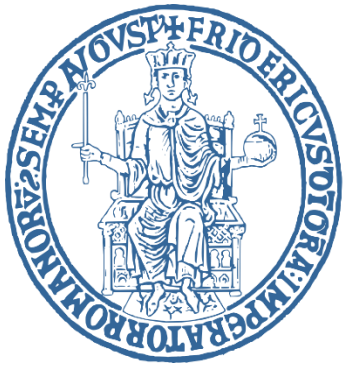
DOCUMENTAZIONE LABORATORIO DI SISTEMI OPERATIVI

Traccia ‘la partita’, nome progetto ‘OneMoreMatch’



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

LG ENGENEERING

Lorenzo Bracale N86003264

Giovanni Bentivoglio N86003364

INTRODUZIONE

Lo scopo di OneMoreMatch è quello della realizzazione di una piattaforma a microservizi attraverso l'uso di strumenti di container, in particolare Docker Compose. Per garantire questa premessa sono state fatte delle scelte significative in termini di separazione dei servizi.

Docker Compose possiede le funzionalità che ci permettono di lanciare i nostri servizi in maniera ordinata e precisa. La creazione dei servizi è semplice quanto l'esecuzione di un programma all'interno della propria macchina. Attraverso il file *compose.yaml* è possibile definire l'ambiente dei servizi, se devono essere connessi ad una rete e quale rete dev'essere, di quante risorse possono usufruire e così via.

Il lato server è stato sviluppato in C all'interno di container con un immagine disponibile su dockerhub che permette l'esecuzione di programmi C grazie alla presenza di un compilatore GCC all'interno di un ambiente Linux.

Il lato client è stato sviluppato in Python per via della sua versatilità e ampio utilizzo. Per eseguire il programma è necessario installare la libreria pillow per python. Con questa libreria, insieme alla libreria tkinter, è stato possibile creare un'interfaccia grafica semplice ma efficace allo scopo dell'applicazione.

Lo sviluppo del sistema si basa sulla simulazione di una partita di calcetto tra due squadre di **cinque giocatori** e **un arbitro** che, a seguito di particolari eventi, inserisce i dettagli degli eventi all'interno di un file di log. Si è pensato, dunque, di gestire queste dinamiche attraverso l'uso di procedure concorrenti sia lato client che lato server. Ogni *attore*, infatti, agisce indipendentemente sebbene debba essere sincronizzato con il resto degli attori e con il resto del sistema.

Il modello sviluppato garantisce l'esecuzione di molteplici partite in contemporanea, trascorse le quali il server attenderà nuovi attori (giocatori e referee). Il referee genererà per ogni partita dei file con statistiche ed eventi.

