



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

Dipartimento di
Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di laurea in
Ingegneria Informatica

Classe n. L-8

Progettazione e Sviluppo di una Serra Automatizzata con Raspberry

Candidati:
Matteo Carminati

Lorenzo Torri

Matricola n.
1066354
1069047

Relatore:
Chiaramo Prof. Davide Brugali

Anno Accademico
2021/2022

INDICE

1. INTRODUZIONE	4
2. ARCHITETTURA DEL PROGETTO	6
TIPOLOGIA DEI DATI	6
FORMATO DI INTERSCAMBIO DEI DATI	7
CHIAMATA HTTP	8
3. EMBEDDED SYSTEM	11
SCHEDA RASPBERRY PI 3 MODEL B	11
SENSORI	13
DHT22	13
CAPACITIVE SOIL MOISTURE SENSOR	16
HC-SR04	21
SCHEDA RELE' E ATTUATORI	26
UML STATE CHART DIAGRAM	32
SCHEDULAZIONE THREAD	38
SPIEGAZIONE CODICE	41
IMMAGINI SERRA	55
4. TECNOLOGIA CLOUD	60
5. APPLICATIVO FLUTTER	66
6. BIBLIOGRAFIA E SITOGRAFIA	72
7. RINGRAZIAMENTI	74

1. INTRODUZIONE

Lo scopo del lavoro di tesi è quello di progettare e successivamente sviluppare una serra completamente automatizzata, che richieda solo qualche piccolo controllo da parte dell'utente.

Seguendo le lezioni del corso di Embedded And Real Time Systems (21038-ENG) del Corso di Laurea di Ingegneria Informatica del Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione, i due candidati hanno acquisito nozioni riguardanti la progettazione di sistemi embedded ed è in questo contesto che è nato l'interesse nel voler provare a svilupparne uno.

Il progetto di tesi proposto è quindi un possibile prototipo di una serra automatizzata, realizzata attraverso il controllore Raspberry e una serie di sensori e di attuatori necessari per poter implementare le funzioni di misura e di controllo dei parametri ambientali.

Gli strumenti tecnici impiegati si discostano da quelli professionali realmente utilizzati per realizzare progetti di questo tipo. Per questo motivo, il progetto presentato è solo un possibile prototipo, sviluppato con un budget contenuto e con componenti elettronici acquistati presso l'azienda di spedizioni online Amazon.com.

Inoltre, per rendere più semplice il controllo e la visualizzazione dei dati processati da Raspberry, il progetto del sistema Embedded è stato integrato con un'architettura Cloud e un'applicazione mobile ad essa collegata, la quale permette all'utente utilizzatore di consultare lo stato del sistema.

La prima è stata creata mediante i servizi di Amazon Web Services (AWS), per poter archiviare in remoto le informazioni della serra; la seconda, invece, è stata sviluppata sfruttando Flutter (framework open source di proprietà di Google), attraverso il linguaggio di programmazione Dart.

In particolare, con lo sviluppo dell'architettura Cloud e dell'applicazione mobile, i due candidati hanno integrato al progetto di tesi alcune conoscenze acquisite

durante il corso di Tecnologie Cloud & Mobile (21059) del Corso di Laurea di Ingegneria Informatica del Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione.

All'interno della relazione di tesi, verrà mostrata dapprima l'architettura generale del progetto. In questo modo è possibile osservare in maniera più approfondita come i componenti utilizzati interagiscano tra di loro.

In seguito, si andranno a ripercorrere le fasi di progettazione e, successivamente, quelle che hanno portato all'effettiva costruzione della serra, con un'analisi ulteriore delle attività di controllo effettuate da Raspberry.

In conclusione, verrà inoltre mostrato nel dettaglio il funzionamento sia del sistema Embedded che dell'applicativo comunicante con l'architettura Cloud.

2. ARCHITETTURA DEL PROGETTO

Come già anticipato nell'introduzione, il progetto di tesi è composto principalmente da tre componenti:

- il sistema Embedded
- l'architettura Cloud
- l'applicazione Mobile

In particolare, il primo componente è interamente dedicato al controllo di parametri ambientali e al funzionamento della serra stessa; il secondo si occupa di salvare nel Cloud i dati processati dalla scheda Raspberry; il terzo serve a visualizzare ciò che viene salvato in remoto, riuscendo così a monitorare il sistema Embedded.

Di seguito, verrà presentata la modalità con la quale i vari componenti comunicano tra di loro per poi, nei successivi paragrafi, andare ad analizzare più in dettaglio ogni singolo componente, approfondendo il funzionamento di ciascuno.

2.1. TIPOLOGIA DEI DATI

Il sistema Embedded produce ed elabora una serie di dati. All'interno della serra sono infatti presenti tre diversi sensori (che verranno spiegati in dettaglio nel paragrafo dedicato allo studio del sistema Embedded), i quali inviano i dati delle misurazioni al controllore del sistema, ovvero alla scheda Raspberry.

Questi rilevamenti riguardano:

- Temperatura dell'aria all'interno della serra (°C)
- Umidità dell'aria all'interno della serra (%)
- Umidità del terreno (%)
- Percentuale di riempimento di acqua della tanica di irrigazione (%)

Inoltre, è stato scritto del codice per raccogliere ulteriori informazioni importanti per il sistema Embedded, come ad esempio lo stato di ogni singolo attuatore (acceso o spento) e l'orario a cui risalgono i dati registrati.

Queste informazioni possono essere visualizzate attraverso un monitor collegato a Raspberry e con vari comandi di “print()” inseriti all’interno del codice. Tuttavia, per rendere il progetto il più realistico possibile, si è scelto di sviluppare un’applicazione mobile, la quale permette di avere una visualizzazione più immediata ed efficace di tutti questi parametri.

Per realizzare la connessione tra Raspberry e applicativo mobile, è stata implementata un’architettura Cloud, sviluppata attraverso i servizi offerti da Amazon Web Services. In particolare, Raspberry carica i dati nell’architettura; questi, in seguito, possono essere scaricati dall’applicazione ogni qual volta un utente lo richieda.

2.2. FORMATO DI INTERSCAMBIO DEI DATI

Quando si parla di scambio di dati è necessario specificare lo standard che si utilizza, per permettere il passaggio di informazioni tra client e server.

Nel progetto, la scheda Raspberry e l’utente che utilizza l’applicazione svolgono il ruolo di client, mentre la tecnologia Amazon sfruttata per lo storage dei dati riveste il ruolo di server.

In particolare, come standard per l’interscambio di dati, si è optato per l’utilizzo di JavaScript Object Notation (JSON).

Questa scelta è giustificata dal fatto che sia Python (linguaggio di programmazione utilizzato per controllare la scheda Raspberry e per programmare alcuni componenti dell’architettura Cloud), sia Dart (linguaggio di programmazione utilizzato per sviluppare l’applicazione mobile) hanno elementi comuni a JavaScript e hanno inoltre strutture predisposte per lavorare con documenti in formato JSON.

Di conseguenza, all'interno del codice di controllo di Raspberry, i dati processati vengono formattati secondo le regole dello standard JSON.

