infine una per la componente y. A partire da queste sottostringhe viene creata la mossa che il giocatore cercherà di scrivere sullo stato, avendo cura di creare un oggetto di tipo *Coordinate* a partire dalle due sottostringhe contenenti le componenti della coordinata finale.

4.4 Problematiche

Verso la fase finale del progetto ci si è resi conto di un problema prestazionale che vedeva i giocatori impiegare un tempo esageratamente alto per completare il proprio turno. Dopo un'analisi iniziale, il collo di bottiglia è stato identificato con la parte che riguarda la scelta della prossima mossa, quindi il processo decisionale che riguarda Prolog.

Inizialmente, l'approccio naîve adottato consisteva nell'invocare il motore di Prolog attraverso una libreria Java, dove sia input che output passavano attraverso dei file temporanei. Questo comportava un overhead dovuto all'istanziazione di una JVM (Java Virtual Machine) per giocatore, unita all'input/output su disco. Successivamente, si è optato per la generazione di eseguibili a partire dalle clausole di Prolog, opportunamente generate in Ada. Questo approccio ha notevolmente ridotto il tempo di esecuzione di ciascun turno, ma tale tempo continuava ad essere alto. Il problema sembra quindi essere dovuto all'elevato numero di istanze del motore di inferenze di Prolog (una per giocatore), che in alcuni casi porta ad un aumento sensibile del tempo di calcolo.

È stato valutato il passaggio ad un motore inferenziale di Prolog interamente scritto in Ada, tuttavia la sua implementazione era basata su un motore inferenziale unico, rendendo così impossibile un'invocazione parallela da parte di più giocatori. Si è quindi deciso di mantenere la soluzione corrente e arginare, per quanto possibile, il problema.

5 Implementazione

In questo capitolo vengono dettagliate le scelte implementative alla base del software di simulazione. Pur non trattando singolarmente ciascuna di esse, si cercherà di mettere in evidenza i punti più significativi, sia nell'ottica della concorrenza che nell'ottica della distribuzione.

5.1 Concorrenza

Questa sezione consta di alcuni aspetti implementativi legati alla concorrenza del sistema. In particolare, saranno dettagliati i seguenti aspetti:

- avvio del sistema;
- riaccodamento dei giocatori;
- terminazione del sistema.

Verrà infine illustrato un diagramma di sequenza che focalizza alcuni degli aspetti sopra elencati.

5.1.1 Avvio del sistema

L'avvio del sistema prevede, per quanto riguarda la parte *Core*, l'inizializzazione di diverse componenti, prime fra tutti i giocatori: i 22 task di tipo Player vengono avviati simultaneamente nel `main` del software (il begin corrisponde quindi ad un co-begin) e, dal momento che sono le uniche entità attive del sistema, la loro terminazione determinerà la terminazione di tutto il software.

Dal momento che l'inizio di una partita prevede prima una configurazione delle squadre e dei relativi giocatori, i 22 task si accodano inizialmente su una risorsa protetta chiamata *GameEntity* attraverso il canale `Rest`, la cui guardia si apre solamente quando arriva il segnale di inizio di una nuova partita o di inizio del secondo tempo. Alla sua apertura, i task recuperano il proprio identificativo attraverso il canale `GetId` del controllore, che corrisponde ad uno dei giocatori in campo. A questo punto, i giocatori fanno il loro ingresso in campo e, una volta raggiunta la loro posizione di riferimento, si accodano nuovamente su *GameEntity*: la guardia del canale `Start1T` si apre solamente quando l'arbitro ha decretato che tutti i giocatori sono in posizione e che la partita è pronta per cominciare. I giocatori procedono quindi il loro ciclo di lettura dello stato, decisione della prossima mossa e successiva scrittura sullo stato.

5.1.2 Riaccodamento

Nel Capitolo 3 si è parlato di come possa accadere che una mossa non sia soddisfacibile al momento della richiesta. La mossa, a seconda della sua tipologia e dell'utilità assegnatavi, può essere rivalutata con una mossa che si avvicini il più possibile alla richiesta originale del giocatore. D'altro canto, ci sono circostanze in cui la mossa è necessaria e una sua rivalutazione causerebbe un comportamento di gioco lontano dalla realtà.