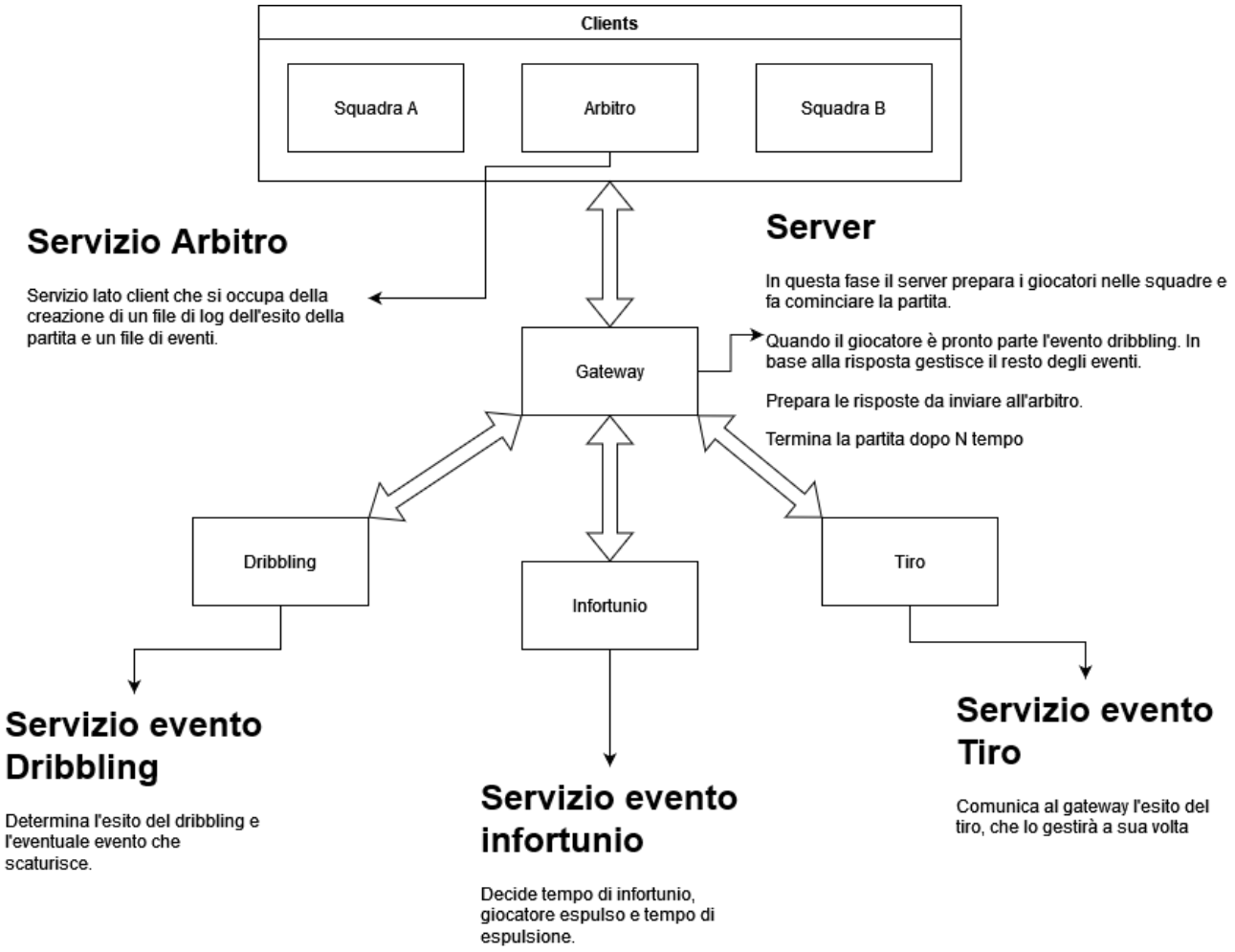
Parlando del sistema, ogni servizio serve per garantire il corretto funzionamento e la separazione delle funzionalità per quanto riguarda gli eventi che i **giocatori** possono effettuare.

L'evento principale è quello del dribbling, attraverso il quale vengono scaturiti il resto degli eventi. Maggiori dettagli verranno presentati nelle successive pagine della documentazione.