Di seguito si riporta un esempio [Fig.1]:

```
1. {
2.     "dati": [
3.         {
4.             "Media temperatura aria": 23,
5.             "Media umidita aria": 30,
6.             "Media umidita suolo": 45,
7.             "Livello acqua": 20,
8.             "Orario": "14:05"
9.         },
10.        {
11.            "Pompa irrigazione": 0,
12.            "Pompa umidita": 0,
13.            "Pompa fertilizzante": 0,
14.            "Ventola": 1
15.        }
16.    ]
17. }
```

Esempio formato dati JSON [Fig.1]

2.3. CHIAMATA HTTP

Il file, osservato precedentemente, viene salvato con un'estensione “.json” e viene caricato sull'architettura Cloud attraverso una chiamata Http di tipo POST all'API Gateway del server.

Si sfrutta la libreria requests di Python per simulare la chiamata avente come corpo il file contenente i dati [Fig.2].

```
import requests
1. URL = "https://a9qj196ajf.execute-api.us-east-1.amazonaws.com"
2.
3. #funzione per inviare i dati al cloud
4. def chiamata_Http():
5.     #il body della chiamata POST è in JSON
6.     contenuto = figli.get_values()
7.     r = requests.post(URL+="/upload", json = contenuto)
8.
```

Codice chiamata HTTP [Fig.2]

In particolare, si specifica che il contenuto del corpo della chiamata è ottenuto mediante la funzione “get_values()” [Fig.3], la quale permette di ottenere il valore di tensione dei pin di input della scheda Raspberry per conoscere lo stato degli attuatori e permette di formattare correttamente tutti i dati osservati nel momento in cui la funzione viene chiamata.

```

1. #creazione file json con tutti i valori utili
2. def get_values():
3.     GPIO.setup(GPIO_pompa_irrigazione, GPIO.IN)
4.     GPIO.setup(GPIO_pompa_umidificazione, GPIO.IN)
5.     GPIO.setup(GPIO_pompa_fertilizzante, GPIO.IN)
6.     GPIO.setup(GPIO_ventola, GPIO.IN)
7.
8.     # Ritorna 1 se OFF o 0 se ON
9.     pompa_irrigazione = GPIO.input(GPIO_pompa_irrigazione)
10.    pompa_umidita = GPIO.input(GPIO_pompa_umidificazione)
11.    pompa_fertilizzante = GPIO.input(GPIO_pompa_fertilizzante)
12.    ventola = GPIO.input(GPIO_ventola)
13.
14.    #otteniamo l'ora
15.    now = datetime.now()
16.    ora = str(now.hour) + ":" + str(now.minute)
17.
18.    ampiezza = altezzariferimento_vuoto - altezza_vuoto
19.    percentuale = int(ampiezza * 100 / altezzariferimento_vuoto)
20.
21.    #formattazione JSON
22.    contenuto = {
23.        "dati": [
24.            {
25.                "Media temperatura aria": int(M_temperatura_aria),
26.                "Media umidita aria": int(M_umidita_aria),
27.                "Media umidita suolo": int(M_umidita_suolo),
28.                "#Lista umidita aria": list(lista_valori_dht22_umidita.queue),
29.                "Livello acqua": percentuale,
30.                "Orario": ora
31.            },
32.            {
33.                "Pompa irrigazione": pompa_irrigazione,
34.                "Pompa umidita": pompa_umidita,
35.                "Pompa fertilizzante": pompa_fertilizzante,
36.                "Ventola": ventola
37.            }
38.        ]
39.    }
40.    return contenuto
41.

```

Codice funzione “get_values()”[Fig.3]

Si precisa, inoltre, che l'invio dei dati, effettuato dalla scheda Raspberry sull'architettura cloud, è dilazionato nel tempo. In questo modo, si evita di appesantire inutilmente il sistema con una quantità eccessiva di dati superflui.

3. EMBEDDED SYSTEM

Con il termine Embedded System, secondo la definizione fornita dall’Ingegnere informatico Marylin Wolf, si intende “*ogni sorta di dispositivo che include un computer programmabile non destinato ad essere un computer general purpose*” (Marylin Wolf).

Il caso in esame tratta un Embedded System costituito da una scheda Raspberry Pi 3 Model B utilizzata come controllore e da una serie di sensori ed attuatori volti rispettivamente alla misurazione e al controllo dei parametri di interesse della serra.

Durante le fasi di progettazione e sviluppo del sistema Embedded, è stato innanzitutto necessario settare la scheda Raspberry, per consentire un corretto collegamento con i vari componenti. Inoltre, è stata eseguita una calibrazione dei sensori, in modo da ottenere misurazioni valide. Infine, è stata pensata e sviluppata tramite codice Python una schedulazione dei thread, la quale è stata successivamente caricata sulla scheda Raspberry in modo da controllare ogni singolo componente della serra secondo le necessità individuate.

In questo paragrafo, dopo una breve introduzione alla scheda Raspberry Pi 3 Model B, verranno quindi analizzati i componenti che costituiscono il sistema Embedded, per poi osservare in dettaglio il funzionamento della serra anche tramite l’utilizzo di diagrammi UML esplicativi appositamente realizzati.

In conclusione, tramite porzioni di codice sviluppate dai due candidati, verrà approfondita in dettaglio la schedulazione dei compiti attuata da Raspberry.

3.1. SCHEDA RASPBERRY PI 3 MODEL B

Raspberry Pi è un computer a scheda singola sviluppato dall’omonima compagnia britannica Raspberry Pi Foundation. Come si può leggere dal titolo della homepage del progetto originale “*Our mission is to put the power of computing and digital making into the hands of people all over the world*”

(Raspberry Pi Foundation Strategy, 2016/2018), appare chiaro lo scopo principale che ha portato alla nascita di questo prodotto.

Raspberry è di fatto un computer che supporta sistemi operativi basati su Kernel Linux. Si differenzia da molte alternative proposte dal mercato poiché, trattandosi di un vero e proprio “mini computer”, può essere impiegato come sistema autonomo, a differenza di hardware come Arduino, che sono dei semplici microcontrollori.

Inoltre è una scheda non dotata di memoria interna, ma si basa su un dispositivo di memoria esterna, ovvero una scheda SD. Di conseguenza, anche il sistema operativo utilizzato da Raspberry deve essere necessariamente installato su tale dispositivo di archiviazione.

I due candidati, per poter lavorare con il controllore, hanno deciso di optare per il sistema operativo sviluppato dalla stessa compagnia produttrice della scheda, ovvero il sistema operativo Raspberry Pi OS, precedentemente noto come Raspbian.

Una funzionalità di notevole importanza sfruttata per la creazione del progetto nella sua interezza, e in particolare per permettere di osservare i dati processati da Raspberry anche a distanza, è stata la possibilità di connettere con facilità la scheda alla rete Internet. Tramite questa connessione, infatti, è stato possibile inserire nel codice di controllo delle chiamate Http al server Cloud, riuscendo così a salvare le varie misurazioni in un database remoto.

Grazie a questa funzionalità della scheda, è stato possibile completare il progetto, arricchendolo con l’architettura Cloud e l’applicazione Mobile.

Il linguaggio di programmazione scelto per lo sviluppo del codice volto al controllo del sistema embedded è stato Python. La motivazione è legata al fatto che sono presenti numerose librerie, sviluppate per utilizzare al meglio i sensori introdotti nel progetto di tesi. Queste introducono un livello di astrazione più alto e semplificano notevolmente l’attività di programmazione.

Tali librerie, e i loro autori, sono state riportate all'interno dei riferimenti bibliografici presenti al termine della tesi.

3.2. SENSORI

Di seguito si analizzano i vari sensori impiegati per il progetto di tesi, con le rispettive porzioni di codice utilizzate per poter ottenere le misurazioni da ognuno.