Il meccanismo che realizza il riaccodamento per una mossa momentaneamente non attuabile utilizza una risorsa protetta chiamata *Guard*, che mette a disposizione due canali: `Wait` e `Update`. Un giocatore la cui mossa non può essere eseguita viene riaccodata su tale risorsa attraverso il canale `Wait`, in attesa che venga eseguita un'azione di movimento (l'unico tipo di azione rivalutabile) sulla zona di campo corrispondente. Dualmente, ogni volta che un giocatore effettua un'azione di movimento, aggiorna *Guard* attraverso il canale `Update` per notificare il movimento in quella particolare zona di campo. Si ha così che, nel caso di azioni di movimento in attesa, esse non vengono "risvegliate" ad ogni altro movimento di un giocatore, ma sono localizzate in una specifica zona di campo.

5.1.3 Terminazione del sistema

La terminazione del sistema viene richiesta dall'utente, attraverso i comandi messi a disposizione nell'interfaccia grafica di *Field*. Una volta ricevuta da *Core*, il controllore fa in modo di inoltrarla anche alle due istanze di *Manager*, che termineranno di conseguenza la loro esecuzione. Inoltre, il controllore aggiorna lo stato in maniera da notificare ai giocatori la richiesta di terminazione. Infine, il controllore richiede la terminazione del software attraverso la procedura `Exit_Program`.

5.1.4 Diagramma di sequenza

Il diagramma di sequenza mostrato in Figura 6 schematizza il flusso base della simulazione. In esso viene rappresentato l'avvio del sistema, che comprende la creazione dei task per i giocatori e l'assegnamento dei loro identificativi. Con il loop centrale si identifica il turno di un giocatore, comprensivo di lettura dello stato (informazioni generali e calcolo del raggio d'azione), decisione della mossa e sua successiva scrittura attraverso il controllore; la sua elaborazione, etichettata *Compute*, comprende anche l'eventuale riaccodamento e/o rivalutazione. Una volta completata la scrittura, la mossa viene

