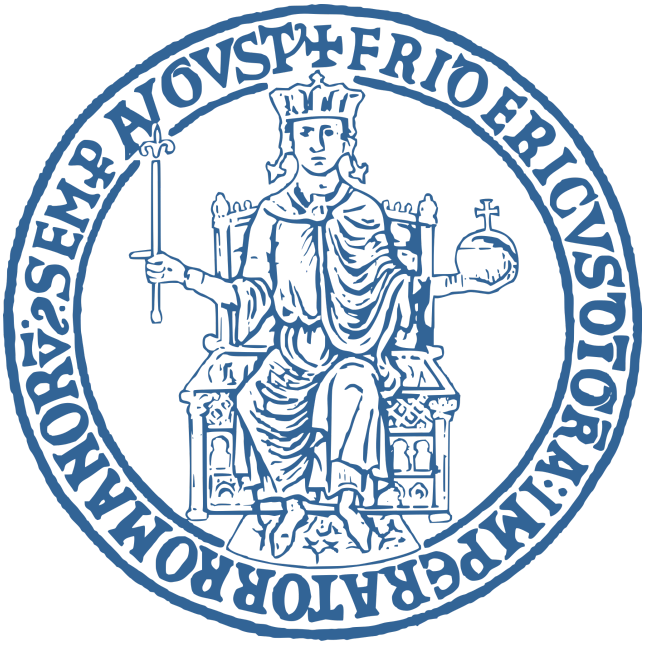


Progettazione e realizzazione di un'applicazione Web Client-Server per simulazione di partite di Calcio a riga di comando:

LSOccerSim

Ascione Antonio *Fortunato D'Errico* *Anastasio Ciro*

N86004176 *N86004203* *N86004139*

17/06/2024

Indice

Sezione 1:

Descrizione commissa

“Lo studente dovrà realizzare la simulazione di un sistema che modella una partita di calcetto. Per ogni partita ci sono due squadre, A e B, e un arbitro. Ogni squadra ha 5 giocatori. Tutti i giocatori in loop cercano di accedere ad un pallone, chi riesce ad impossessarsi del pallone, inizia a giocare la partita.
La simulazione della partita può avvenire secondo i seguenti eventi:

- Infortunio -> il giocatore si infortuna e quindi lascia la partita per un numero casuale di minuti;
- Tiro-> il giocatore tira in porta. In maniera casuale farà goal. L'arbitro comunicherà il goal;
- Dribbling -> come per il tiro, questo avviene in maniera casuale. Se il dribbling riesce, allora può tentare nuovamente un altro tiro, altrimenti il pallone viene liberato e può essere preso da un altro giocatore.

L'arbitro crea un file di log con gli eventi della partita, inclusi numero di goal, infortunio (associato al giocatore e squadra), numero tiri falliti, numeri di dribbling. La partita terminerà dopo ‘N’ tempo.
Le squadre vengono create da un capitano (client), il quale accetta nella sua squadra altri 4 client.
Sia il capitano che gli altri giocatori partecipano alla partita.

Opzionale: l'infortunio coinvolge due giocatori: il giocatore infortunato, e un giocatore della squadra opposta scelto in maniera random. Il giocatore che ha fatto il fallo, avrà una penalizzazione di ‘P’ tempo”.

Sezione 2:

Scelte implementative

2.1 Lato client

Il Client della nostra applicazione è dotato della sola logica di apertura della Socket per comunicare, e di interpretazione dei messaggi del server che sono composti da due parti:

- Il tipo di messaggio;
- Il contenuto del messaggio.

Per ogni tipo, il client stampa sempre il contenuto, poi esegue un'azione diversa a seconda del tipo:

- Messaggi di lettura: il client manda al server (in attesa) un messaggio vuoto di conferma della lettura;
- Messaggi di risposta: il client chiede all'utente un input da inserire nel messaggio che verrà rimandato al server;
- Messaggi senza ritorno: il client, dopo la stampa, non restituisce nulla al server, che continua la sua esecuzione;
- Messaggi di errore: il client ritorna al server un messaggio con scritto *CLI_ERROR;*
- Messaggio di uscita: l'ultimo messaggio ricevuto dal client, stampa il contenuto e termina la sua esecuzione.

Se il messaggio non rientra in nessuna di queste categorie, il client manda un *SRV_ERROR*.