3.2.1. DHT22

Il DHT22 è un sensore che permette di misurare la temperatura dell'aria e l'umidità relativa.

In particolare, l'umidità relativa è così definita: “*è un indice della quantità di vapore contenuto in una miscela aeriforme. E' definita come il rapporto della densità del vapore contenuto nel miscuglio e la densità del vapore saturo alla temperatura della miscela*”(Wikipedia). Essendo quindi un rapporto, viene spesso rappresentata attraverso una percentuale.

Questi due parametri sono fondamentali per il corretto monitoraggio della serra:

- la temperatura è infatti importante per sapere se, in base alla tipologia di pianta che si sta coltivando, all'interno della serra ci sia troppo caldo o freddo;
- l'umidità relativa è anch'essa importante da monitorare; piante diverse infatti necessitano di livelli di umidità differenti per potersi sviluppare al meglio.

Per queste motivazioni, si è deciso di inserire nella serra un sensore che permettesse di monitorare tali parametri. La scelta è quindi ricaduta sul sensore DHT22 per via delle sue dimensioni contenute e per la facilità con cui può essere trovato sul mercato a un prezzo vantaggioso.

Il DHT22 riesce a misurare la temperatura all'interno della serra grazie al sensore di temperatura NTC (termistore).

Per quanto riguarda invece l'umidità relativa, la presenza del vapore acqueo viene misurata osservando la differenza di resistenza elettrica prodotta da due elettrodi; questi lavorano in concomitanza a un terzo elemento, il substrato, il quale è in grado di trattenere l'umidità. Il fenomeno fisico che permette di leggere il valore di umidità da parte del sensore è il rilascio di ioni da parte del substrato che ha assorbito l'umidità. Questi ultimi, avendo una loro carica elettrica, vanno ad alterare la resistenza elettrica prodotta dai due elettrodi.

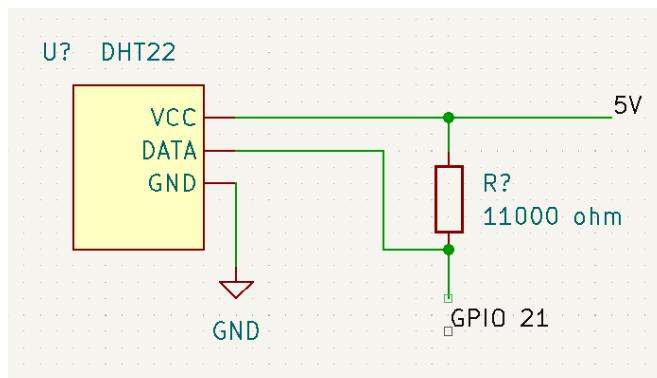
La differenza tra il valore finale di resistenza e il valore iniziale è inversamente proporzionale all'umidità relativa presente all'interno della serra. Il passaggio da un delta di potenziale al valore di umidità relativa è controllato dal sensore stesso, attraverso un circuito integrato.

Il sensore DHT22 è dotato di 4 pin; tuttavia di questi se ne usano solamente 3:

- un VDD che corrisponde all'alimentazione a 5 V;
- un GND che corrisponde al ground;
- un pin di segnale, ovvero il pin di output che genera le misurazioni.

L'ultimo pin necessita di un voltaggio pari a 3.3 V, per cui è stato necessario introdurre una resistenza di un valore pari a $11\text{ k}\Omega$ tra l'alimentazione di 5 V fornita dalla scheda Raspberry e il pin di segnale.

Di seguito si riporta lo schema circuitale del sensore [Fig.4]:



Schema circuitale DHT22 [Fig.4]

Come anticipato in precedenza, per poter leggere i valori dei sensori sono state impiegate delle particolari librerie sviluppate da altri programmati. Queste

hanno quindi permesso di alzare il livello di astrazione e di poter gestire con maggiore facilità i componenti senza dover lavorare al Physical Layer dell'architettura ISO/OSI.

Per il DHT22 è stata sfruttata la libreria sviluppata da adafruit, il cui codice sorgente è presente nella bibliografia del progetto.

Quindi, dopo aver installato correttamente la libreria su Raspberry, i due candidati hanno sviluppato un codice [*Fig.5*], per poter ottenere dal sensore le due misurazioni necessarie.

```
1. # import libreria per dht22
2. import Adafruit_DHT as dht
3.
4. #codice per lettura valori del sensore DHT22
5. def dht22():
6.     DHT = 21
7.     h,t = dht.read_retry(dht.DHT22, DHT)
8.     return h,t
9.
```

Codice misurazioni dht22 [Fig.5]

Si osserva quindi alla riga 2 il comando necessario per importare la libreria nel codice Python. Successivamente, all'interno della funzione, viene utilizzato il modulo dht per richiamare la funzione necessaria per la lettura del pin di segnale del DHT22. Infatti vengono specificati come parametri della funzione stessa la tipologia di sensore con cui si sta lavorando e il numero di porta (GPIO21), che è collegata al pin di segnale del sensore.

Questo permette quindi di estrarre i valori dell'umidità relativa e della temperatura dell'aria, che saranno utili per monitorare la serra.

3.2.2. CAPACITIVE SOIL MOISTURE SENSOR

Il Capacitive Soil Moisture Sensor è un sensore capacitivo che permette di misurare il livello di umidità del suolo. Sulla base del valore registrato, è dunque

possibile monitorare al meglio il sistema di irrigazione, scegliendo la quantità di acqua da fornire alla pianta coltivata.

In realtà, il calcolo dell'umidità del suolo è molto complicato; esso infatti dipende da innumerevoli fattori. Nel progetto di tesi presentato, quindi, quando si parlerà di umidità del suolo, si intende più nello specifico la percentuale di acqua presente all'interno del terreno monitorato. Tale valore, però, non può essere considerato universalmente corretto. Ad esempio, ottenere una misurazione pari a 0% tramite il Capacitive Soil Moisture Sensor non indica necessariamente che il terreno sia completamente privo di acqua.

Nonostante queste approssimazioni, il sensore, per il progetto presentato, rimane comunque una buona soluzione, che permette di avere un'idea generale della quantità di acqua presente nel suolo in esame.

Come per il DHT22, anche il Capacitive Soil Moisture Sensor sfrutta la variazione di potenziale generata dagli ioni per calcolare la percentuale di umidità.

In particolare, come suggerisce il nome del sensore, è presente un condensatore con i piatti negativo e positivo e tra di essi è presente il dielettrico. Il sensore misura la variazione della capacità del condensatore dovuta a cambiamenti nati nel dielettrico. L'acqua nel terreno, con i suoi ioni, va infatti a modificare il dielettrico, portando così ad un cambiamento della capacità del sensore stesso.

La variazione di capacità che si genera all'interno del sensore viene letta attraverso il circuito integrato 555, il quale genera in uscita un segnale continuo espresso in Volt, proporzionale alla differenza di capacità registrata.

Il Capacitive Soil Moisture Sensor è quindi un sensore analogico, ovvero, a differenza degli altri sensori utilizzati in questo progetto, genera in uscita un segnale continuo che è proporzionale al misurando.

Tuttavia la scheda Raspberry, a differenza di Arduino ad esempio, non ha pin adeguati per leggere dei segnali di tipo analogico. Per questo motivo, è stato necessario introdurre anche un convertitore di segnale da analogico a digitale.