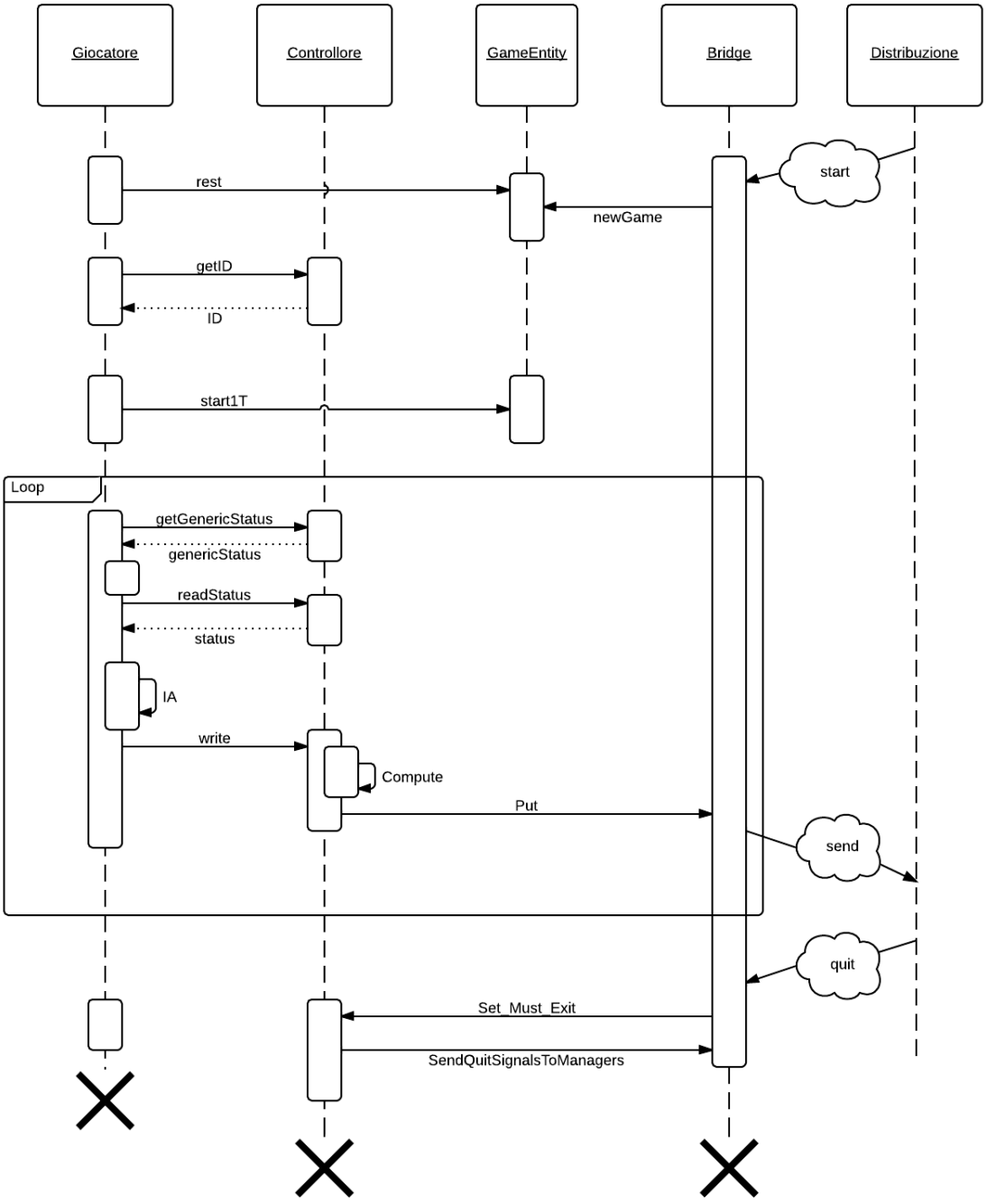

Figura 6: Diagramma di sequenza che comprende avvio del sistema, mossa di un giocatore e terminazione del sistema.

aggiunta nel buffer ed inviata alla distribuzione. Viene infine rappresentata la terminazione del sistema, intesa come la terminazione delle entità distribuite unitamente alla successiva terminazione di *Core*.

5.2 Distribuzione

La comunicazione tra la componente *Core* e le componenti *Field* e *Manager* è stata realizzando aggiungendo un modulo a *Core* che agisce come web server. Per rendere possibile questa interazione si è ricorsi all'utilizzo di AWS (Ada Web Server). All'avvio del sistema, viene inizializzato un web server su *localhost:28000*, a cui *Field* e *Manager* si connettono e con il quale successivamente scambiano informazioni.

Come già illustrato in Sezione 3.3.2, 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. Si ha quindi che ciascuna istanza di *Manager*, una volta avviata, invia una richiesta HTTP di tipo GET a *Core*, richiedendo le statistiche dei giocatori della squadra. La componente *Field* effettua invece chiamate HTTP di tipo GET verso *Core* al fine di richiedere azioni quali l'avvio di una nuova partita, l'avvio del secondo tempo della partita corrente, la messa in pausa della partita corrente, ed infine la terminazione forzata della partita.

Dal momento che *Manager* e *Field* sono state realizzate in Java, per facilitare tale tipo di comunicazione è stata utilizzata la libreria open source Apache HttpComponents, che fornisce una completa implementazione del protocollo HTTP in Java e consente di avere accesso a funzionalità avanzate e maggiore 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 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.

Core mette infatti a disposizione i seguenti canali di comunicazione ai quali le componenti interessate si iscrivono per ricevere informazioni:

- */managerVisitors/registerForStatistics* è il socket 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 socket 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 socket a cui si connette *Field*.

Per realizzare questo tipo di comunicazione si è deciso di utilizzare la tecnologia dei WebSocket. Questo particolare tipo di socket mantiene attivo un canale di comunicazione tra due componenti, ad esempio *Core* e *Field*, permettendo così la fruizione di contenuti di *Core* da parte di *Field*. Il principale vantaggio nell'utilizzo dei WebSocket consiste nel fatto che l'invio di nuovi contenuti disponibili da parte dell'entità produttore verso le entità consumatore avviene senza alcuna richiesta o sollecitazione da parte di questi ultimi: non appena è disponibile un nuovo contenuto, il produttore lo invia autonomamente ai consumatori, i quali sono in ascolto sui corrispondenti WebSocket aperti. 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.