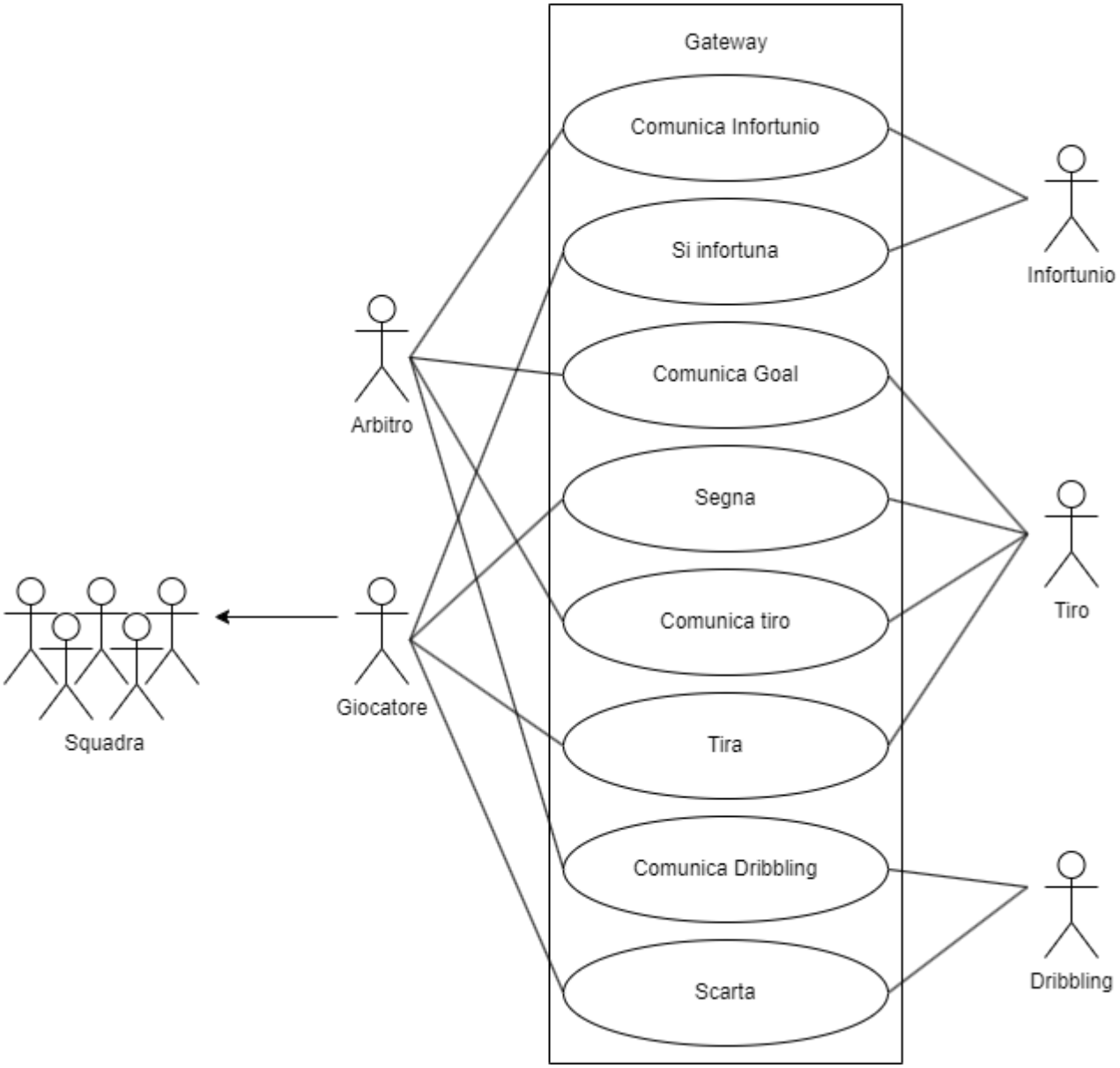
Il tempo per lo svolgimento della partita non è determinato da tempistiche fisiche ma si è deciso di considerare un ***quanto di tempo*** come un *evento* che l'arbitro registra. Ad ogni evento, dunque, viene decrementata la variabile la quale, attraverso un semplice decremento fino a zero, determina la fine della partita.

PROGETTAZIONE

La fase di progettazione è iniziata con l'analisi delle **funzionalità** dell'applicativo che volevamo sviluppare e richieste dalla traccia, per fare ciò abbiamo costruito questo diagramma che ci aiutasse ad individuare i **Clients**, che si sarebbero interfacciati con l'applicativo, divisi in due squadre e l'arbitro che gestisce il decorso della partita. Il diagramma descrive anche un punto centrale da dove sviluppare le funzionalità e le loro interazioni, ossia il **Gateway**, che dopo aver creato le squadre, tramite l'evento **dribbling**, riuscisse a garantire un susseguirsi di eventi i cui risultati fossero determinati casualmente da quest'ultimi fino al termine della partita, ossia un numero finito di azioni.