Questo consente di trasformare il segnale in continua inviato dal sensore in un valore numerico leggibile dalla scheda Raspberry. Successivamente, attraverso una porzione di codice, tale valore può essere convertito nel corrispondente valore percentuale, che indica l'umidità del suolo.

Il convertitore analogico digitale utilizzato è l'ADS1115.

L'ADS1115 offre una precisione di 16-bit con 860 campionamenti al secondo.

La particolarità di questo convertitore è che il segnale che produce in uscita necessita del protocollo I²C.

“I²C è un sistema di comunicazione seriale bifilare utilizzato tra circuiti integrati” (Wikipedia). E’ quindi un protocollo seriale che permette la comunicazione tra alcune tipologie di dispositivi. In particolare, si tratta di un protocollo di tipo master/slave; perciò, all’interno della comunicazione, è sempre presente un dispositivo che ha il ruolo di master, che emette il segnale di clock; tutti gli altri devono sottostare alle regole imposte dal master; dunque si sincronizzano rispetto al segnale, senza poterlo modificare: per questo vengono definiti slave.

Il master ha il compito quindi di controllare il bus I²C che permette la comunicazione tra i vari dispositivi.

Il protocollo necessita inoltre di due linee seriali di comunicazione, ovvero il Serial Data (SDA) per lo scambio di dati e il Serial Clock (SCL) per il segnale di clock citato sopra, indispensabile poiché il protocollo è sincrono e quindi necessita di una sincronizzazione tra i vari dispositivi per assicurare una corretta comunicazione.

La scheda Raspberry è propensa all’utilizzo del protocollo di comunicazione I²C. E’ necessario settare come attiva la modalità di I²C Bus presente nelle impostazioni generali; in questo modo si predispongono al ruolo di SDA e SCL due pin della scheda.

Facendo ciò, è quindi possibile collegare il convertitore analogico digitale e il Capacitive Soil Moisture Sensor, ottenendo così le misurazioni effettuate da quest'ultimo.

In particolare il sensore ha tre pin:

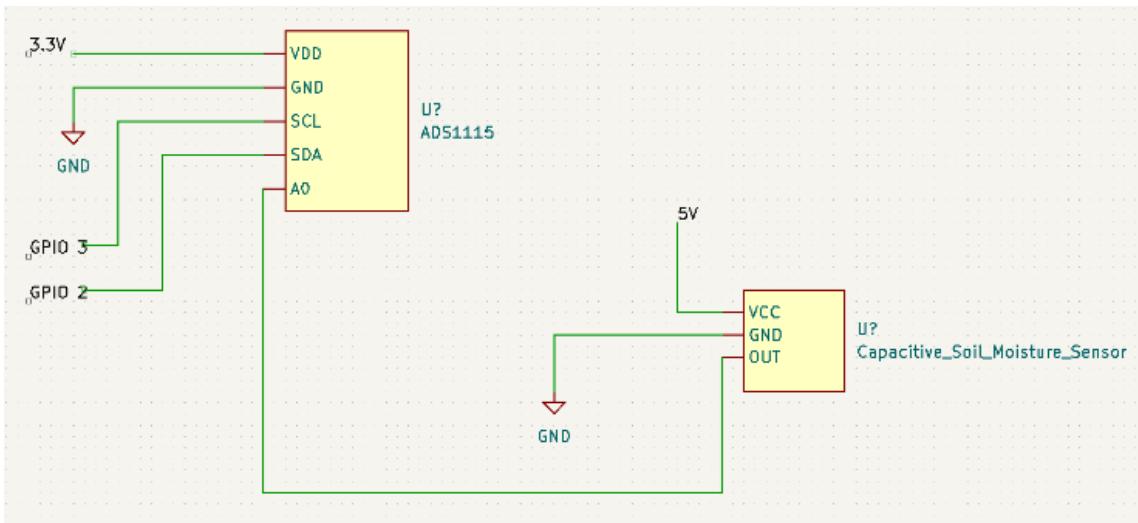
- un VDD che corrisponde all'alimentazione a 5 V;
- un GND che corrisponde al ground;
- un OUT che corrisponde al segnale in uscita;

L'ADS1115 ha invece molteplici pin; per questo progetto ne sono stati utilizzati solamente cinque:

- un VDD che corrisponde all'alimentazione a 3.3 V;
- un GND che corrisponde al ground;
- un SCL che corrisponde, per quanto detto in precedenza, alla linea seriale per lo scambio di dati e che va quindi collegata al pin GPIO03. Quest'ultimo, con la modalità I²C Bus attiva su Raspberry, è settato per lavorare come SCL;
- un SDA che corrisponde, per quanto detto in precedenza, alla linea seriale per il segnale di clock. Analogamente alle considerazioni fatte per SCL, anche questa deve essere collegata al pin GPIO02 della scheda Raspberry;
- un A0 che corrisponde al pin necessario per ottenere in ingresso il segnale analogico da convertire. Questo pin deve essere dunque collegato al pin OUT del Capacitive Soil Moisture Sensor.

Quindi, in conclusione, il sensore rileva la variazione di capacità e produce in uscita un segnale in continua del voltaggio. Tale segnale arriva in ingresso al convertitore analogico digitale tramite il pin A0; successivamente viene convertito in un segnale digitale e tramite il pin SDA viene inviato alla scheda Raspberry mediante il bus di comunicazione I²C.

Di seguito si riporta lo schema circuitale del Capacitive Soil Moisture Sensor abbinato alla scheda ADS1115, indispensabile per la conversione del segnale da analogico a digitale [*Fig.6*].



Schema circuitale Capacitive soil moisture sensor e ADS1115 [Fig.6]

Anche in questo caso sono state utilizzate delle librerie per facilitare l'utilizzo dei vari componenti necessari per ottenere il valore percentuale dell'umidità.

E' stata impiegata una libreria per leggere i valori del convertitore analogico digitale, una per poter utilizzare il bus I²C e l'ultima per utilizzare una nomenclatura particolare dei pin della scheda.

Si riporta di seguito il codice sviluppato [Fig.7] per poter ottenere il valore in percentuale dell'umidità del suolo:

```

1. import board
2. import busio
3. import adafruit_ads1x15.ads1015 as ADS
4. from adafruit_ads1x15.analog_in import AnalogIn
5.
6. # la funzione serve per trasformare il valore letto dal capacitivo in un
    valore in % di umidità
7. def map(x, in_min, in_max, out_min, out_max):
8.     return int((x-in_min) * (out_max - out_min) / (in_max - in_min) +
    out_min)
9.
10.
11.#per lettura valori del Capacitive soil moisture sensor
12.def capacitive():
13.    #attività del capacitivo
14.    i2c = busio.I2C(board.SCL, board.SDA)
15.    #creiamo l'oggetto ads a partire dal bus
16.    ads = ADS.ADS1015(i2c)
17.    #usiamo i 4 canali dell'ads
18.    chan1 = AnalogIn(ads, ADS.P0)
19.    chan2 = AnalogIn(ads, ADS.P1)
20.    chan3 = AnalogIn(ads, ADS.P2)
21.    chan4 = AnalogIn(ads, ADS.P3)
22.
23.    print('CAPACITIVE')
24.    umidita = (map(chan1.voltage, 2.990091, 0.604018, 40, 100))
25.
26.    return umidita
27.

```

Codice misurazioni Capacitive Soil Moisture Sensor [Fig.7]

Dopo aver importato correttamente le varie librerie, si procede alla lettura del codice a partire dalla funzione “capacitive()”.