In fase decisionale si è osservato che l'accoppiamento tra la componente *Core* e il server per la comunicazione con le altre componenti potesse non essere desiderabile. Ad ogni modo, si è deciso di procedere ugualmente con questa decisione, sacrificando una parte di disaccoppiamento del sistema in favore dell'uso di una tecnologia relativamente nuova ed interessante come quella dei WebSocket, che è risultata essere estremamente affidabile e di facile utilizzo. Si noti che, al momento dell'integrazione, il supporto di Ada Web Server ai WebSocket era ancora molto sperimentale e non ufficializzato: fortunatamente, l'implementazione esistente vantava già un ottimo livello di stabilità.

Ada Web Server mette a disposizione una serie di API per facilitare l'uso dei WebSocket. Per aprire un nuovo socket è sufficiente utilizzare la funzione *Register*, passando come parametro l'indirizzo su cui renderlo disponibile. Per inviare informazioni viene messa a disposizione la funzione *Send*, specificando il WebSocket sul quale spedire i dati assieme alla tipologia di dato che si sta trasmettendo.

Come spiegato in sezione 3.3.1, non vi è un continuo flusso di informazioni verso le componenti distribuite, ma viene utilizzato un buffer all'interno di bridge output, il cui scopo consiste nel 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.

5.2.1 Codifica delle Informazioni

Tutti i messaggi scambiati secondo i modelli appena descritti sono codificati utilizzando il formato JSON, 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 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 metodo *toJson()* messo a disposizione dalla libreria. Similmente, la conversione di una semplice stringa JSON nel corrispondente oggetto Java è resa possibile dal il metodo *fromJson()*.

5.2.2 Interfacce Grafiche

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

6 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)

- Scompattare [AUNIT FILE] e lanciare i seguenti comandi

  ./configure
  make
  make install (eseguire da root)

- Scompattare [ADA UTIL FILE] e lanciare i seguenti comandi

  ./configure
  make
  make install (eseguire da root)

- 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.

6.1 Repository

Tutto il codice generato viene versionato in un apposito repository ospitato dal servizio github ed è accessibile liberamente. È quindi possibile scaricare tutto il progetto e la relativa documentazione.
Il repository è stato strutturato in moduli rappresentanti le varie componenti del progetto:

- *GUISoccerField* contiene il codice relativo alla GUI del campo di gioco, sviluppata in Java
- *ManagerGUI* contiene il codice relativo alle GUI dei manager, sviluppate in Java
- *Players-Generator* contiene il codice relativo alle GUI di configurazione delle squadre, sviluppate in Java
- *SCDCommunication* contiene il codice relativo allo scambio di informazioni tra le componenti distribuite *Field* e *Manager* e la componente *Core*, anch'esso sviluppato in Java
- *Soccer-SCD* contiene il codice del progetto vero e proprio, sviluppato in Ada
- *relazione* contiene la documentazione del progetto

Per scaricare in locale tutto il progetto è sufficiente aprire un terminale e lanciare il comando

```
git clone --recursive https://github.com/dextorer/SCD.git
```

7 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 l'importanza di creare in fase di design un'architettura che sia esente dalle condizioni di deadlock e di starvation, ma che permetta allo tempo stesso un certo livello di controllo all'interno del sistema. La scelta di avere un unico punto di sincronizzazione, identificato con lo stato, ha ridotto il potenziale parallelismo tra le entità, aumentando il livello di contesa nell'ottenere le risorse. Uno dei problemi che hanno inciso maggiormente nella fase di sviluppo è consistito nel trovare gli errori quando le dinamiche di concorrenza non rispettavano le attese: ad ogni modo, il modello di concorrenza di Ada ci ha permesso di utilizzare meccanismi che rispecchiassero i comportamenti che volevamo ottenere, agevolando di molto la risoluzione dei problemi.

Diversamente, la distribuzione ha messo il luce le difficoltà nel coordinare entità remote che molto spesso sono regolate dall'interazione con l'utente. La scelta di utilizzare i WebSocket si è rivelata efficace e non ha presentato particolari problemi, salvo un maggiore accoppiamento all'interno del sistema. La parte che riguarda l'invio dei dati da parte di *Core* e la loro successiva rappresentazione grafica si è rivelata invece più problematica, in quanto ha reso evidenti le necessità di adottare un filtro sugli eventi e un buffer di eventi regolato da un timer, sia per regolare il flusso di rete (riducendo quindi il rischio di congestioni) che per garantire una certa fluidità di visualizzazione della partita.