Successivamente abbiamo voluto anche formalizzare il nostro diagramma fornendo una descrizione più accurata dei casi d'uso, Tramite l'utilizzo del seguente Use Case Diagram dove i servizi sono rappresentati dagli eventi stessi e i richiedenti dei servizi sono I singoli giocatori delle due squadre e l'arbitro che, data la diversa natura basata su esigenze diverse, sfruttava i medesimi servizi per ottenere e comunicare i risultati degli eventi generati


Gateway
Comunica Infortunio
Si infortuna
Comunica Goal
Segna
Comunica tiro
Tira
Comunica Dribbling
Scarta
Arbitro
Giocatore
Squadra
Infortunio
Tiro
Dribbling

SVILUPPO: LATO SERVER

DOCKER COMPOSE

Per garantire il corretto funzionamento dell'architettura completa abbiamo deciso di coordinare i servizi all'interno di container che vengono avviati in modo coordinato attraverso l'uso di docker compose.

All'interno del file compose.yaml abbiamo descritto il comportamento dei container, il loro contenuto e la rete locale con driver bridge a cui sono connessi. Attraverso questa è possibile, infatti, dare un nome di dominio ai container così da poterli raggiungere all'interno delle applicazioni al loro interno in maniera dinamica a prescindere dal loro indirizzo ip.

```yaml
networks:
    omm:
        name: omm
        driver: bridge


services:

    gateway:
        hostname: gateway
        deploy:
          resources:
            limits:
              cpus: '0.75'
              memory: 50M
        tty: true

        build:
            context: ./gateway/
        image: gcc:latest
        volumes:
            - ./gateway:/gateway
        container_name: gateway
        command: bin/bash /gateway/start.sh
        depends_on:
            - dribbling
            - infortunio
            - tiro
        ports:
            - 8080:8080
        networks:
            - omm
```

La limitazione sulla CPU imposta a gateway è stata fatta per alleggerire il carico della macchina, soprattutto per le fasi di debug. Dato lo scope limitato del progetto e l'assenza di una necessità relativa alle prestazioni, abbiamo deciso di lasciare questa limitazione.

I volumi sono le cartelle stesse in cui si trovano i programmi, in questo modo docker non genera delle immagini ad hoc per il nostro programma ma attraverso l'immagine contenente gcc in ambiente linux è in grado di compilare il codice ed eseguirlo. I volumi servono anche nello scopo di uno sviluppo rapido dell'applicazione in quanto i cambiamenti del codice vengono considerati all'apertura del container e non alla generazione dell'immagine.

L'utilizzo di command alla definizione dei container permette l'utilizzo di comandi di bash automatici. Usiamo questa funzionalità per compilare e lanciare i nostri programmi.

GATEWAY

Il punto focale del lato server è rappresentato dal gateway, esso si occupa infatti di comunicare sia con il client che con i vari servizi implementati. Viene lanciato un programma in C all'interno di un container che svolge la maggior parte delle operazioni relative al funzionamento del sistema.

Per ogni processo partita viene generato un processo: il processo figlio è l'arbitro e il padre è per i giocatori. Abbiamo deciso di creare due processi attraverso la procedura fork per assicurarci che lo sviluppo delle funzionalità dell'arbitro avvenga in concomitanza con le funzionalità dei giocatori così da velocizzare lo sviluppo. Vengono usate due pipe, una per comunicare gli eventi all'arbitro e l'altra pipe per permettere un attesa passiva al giocatore in attesa che l'evento venga gestito dall'arbitro.

```
//creato il processo dell'arbitro, gestisce la comunicazione col client.
if ((refPid = fork()) == 0) {
	close(pipe_fd[1]);
	close(event_pipe[0]);
	for (int k = 0; k < 10; k++) {
		printf("referee got player: %c%d\n", squadre[k], k);
	}

	refereeProcess(&refereeSocket);
}
printf("main: referee started\n");
close(pipe_fd[0]);
close(event_pipe[1]);
```