Si comincia settando il bus I²C per lo scambio dei dati tra Raspberry e convertitore e per il segnale di clock (riga 14).

Successivamente si crea l’oggetto ADS1115 associato al bus di prima (riga 16).

Infine si leggono i valori di input presenti sui 4 diversi canali dell’ads.

Da notare che, per lo scopo del progetto di tesi riportato, per quanto detto sopra, solamente il canale A0 è collegato al sensore, gli altri tre canali rimangono inutilizzati.

Successivamente, alla riga 24, si ottiene il valore dell’umidità, richiamando la funzione di “map()”. Questa funzione permette di poter convertire il valore espresso in tensione e letto da Raspberry in un valore che esprima l’umidità in

percentuale. Per questa operazione è necessaria infatti una formula particolare, dato che umidità del suolo e voltaggio sono inversamente proporzionali.

Quindi, utilizzando due estremi di voltaggio leggibili sul canale (2.99 e 0.6 V), associati ai loro valori in percentuale dell'umidità (40 e 100), si ottiene tale formula, dove la variabile x rappresenta il voltaggio letto da Raspberry che deve essere convertito:

$$\frac{(x-2.99) \cdot (100-40)}{(0.6-3) + 40}$$

In questo modo, si riesce quindi ad ottenere l'umidità del suolo espressa in valore percentuale. Questa è una variabile fondamentale per il controllo della serra, in quanto permette di capire indicativamente quando è richiesta un'irrigazione.

3.2.3. HC-SR04

Per monitorare il livello dell'acqua presente all'interno della tanica impiegata per l'irrigazione è stato utilizzato un sensore ad ultrasuoni HC-SR04.

Il sensore HC-SR04 è costituito da un trasmettitore ad ultrasuoni, un ricevitore e un circuito di controllo.

Il funzionamento del sensore si basa sulla riflessione che le onde ad alta frequenza (gli ultrasuoni), trasmesse dal sensore stesso, subiscono quando intercettano un oggetto o un fluido diverso dall'aria. Le onde emesse dal trasmettitore, dunque, dopo aver incontrato un ostacolo, vengono parzialmente riflesse verso il sensore, per poi essere captate dal ricevitore ed infine processate dal circuito di controllo.

Infatti, conoscendo la velocità di propagazione del suono nell'aria e misurando il tempo impiegato dall'onda riflessa per tornare al sensore, è possibile determinare con sufficiente precisione la distanza che intercorre tra il sensore e l'ostacolo, che

nel caso esaminato corrisponde alla distanza tra il sensore e la superficie dell'acqua presente nella tanica.

In particolare HC-SR04 è composto da 4 differenti pin:

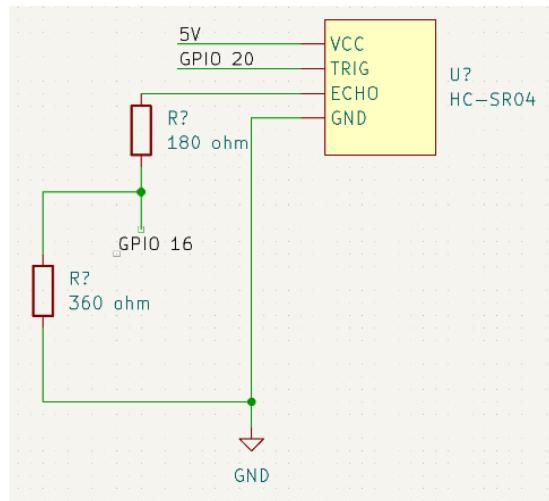
- un VDD che corrisponde all'alimentazione a 5 V;
- un GND che corrisponde al ground;
- un pin Echo Pulse Output (ECHO) che serve per misurare la durata della trasmissione;
- un pin Trigger Pulse Input (TRIG) che triggerà il sensore ad inviare un impulso ad ultrasuoni.

Il sensore, quindi, attraverso il pin di TRIG, invia l'onda ad ultrasuoni e setta a "high", cioè a 5V, il pin di ECHO, il quale manterrà questo valore fino a quando non verrà rilevata l'onda riflessa, istante nel quale ECHO torna ad essere "low", cioè a 0 V.

Con il codice Python, quindi, si può misurare il tempo impiegato dal segnale per percorrere due volte lo spazio tra il sensore e l'ostacolo e, tramite semplici calcoli matematici, è possibile ricavare la distanza che intercorre tra il sensore e l'ostacolo stesso.

E' importante precisare che il segnale di ECHO è settato a 5 V, nonostante i pin di input di Raspberry siano a 3.3 V; pertanto, onde evitare possibili danneggiamenti alla scheda, è necessario utilizzare un partitore di tensione, in modo da abbassare il segnale di 5 V del pin ECHO a 3.3 V.

Di seguito, viene riportato lo schema circuitale che mostra i collegamenti effettuati per inserire correttamente il sensore HC-SR04 [*Fig.8*].



Schema circuitale HC-SR04 [Fig.8]

Anche per questo sensore, sono state utilizzate delle librerie esterne, nello specifico sono state importate la libreria RPi.GPIO e la libreria Time.

La prima permette di settare correttamente i pin GPIO della scheda Raspberry.

La seconda invece permette di sfruttare gli intervalli di sleep e misurare gli istanti di tempo, entrambe operazioni fondamentali per calcolare il tempo che intercorre tra l'emissione dell'onda e la ricezione di quella riflessa.

Si riporta di seguito [Fig.9] l'intero codice sviluppato per settare il sensore HC-SR04 e successivamente calcolare la distanza tra sensore e ostacolo:

```

1. # librerie per hcsr04
2. import RPi.GPIO as GPIO
3. import time
4.
5. def hcsr():
6.     GPIO.setmode(GPIO.BCM)
7.
8.     TRIG = 20
9.     ECHO = 16
10.
11.    count = 5
12.    somma = 0
13.    i = 0
14.
15.    while i < count:
16.        GPIO.setwarnings(False)
17.        GPIO.setup(TRIG, GPIO.OUT)
18.        GPIO.setup(ECHO, GPIO.IN)

```

```

19.      # siamo sicuri che il trigger sia a false
20.      GPIO.output(TRIG, False)
21.      print("Waiting for sensor to settle")
22.      time.sleep(2)
23.
24.      # il segnale in ingresso necessita di un impulso di 10us
25.      GPIO.output(TRIG, True)
26.      time.sleep(0.00001)
27.      GPIO.output(TRIG, False)
28.
29.      while GPIO.input(ECHO) == 0:
30.          pulse_start = time.time()
31.
32.      while GPIO.input(ECHO) == 1:
33.          pulse_end = time.time()
34.
35.      # calcolo del tempo che impiega il segnale ad andare e tornare
36.      # contro l'oggetto
37.      pulse_duration = pulse_end - pulse_start
38.
39.      # velocità del suono è uguale a 34300 cm/s e il tempo è metà in
40.      # quanto deve fare andata e ritorno
41.      distance = pulse_duration * 34300/2
42.      distance = round(distance, 2)
43.
44.      i += 1
45.      somma += distance
46.

```

Codice misurazioni HC-SR04 [Fig.9]

Dopo aver settato i pin della scheda Raspberry alla modalità di lettura BCM (riga 6), si specificano i pin assegnati al TRIG e a ECHO (riga 8 e 9) e si inizializzano le variabili count, somma e i.