Infine, la scelta di usare Prolog per realizzare l'intelligenza artificiale dei giocatori ha avuto due effetti. Se da un lato ha permesso di esplorare un linguaggio appositamente progettato per questo tipo di compito e di avere quindi un'espressività maggiore da parte dei giocatori, dall'altro ha inficiato notevolmente la velocità di esecuzione del software. Quest'ultimo aspetto non era stato né previsto né tantomeno esplorato in fase di analisi, di conseguenza non c'è stato altro da fare che cercare di ridurlo quanto più possibile. Alla luce di ciò, l'uso di Prolog si è rivelato controproducente.

A Manuale d'uso

Una volta avviato il sistema, all'utente viene presentata immediatamente la GUI del campo di gioco:

Figura 7: GUI del campo di gioco.

Sopra la rappresentazione del campo sono presenti i controlli per modificare lo stato del gioco, attraverso i quali è possibile iniziare una nuova partita, mettere in pausa quella corrente oppure terminarla. In basso, il pannello *Statistics* riporta il risultato corrente della partita ed il tempo di gioco. Infine, è stato inserito un *Log* in cui è possibile visualizzare gli eventi principali della partita in corso.

Per poter iniziare una nuova partita devono essere configurate entrambe le squadre partecipanti, tramite le interfacce dei manager. Per fare ciò,

all'avvio del sistema, oltre alla GUI del campo di gioco, vengono presentate due semplici GUI (una per ogni squadra) da cui è possibile avviare i manager:

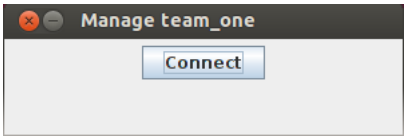

Figura 8: Finestra di avvio dei manager.

Premendo il pulsante *Connect* viene di fatto avviato il manager, il quale si connette alla partita. Una volta premuto il pulsante *New Game* presente nell'interfaccia del campo di gioco, viene presentata all'utente la GUI di configurazione della squadra:

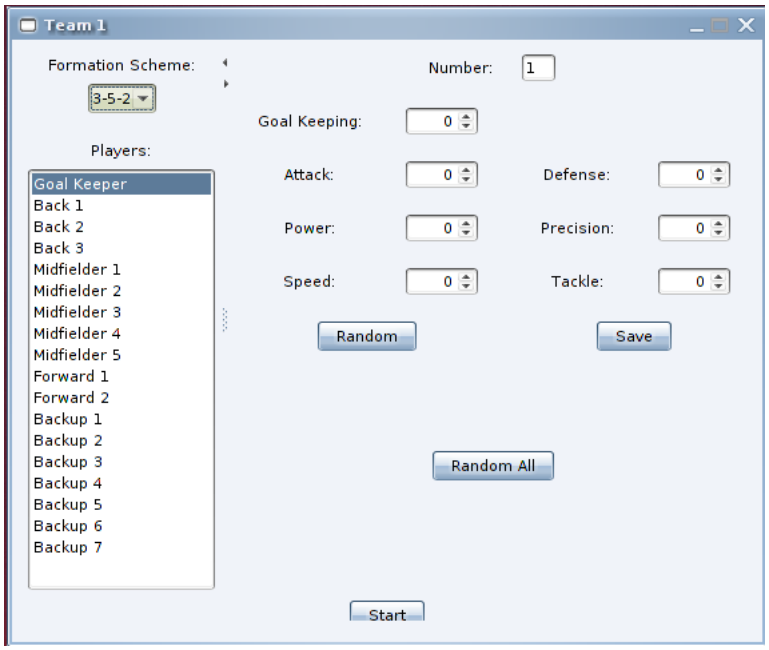

Figura 9: GUI di configurazione della squadra.