Segue un frammento del codice del Client:

```c
int sock_fd, status;
struct sockaddr_in servaddr;

if ((sock_fd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Failed to create socket");
    exit(EXIT_FAILURE);
}

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(12345);
if (argc == 2) { servaddr.sin_addr.s_addr = inet_addr(argv[1]); }
else { perror("Address"); exit(1); }

int n;

if (status = connect(sock_fd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0)
{
    perror("client");
    exit(EXIT_FAILURE);
}
```

```c
    for(int i = 0; i < strlen(printBuff); i++)
    {
        if(printBuff[i] == '[') i= i+3;
        printf("%c", printBuff[i]);
    }

    switch (rBuffer[1])
    {
        case '0':
            write(sock_fd, "", 1);
            break;
        case '1':
            char wBuffer[BUFFSIZE];
            fgets(wBuffer, BUFFSIZE, stdin);
            write(sock_fd, wBuffer, strlen(wBuffer));
            break;
        case '2':
            break;
        case '3':
            write(sock_fd, "CLI_ERROR", 10);
            break;
        case '9':
            rdout = -1;
            break;

        default: write(sock_fd, "SRV_ERROR", 10);
            break;
    }
```

2.2 Lato server

Dal lato server la struttura è più complessa, in quanto contiene la totalità della logica di funzionamento:

1. Creazione della Socket;
2. Accettazione dei Clients tramite multi-threading;
3. Inserimento dei Dati e mantenimento giocatori;
4. Gestione del popolamento delle squadre;
5. Svolgimento della partita e avvenimento degli eventi;
6. Realizzazione output formattato;
7. Registrazione della partita in un file creato appositamente.

2.2.1 Creazione della Socket

L'esecuzione del server inizia con la creazione della Welcoming Socket sulla quale effettua una bind alla porta "12345" e poi si mette in ascolto ("listen") sulla stessa per accettare eventuali richieste da uno dei qualsiasi client, con una coda massima di 10 client.

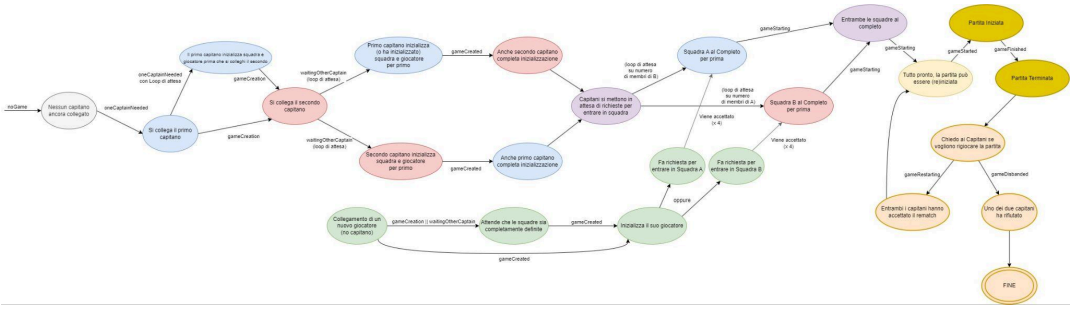
2.2.2 Accettazione dei Clients (multi-threading)

Dopo aver accettato ("accept") un nuovo client, viene creata una nuova socket dedicata alla comunicazione tra il server e quello specifico client e per ognuno di questi ultimi creiamo un thread apposito, a cui passiamo la socket precedentemente creata e diamo l'informazione Detach, per agire in maniera slegata dal main.
In ogni thread, eseguiamo dapprima la logica di creazione del giocatore.

2.2.3 Inserimento dei Dati e mantenimento giocatori

Nelle funzioni dei thread, al giocatore viene assegnato un ruolo, che può essere di capitano (i primi due Clients collegati) o di semplice

partecipante: per fare ciò utilizziamo una serie di structs (Referee, Team, Player) insieme a una enum che rappresenta tutti i possibili stati della partita, dalla creazione fino alla sua terminazione.
Il file server.c è dotato di una variabile globale Ref del tipo struct referee, che viene inizializzata con i suoi valori default a inizio esecuzione, e contiene le due squadre sotto forma di struct team, che a loro volta mantengono i 5 giocatori sotto forma di struct player: ognuna di queste strutture mantiene tutte le informazioni che descrivono queste entità. La logica di decisione dei capitani è definita in questo grafo:



Cliccare per visualizzare la corrispondente immagine vettoriale

Una volta assegnato il ruolo, i capitani daranno il nome alla propria squadra, a cui sarà assegnato anche un valore char a seconda che siano la squadra A o la squadra B; tutti i giocatori (anche i capitani) inseriranno un nome e un numero di maglia per identificare il loro giocatore: tutte queste informazioni saranno inserite nei campi delle struct corrispondenti.
A questo punto, ogni giocatore dovrà fare richiesta di ingresso in una delle due squadre.

2.2.4 Gestione e popolamento delle squadre

Per le richieste e risposte relative all'ingresso in una squadra, usiamo una combinazione di pipe e una struct definita ad-hoc, che non è dissimile da una Queue implementata su Array:

```
#define QSIZE 15

struct playerQueue
{
    short added;
    short viewed;
    struct player players[QSIZE];
};
```

Sono implementate poi varie funzioni per utilizzarla più agilmente, come IsEmpty, Head, Next e Insert.