E’ importante specificare che, con l’obiettivo di ottenere una misurazione sufficientemente precisa della distanza rilevata, si è deciso di eseguire una media di cinque misurazioni. Per questa ragione la funzione “hcsr()” ritorna il rapporto tra la somma delle distanze rilevate e il numero di misurazioni effettuate (riga 45).

All’interno del ciclo while, necessario per effettuare le cinque misurazioni, si setta il pin di TRIG come output (riga 17) e il pin di ECHO come input (riga 18).

A questo punto, dopo essersi assicurati che il trasmettitore sia effettivamente spento, cioè TRIG sia settato a false (riga 20), si accende il trasmettitore settando a true il TRIG (riga 25), per poi spegnerlo dopo dieci microsecondi (riga 27).

Il ciclo while di riga 29 permette di rilevare il valore di inizio della trasmissione; infatti si rimane all'interno del ciclo fino a quando ECHO è settato a low, cioè fino all'istante in cui inizia la trasmissione del segnale emesso dal sensore, ovvero il momento in cui TRIG si attiva.

Il ciclo while di riga 32, invece, permette di rilevare l'istante precedente all'arrivo del segnale riflesso. Si rimane in questo ciclo fino a quando ECHO è settato a 5 V, cioè dall'istante in cui inizia la trasmissione fino all'istante in cui viene rilevata l'onda riflessa.

Ora, conoscendo l'istante di inizio e di fine della trasmissione si calcola la durata della pulsazione (riga 36).

Inoltre si assume che la velocità di propagazione del suono nell'aria sia approssimabile a circa 343 m/s e si ricorda che la durata della pulsazione precedentemente calcolata è in realtà l'intervallo di tempo impiegato dal segnale per percorrere due volte la distanza tra sensore e ostacolo.

Quindi, fatte tali considerazioni, si può utilizzare la formula riportata alla riga 39 del codice per ottenere la misurazione effettiva in cm della distanza tra sensore e ostacolo.

In conclusione, conoscendo l'altezza della tanica, è possibile poi ricavare il livello di acqua rimanente all'interno della tanica destinata all'irrigazione e quindi la percentuale di riempimento della tanica stessa.

3.3. SCHEDA RELE' E ATTUATORI

I dati prodotti dai sensori servono alla scheda Raspberry per monitorare le condizioni ambientali presenti all'interno della serra. Attraverso l'azionamento di una serie di attuatori, la scheda è inoltre in grado di intervenire sui parametri misurati tramite i sensori, andando quindi a ristabilire le condizioni ambientali ottimali per la crescita della pianta coltivata.

Con il termine attuatore si intende “*un meccanismo attraverso cui un agente agisce su un ambiente*” (Wikipedia), dove, nel caso in esame, l'agente è un componente che trasforma il segnale in input, ovvero la corrente, in un effetto desiderato.

Per svolgere questo compito è stato necessario introdurre nel progetto una scheda relè.

Si tratta di un componente collegabile alla scheda Raspberry che permette, attraverso degli impulsi inviati dal controllore, di attivare o disattivare gli attuatori collegati ai diversi relè che compongono la scheda stessa.

In particolare, la scheda acquistata per il progetto possiede otto diversi relè, nonostante ne siano stati impiegati solamente cinque.

I relè sono dei dispositivi che, attraverso dei fenomeni elettromagnetici, riescono a svolgere il ruolo di interruttori. Facendo passare la corrente attraverso una bobina, è possibile generare un campo magnetico, che, con la forza magnetica da esso generata, permette di spostare altri componenti.

Sfruttando tale principio fisico è possibile quindi cambiare la posizione di alcuni componenti metallici, che agiscono di fatto come se fossero degli interruttori, andando così, a seconda dell'intensità della forza prodotta dalla bobina, a chiudere o ad aprire i circuiti ad essi collegati.

I circuiti in questione sono dotati di alimentazione, ground, e presentano un collegamento ad un attuatore, il cui funzionamento viene quindi regolato dal relè stesso.

La scheda relè è dotata di dieci pin differenti (anche se solo sette sono stati impiegati nel progetto):

- un VDD che corrisponde all'alimentazione a 3.3 V;
- un GND che corrisponde al ground;
- otto diversi pin (di cui ne sono stati sfruttati solo cinque) che servono per ricevere il segnale d'impulso ricevuto dalle porte GPIO di Raspberry, per poter aprire o chiudere il circuito collegato al relè;

Per quanto riguarda invece i cinque diversi circuiti associati ai corrispondenti relè, bisogna sottolineare che gli attuatori impiegati nel progetto lavorano tutti ad un voltaggio di 5 V. Si è preferito infatti, per ragioni di sicurezza, utilizzare dei componenti che lavorassero a bassa tensione. Inoltre questa scelta ha permesso di sfruttare direttamente i pin della scheda Raspberry a 5V, senza dover introdurre l'utilizzo di una batteria esterna.

In particolare, l'alimentazione, fornita dal controllore, va collegata nel foro centrale del relè, mentre il cavo che porta l'alimentazione all'attuatore stesso va inserito nel foro a sinistra. Per ultimo, il ground dell'attuatore va collegato a un ground di Raspberry. In questo modo il circuito passa per il relè, il quale decide quando la corrente può fluire fino all'attuatore, agendo quindi da interruttore.

I cinque diversi attuatori impiegati per le attività di controllo sono:

- una pompa, inserita all'interno di una tanica d'acqua, che, collegata ad un tubo, permette di irrigare il terreno della serra;
- una seconda pompa, anch'essa inserita in una piccola tanica d'acqua, che permette di umidificare la serra attraverso un piccolo ugello;
- una terza pompa, inserita in una tanica contenente del fertilizzante, che permette di fertilizzare il terreno quando necessario;
- una ventola, utile sia per il controllo del livello di umidità dell'aria sia per favorire un ricircolo continuo;
- una striscia led, necessaria per la simulazione del ciclo di luce e buio a cui sottoporre la pianta. L'utilizzo delle luci led è fondamentale per due

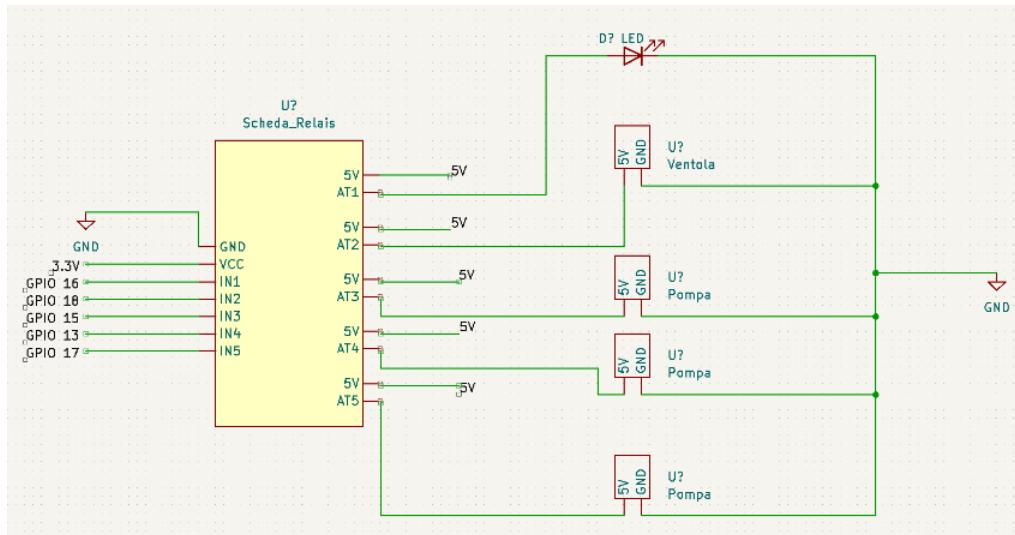
ragioni: le pareti della serra limitano notevolmente il passaggio dei raggi solari e quindi è necessaria una luce artificiale; inoltre con i led si possono scegliere i cicli di luce a proprio piacimento, senza essere vincolati dalla luce solare.