L'interfaccia è suddivisa in due parti: il pannello laterale e il pannello delle statistiche. Il pannello laterale permette di cambiare la formazione della squadra utilizzando il menù a tendina, intitolato *Formation Scheme* e posizionato in alto; inoltre permette di selezionare comodamente ciascun giocatore dalla lista *Players*. Dopo aver selezionato un giocatore, è possibile vedere e modificare le sue statistiche dal pannello di destra. All'avvio, tutte le statistiche dei giocatori assumono valore zero. Dal pannello delle statistiche è possibile impostare manualmente il numero di maglia del giocatore e tutte

le sue statistiche. Queste ultime devono essere comprese tra zero e cento. Il pulsante *Random* permette di generare casualmente tutte le statistiche del giocatore correntemente selezionato. Una volta assegnati i valori desiderati alle statistiche, le modifiche effettuate vanno salvate tramite il pulsante *Save*. È anche possibile generare casualmente e salvare tutte le statistiche di tutti i giocatori utilizzando il pulsante *Random All*. Infine, premere *Start* per terminare la configurazione della squadra.

Una volta terminata la configurazione di entrambe le squadre, la partita ha inizio e i giocatori iniziano ad entrare in campo. Ecco lo stato dell'interfaccia del campo di gioco una volta configurate le squadre e avviata la partita:

Figura 10: GUI del campo di gioco a partita avviata.

Durante il corso della partita ciascun manager ha a disposizione una GUI dalla quale può gestire la propria squadra:

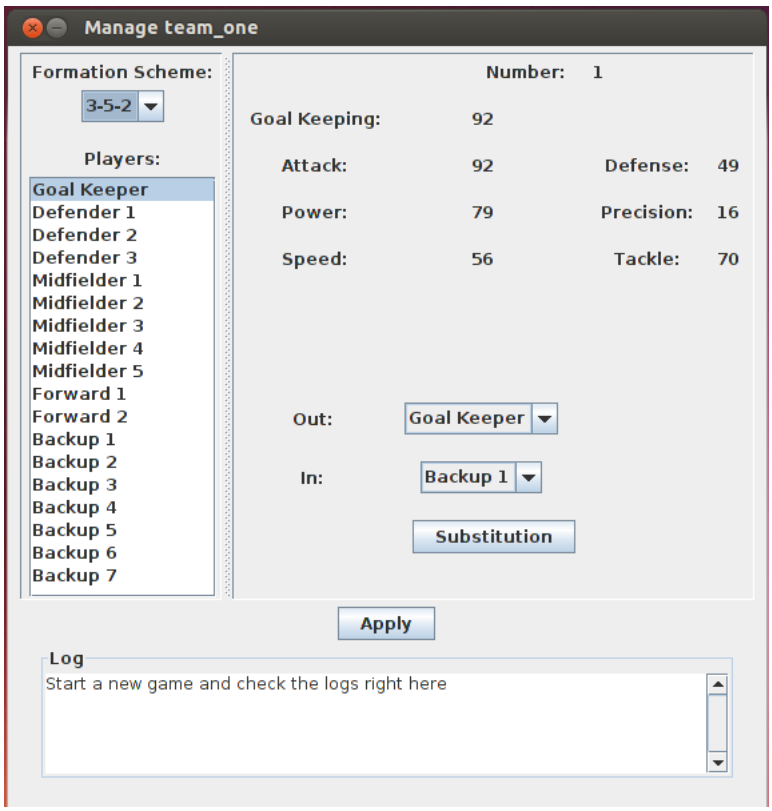

Figura 11: GUI del manager.

Sulla sinistra, si può vedere lo stesso pannello laterale presente nella finestra di configurazione delle squadre: da esso è possibile cambiare la formazione della squadra e selezionare un giocatore.
Il pannello di destra riporta le statistiche del giocatore selezionato. Si noti che non è possibile modificare le statistiche dei giocatori a partita avviata. In questo pannello sono presenti anche i comandi per avviare una sostituzione. In questo contesto, il menù a tendina *Out* permette di selezionare il giocatore da sostituire, mentre il menù a tendina *In* permette di selezionare il giocatore da far entrare in campo. Per confermare la sostituzione è sufficiente premere *Substitution*. È possibile effettuare una sostituzione per volta.
Per salvare e applicare le modifiche effettuate, siano esse una cambio di formazione o una sostituzione, premere il pulsante *Apply*.
Infine, in basso, è presente un *Log* da cui è possibile visualizzare comodamente gli eventi principali della partita in corso.