Il main() si occupa dell'inizializzazione delle variabili condivise, dei semafori, dei mutex, delle pipe e delle procedure concorrenti.

```
//inizializzazione delle variabili e mutex in memoria condivisa
N = (int*)mmap(
	NULL, sizeof(int), PROT_READ | PROT_WRITE,
	MAP_SHARED | MAP_ANONYMOUS, -1, 0);

tempoFallo = (int*)mmap(
	NULL, TEAMSIZE*sizeof(int), PROT_READ | PROT_WRITE,
	MAP_SHARED | MAP_ANONYMOUS, -1, 0);

tempoInfortunio = (int*)mmap(
```

```
        NULL, TEAMSIZE*sizeof(int), PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS, -1, 0);

globalVar = (pthread_mutex_t*)mmap(
        NULL, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS, -1, 0);

for (int i = 0; i < TEAMSIZE; i++) {
        tempoFallo[i] = -1;
        tempoInfortunio[i] = -1;
}

*N = 60;

// resto del codice per l'inizializzazione di tempoFallo e tempoInfortunio in gateway.c
```

Ogni giocatore è un thread che attende sulla risorsa "pallone" rappresentata da un mutex. L'arbitro è un processo che, una volta ricevuto un evento che un giocatore ha generato, inizializza un thread per informare il client dell'avvenuto.

Data la mole di thread e i due processi, vengono utilizzati in concomitanza mutex, pipe e semafori, alcuni di essi sono inizializzati all'interno della memoria condivisa tra i due processi. Lo scopo dei mutex è quello di evitare race conditions sulle variabili che vengono condivise tra i vari thread e processi. Lo scopo dei semafori e quello di sincronizzare i thread e i processi in modo tale che determinate procedure vengano svolte in ordine. Alle pipe abbiamo dato un ruolo simile con l'aggiunta di comunicare all'arbitro gli eventi fatti dal giocatore.

I servizi sono indipendenti dallo stato del client, quindi, indipendentemente dalla partita forniscono un risultato coerente. Il controllo del risultato ed eventuale ritrasmissione viene effettuata dal giocatore in gateway.

I SERVIZI: DRIBBLING, INFORTUNIO, TIRO

Lo sviluppo dei servizi si basa sull'idea che in qualsiasi momento un giocatore può compiere un evento e, in base a quale servizio riceve questa richiesta, sappiamo quale evento sta cercando di generare. Ogni servizio attende immediatamente l'azione di un giocatore qualsiasi. Ogni servizio invia una risposta di tipo diverso ai giocatori: (esempio di Dribbling)

```
while (1) {
        printf("main: waiting for player...\n");
        client = accept(serverSocket, (struct sockaddr*)&clientAddr, &len);
        printf("main: connection accepted!\n");
        if (fork() == 0) service((void*)&client), exit(0);
        printf("main: thread creato!\n");
}
```

- Dribbling: riceve in ingresso il giocatore in possesso della palla e decide un giocatore designandolo come avversario, a quel punto determina in maniera casuale l'esito del dribbling oppure se si verifica un infortunio, quest'ultimo non potrà gestirlo.
- Infortunio: quando un giocatore riceve da dribbling un messaggio di infortunio comunica a questo servizio l'accaduto e attende da esso i dettagli dell'infortunio, in particolare la durata di quest'ultimo e il tempo di fallo del giocatore che ha causato l'infortunio. La durata dell'infortunio viene determinata casualmente mentre quella di fallo è la metà della prima così da simulare una gravità del fallo proporzionale all'infortunio causato.
- Tiro: riceve in ingresso il giocatore che intende effettuare il tiro, questo servizio molto semplice comunica l'esito del tiro al giocatore.

Oltre a tiro anche Dribbling e Infortunio comunicano al giocatore l'evento e i suoi dettagli.

La gestione della concorrenza degli eventi è già gestita nel gateway. Se ci sono eventi che avvengono nello stesso momento, allora sono di partite diverse

Per riassumere il lavoro dei servizi: previa inizializzazione attendono un evento generato da un giocatore e, dopo averlo gestito, comunicano l'esito a quest'ultimo.