Tuttavia si specifica che i led utilizzati, per via del basso voltaggio, non forniscono una quantità di lumen sufficiente; pertanto rappresentano solo una simulazione e non una soluzione realmente funzionante.

Si ricorda che il progetto è solo un prototipo: di conseguenza, anche gli attuatori sono lontani dai componenti professionali impiegati nelle serre reali.

Queste limitazioni sono legate sia al budget a disposizione, sia soprattutto alle limitazioni imposte dal voltaggio utilizzato.

Di seguito viene riportato lo schema circuitale che mostra la scheda relè e i collegamenti con gli attuatori [Fig.10].



Schema circuitale scheda relè [Fig.10]

All'interno del file “Relais.py” è contenuto tutto il codice necessario per controllare l'accensione o spegnimento degli attuatori. Si tratta di quattro diverse funzioni, abbastanza simili l'una con l'altra, ma che rimangono separate in quanto svolgono compiti leggermente diversi.

Per questo motivo, per tutte e quattro le funzioni verrà spiegato il loro scopo, ma il codice verrà analizzato solamente per la prima, ovvero la più completa, dato che poi gli stessi comandi si ritrovano nelle funzioni successive.

- la prima funzione ha lo scopo di attivare un attuatore e di mantenerlo acceso per un intervallo di tempo passato come parametro nella funzione stessa;
- la seconda ha lo scopo di attivare il relè a cui sono collegati i led e di mantenerlo acceso;
- la terza va invece a spegnere i led e quindi ad aprire il circuito del relè a cui sono collegati;
- l'ultima, invece, apre tutti i circuiti di tutti e cinque i relè, andando quindi a spegnere tutti gli attuatori.

Si osserva inoltre che in tutte e quattro le funzioni sono state impiegate due librerie. La prima è necessaria per riferirsi ai pin della scheda Raspberry, la seconda serve invece per poter sfruttare gli intervalli di tempo necessari per tenere accesi gli attuatori fino a un determinato istante.

Si riporta, quindi, come già anticipato, il codice della prima funzione [Fig.11]:

```
1. import RPi.GPIO as GPIO
2. import time
3.
4. # funzione da richiamare per gestire la scheda relais nel caso di pompe
   e ventola
5. def relais_attuatori(num_porta, tempo):
6.     # per settare la nomenclatura dei GPIO
7.     GPIO.setmode(GPIO.BCM)
8.     # GPIO.cleanup()
9.
10.    GPIO.setwarnings(False)
11.    # settaggio pin che vogliamo usare con la nomenclatura bcm
12.    GPIO.setup(num_porta, GPIO.OUT)
13.
14.    GPIO.output(num_porta, GPIO.LOW)
15.    time.sleep(tempo)
16.    GPIO.output(num_porta, GPIO.HIGH)
```

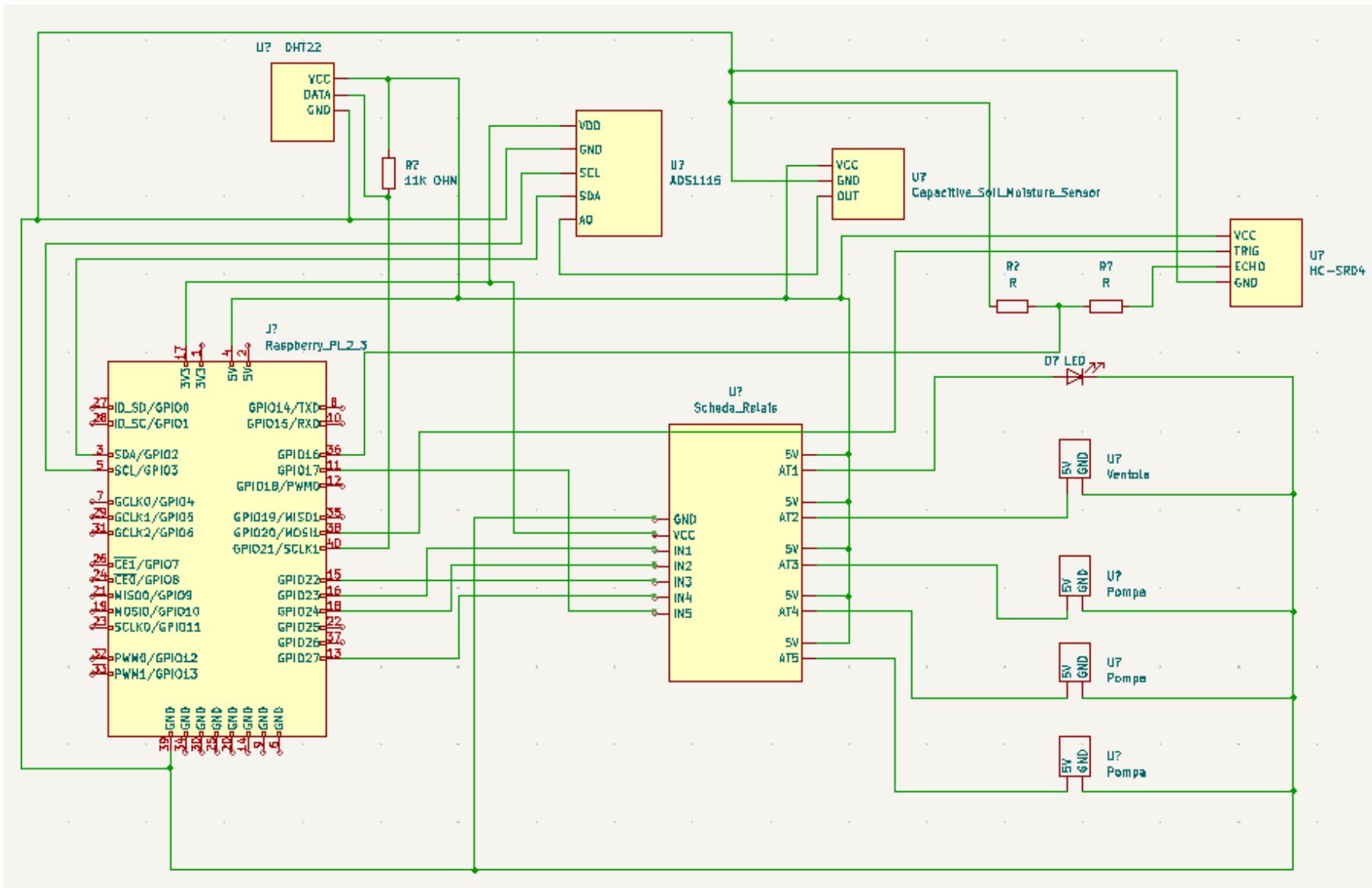
Codice controllo scheda relè [Fig.11]

Dopo aver importato le due librerie, si setta la nomenclatura che verrà utilizzata per riferirsi ai pin di Raspberry (riga 7); poi si inizializza come porta di output il pin passato come parametro alla funzione (riga 12).