Nel momento in cui un client (non capitano) completa la registrazione come giocatore, gli sarà chiesto se vuole essere parte della squadra A o B, alla cui scelta seguirà l'inserimento automatico nella coda corrispondente e l'avvio della lettura della Pipe da cui riceverà la risposta: il capitano di quella squadra riceverà il giocatore dalla coda, deciderà se accettarlo o meno, e manderà quindi questa scelta tramite la pipe secondo l'encoding R-T-P (Response-Team-Position).
Se il client venisse rifiutato dal capitano della squadra in cui ha fatto richiesta di entrare allora il suo thread morirebbe e dovrebbe collegarsi un nuovo client per completare le squadre e far iniziare la partita. Invece, se il giocatore viene accettato riceverà la risposta e imposterà tutte le informazioni correttamente nella sua struct player.
Quando ogni capitanoavrà accettato 4 giocatori (per un totale di 5 membri), la partita avrà inizio.

2.2.5 Svolgimento della partita e avvenimento degli eventi

I thread che rappresentano i giocatori delle due squadre accedono concorrentemente ad un semaforo (metaforicamente la palla) e quando la ottengono eseguono uno dei diversi eventi tra: tiro, eventualmente seguito da un goal; dribbling, eventualmente seguito da un tiro; infortunio che porta i thread interessati, chi subisce l'infortunio e chi lo causa, ad attendere un numero randomico di secondi prima di poter rimettersi in coda per accedere al semaforo.

Per far si che durante la partita non si verifichino situazioni anomale, come ad esempio che la maggior parte degli eventi siano infortuni, abbiamo scelto di implementare una studio probabilistico, assegnando a tutte le possibili azioni una determinata probabilità di avvenimento:

TIRO -> 30% – DRIBBLING -> 65% – INFORTUNIO -> 5%

Solamente per i primi due eventi sopra citati, è presente un' ulteriore probabilità, cioè abbiamo il 60% di possibilità che un dribbling vada a buon fine e il 33% che da un tiro scaturisca un goal; inoltre, se il dribbling ha successo la probabilità che il conseguente tiro vada a buon fine aumenta e diventa del 40%.
La partita termina dopo 180 secondi ed un recupero che permette a tutti i thread che si erano messi in coda per il semaforo, prima della fine della durata della partita, di accedere ad esso, eseguire un' azione e poi terminare.

2.2.6 Realizzazione output formattato

L'output a riga di comando è stato formattato in modo tale da avere l'indicazione del minuto della partita e delle azioni che i thread svolgono in quel minuto. Per ogni evento svolto da un giocatore viene indicato il suo numero di maglia, il nome e la squadra a cui esso appartiene; inoltre, si riporta l'esito dell'evento, cioè se il tiro è fallito o ha portato ad un goal, in questo caso si riporta poi il risultato aggiornato, se il dribbling è fallito o ha portato ad un tiro, quindi l'esito di quest'ultimo come già descritto, se il giocatore si è infortunato e il minuto della partita a cui ritornerà o se ha terminato la sua partita e per colui che ha commesso il fallo se è stato espulso o il minuto a cui tornerà a giocare. Tutti i messaggi così formattati vengono mostrati dal server e da tutti i client a riga di comando e

sono inseriti anche all'interno del file di log, di cui si parla al punto successivo.

2.2.7 Registrazione della partita in un file creato appositamente

All'inizio di ogni partita viene creato un nuovo file di log il cui nome è la data al momento dell'avvio della partita (es: Thu May 2 12:23:40 2024-logFile.txt) e contiene tutte le informazioni relative allo svolgimento della stessa, precedentemente formattate a riga di comando. Inoltre, conclusa la partita, riporta alcune statistiche, quali: il numero di gol, il numero di dribbling e il numero di tiri falliti in totale.

Sezione 3:

Containerizing (Docker)

Per soddisfare il requisito di dominio della creazione di un'Immagine per la containerizzazione della web-app, utilizziamo Docker solamente sul lato server, per rendere possibile la sua esecuzione su qualsiasi macchina che ha la possibilità di usare Docker.

3.1 Dockerfile

Abbiamo quindi inserito il nostro Dockerfile nella stessa directory del codice e del compilato del server; il file segue questa struttura:

1. **FROM alpine:latest** -> Utilizziamo alpine come base image;
2. **RUN apk add libc6-compat** -> Aggiungiamo la libreria libc6 per permettere l'esecuzione del compilato c;
3. **WORKDIR /app** -> Creiamo una directory nel Container che verrà utilizzata come dir. corrente;
4. **COPY . /app** -> Facciamo una copia della directory in cui si trova il Dockerfile, quindi anche il nostro eseguibile server;
5. **CMD ["./server"]** -> Eseguiamo il nostro server nel Container;
6. **EXPOSE 12345** -> Esponiamo la porta 12345, la stessa che usiamo per il nostro server.