SVILUPPO: LATO CLIENT

Il nostro client sviluppato in **python** funge principalmente come un'interfaccia **rapida** ed **intuitiva** tra l'**utente** e il **backend**.

Possono essere lanciati otto client che si comportano da giocatori che cercano di accedere alla partita mandando una richiesta al server, ad ogni client viene chiesto il nome del giocatore, e riceve dal server il numero della partita di appartenenza.

Inizialmente l'utente si troverà tre finestre davanti, le prime due ai lati serviranno per **formare le due diverse squadre** i cui rispettivi capitani verranno generati **casualmente** tra tutti i giocatori disponibili all'avvio dell'applicativo, successivamente, dopo aver caricato i nuovi giocatori, l'utente potrà decidere le squadre di appartenenza dei **restati otto giocatori** ricevuti tramite server cliccando sugli appositi button e confermando infine le due formazioni.

Fatto ciò, l'utente potrà visualizzare nella finestra centrale sotto il logo dell'applicazioni sia il futuro **punteggio della partita** sia un button per iniziare la simulazione.

In qualsiasi momento della simulazione l'utente avrà la possibilità di decidere la **velocità di simulazione** tra due diverse modalità, una **rapida** di default, e una più **lenta**, adatta per dare una sensazione di simulazione più user friendly.

Avviata la partita l'utente potrà visualizzare in **tempo reale** la modifica dei punteggi e le **ultime quattro azioni della partita** nella finestra centrale, al termine della partita verrà aperta una message box che informerà l'utente del termine della simulazione e lo informerà della generazione di **due file di log**, uno per visualizzare gli eventi della partita e uno per le statistiche generali come tiri falliti, dribbling avvenuti con successo, etc..

Se durante l'esecuzione dell'applicativo l'utente avrà **problemi di collegamento** ai servizi verrà aperta una message box che chiederà all'utente di riavviare l'applicativo descrivendo la problematica presentata.

Un aspetto importante dell'applicazione è stato decidere come rappresentare le diverse tipologie di **client**, ossia i giocatori e l'arbitro, la nostra scelta è stata rappresentare l'arbitro e i due capitani come **thread**, i due capitani inviano le informazioni delle squadre al server previa pressione del bottone, l'arbitro conferma l'inizio della partita al server e insieme ad esso permette la continua comunicazione nel client e nei due log esterni.

```
def captainThread(team,sock):
    with mutex:
        for player in team:
```

```
            if team == teamA:
                associa_squadra(int(player),'A')
                comando = f"A{int(player)}"
                comando += '\0'
                sock.send(comando.encode())
                sock.recv(1024)
            else:
                associa_squadra(int(player),'B')
                comando = f"B{int(player)}"
                comando += '\0'
                sock.send(comando.encode())
                sock.recv(1024)
```

Il thread dell'arbitro inserisce gli eventi in una coda e verranno gestiti da un altro thread.

```
def refereeThread(conn, msgQueue):
    global errore
    try:
        s = socket.socket()
        s.connect(conn)
        comando = "sono l'arbitro"
        comando += "\0"
        s.send(comando.encode())
    except socket.error as err:
        print(f"qualcosa e' andato storto err: {err}, sto uscendo... \n")
        with mutex:
            if not errore:
                errore = True
                window.event_generate('<<error_event>>')

    stop = True
    while stop:
        try:
            # Riceve i dati dal server
            data = s.recv(2048)
            msgQueue.put(data);
            ack = "ack\0"
            s.send(ack.encode())
            if not data:
                # Il server ha chiuso la connessione
                print("Connessione chiusa dal server. Partita terminata?")
                msgQueue.put(b"partitaTerminata")
                break
        except Exception as e:
            print("Errore nella ricezione dei dati:", e)
            break
```

Per lo sviluppo dell'interfaccia grafica ci siamo serviti della libreria standard **Tkinter** che abbiamo utilizzato per implementare i diversi elementi dell'interfaccia (labels, buttons, checkbox, etc…) e **Pillow** che invece abbiamo sfruttato per poter inserire immagini nel nostro applicativo.