Successivamente avviene la chiusura del circuito e si permette quindi il passaggio di corrente, con conseguente accensione dell'attuatore collegato (riga 14). Si mantiene questa situazione per un intervallo di tempo specificato dal parametro della funzione (riga 15), per poi andare a riaprire il circuito e spegnere così l'attuatore stesso.

Nella pagina seguente viene mostrato lo schema circuitale dell'intero progetto, contenente la scheda Raspberry, la scheda relè, l'ADS1115, i sensori e gli attuatori utilizzati [*Fig.12*].

In seguito, nei successivi paragrafi, si andranno ad osservare in dettaglio le azioni da effettuare per il controllo della serra e, attraverso la spiegazione del funzionamento della serra stessa, si potrà capire quando effettivamente le funzioni legate agli attuatori verranno richiamate.



“Schema circuitale completo” [Fig.12]

3.4. UML STATE CHART DIAGRAM

In questo paragrafo si inizierà a mostrare il funzionamento del sistema Embedded.

Prima di osservare direttamente il codice e analizzare la schedulazione delle attività, è necessario comprendere gli stati nei quali la serra potrebbe trovarsi.

A tale scopo si sfrutterà lo statechart diagram [*Fig.13*], ovvero il diagramma degli stati sviluppato con UML per poter avere una visione migliore del funzionamento e del ciclo di attività del sistema Embedded.

Come premessa, si precisa che nello statechart diagram molto spesso i nomi delle attività non coincidono esattamente con i nomi reali delle funzioni presenti nel codice, ma si è preferito procedere in questo modo per rendere il più esplicativo possibile il diagramma che ha il solo scopo di mostrare il funzionamento del sistema.

Inoltre nello schema non vengono trattate tutte le attività, per esempio non si mostra come avviene la comunicazione tra sistema Embedded e architettura Cloud. Nel diagramma infatti si vuole mostrare solo ciò che è puramente funzionale al controllo della serra.

Lo schema si divide in due macrostati “Luce accesa” e “Luce Spenta”.

Si presuppone che la prima accensione della scheda Raspberry e di tutto il sistema Embedded avvenga in un orario compreso tra le 08:00 e le 20:00.

Lo schema comincia dal simbolo di initial state (pallino nero) presente nel macrostato “Luce accesa”; successivamente viene svolta l’attività di ingresso del macrostato. Tale attività si occupa di chiudere il circuito dei led attraverso l’utilizzo della scheda relè, portando all’accensione delle luci.

Il ciclo di funzionamento della scheda Raspberry continua ad alternarsi tra questi due macrostati; infatti nel decision node, presente al centro del diagramma,

quando l'orario torna ad essere successivo alle ore 20:00, si passa allo stato “Luce Spenta”, che porta ad aprire il circuito e quindi allo spegnimento dei led.

Il sistema rimarrà nello stato di “Luce Spenta” fino alle ore 08:00 del giorno successivo.

In questo modo è stato gestito il ciclo di luce e buio per la serra. Si precisa che, qualora ve ne sia la necessità, tale ciclo è liberamente modificabile.

Ora si passa ad un'analisi più approfondita del corpo dello stato “Luce Accesa”.

Come premessa, bisogna specificare che, per permettere ai due sensori principali del sistema Embedded, ovvero il DHT22 e il Capacitive Soil Moisture Sensor, di alternarsi correttamente nelle rispettive misurazioni, si è deciso di associare a ciascuno un thread. I due thread hanno inoltre in comune un lock, il quale deve essere necessariamente acquisito per poter effettuare una rilevazione. Questo concetto verrà ripreso nel dettaglio nel paragrafo successivo, quando si parlerà di schedulazione; al momento queste informazioni sono sufficienti per comprendere il diagramma nella sua interezza.

All'inizio delle attività, appena entrati nel macrostato “Luce Accesa”, si passa subito nel sottostato “Termine Attività”, che obbliga i due thread a rilasciare il lock. La funzione di rilascio del lock è fondamentale per le fasi successive a quella iniziale che si sta considerando. In generale, però, la funzione di rilascio del lock è importante anche in questo momento poiché garantisce che, qualora ci sia un thread che abbia ancora il possesso di un lock da cicli precedenti, lo rilasci. Successivamente, se l'orario è precedente alle ore 20:00, si entra in uno stato di idle nel quale il sistema attende fino a che uno dei due thread acquisisca il lock.

In base al thread che possiede il lock si procede in una delle due direzioni proposte dal decision node in basso al centro: “Lettura DHT22” o “Lettura Capacitive”.

Si presenta prima il caso in cui sia il thread del DHT22 ad acquisire il lock, seguendo quindi la parte destra dello schema UML.

Inizialmente si entra nello stato “Lettura DHT22” dove, attraverso il codice mostrato nel paragrafo dedicato al sensore DHT22, viene effettuata la misurazione della temperatura ambientale e del livello di umidità dell’aria, andando quindi ad aggiornare le medie rispettive.

Come verrà spiegato nel paragrafo successivo, queste attività di aggiornamento sono fondamentali, in quanto le decisioni che portano all’attivazione degli attuatori devono essere prese basandosi sulle medie di numerose rilevazioni fatte dai sensori e non sulla singola misurazione, la quale potrebbe essere affetta da errori.

Successivamente all’operazione di lettura del DHT22, si presenta un nodo decisionale con tre diverse possibilità:

- qualora la media delle misurazioni dell’umidità dell’aria dovesse essere un valore compreso tra un minimo e un massimo prestabiliti, ci si trova nella situazione ideale, di conseguenza si segue il ramo con [else];
- qualora invece la media dell’umidità dell’aria sia un valore maggiore di un massimo accettabile, si entra nello stato “Ventilazione”.

In questo stato, si va ad accendere e successivamente spegnere la ventola, con lo scopo di ridurre il livello di umidità, attraverso un ricircolo di aria. La ventola viene gestita dal codice già presentato nel paragrafo “Scheda relè e attuatori”;

- infine, se la media dovesse essere un valore minore di un minimo accettabile, si entra nello stato “Umidificazione”. In questo caso, infatti, è necessario accendere la pompa dell’umidificazione, per far sì che l’umidità possa alzarsi oltre il minimo consentito.

Al termine di una qualsiasi di queste tre diverse opzioni si ritorna allo stato “Termine Attività”.

Si analizza ora la situazione in cui sia il thread del Capacitive Soil Moisture Sensor ad aver acquisito il lock.

Si entra quindi in uno stato chiamato “Lettura Capacitive” che si occupa di effettuare la misurazione e, successivamente, di aggiornare la media dell’umidità del suolo.

Terminata l’attività di lettura si presenta un nodo decisionale, che propone due alternative:

- qualora la media dell’umidità del suolo sia maggiore di un valore minimo accettabile, si segue il ramo senza entrare in ulteriori stati e si ritorna allo stato “Termine Attività”;
- nel caso opposto, invece, ci si trova in una situazione in cui l’umidità del suolo è troppo bassa; ciò significa che la pianta necessita di essere irrigata.

Si entra dunque nello stato “Irrigazione”, che si occupa di accendere e successivamente spegnere la pompa preposta all’irrigazione. Inoltre viene attivato il sensore HC-SR04, collocato sul tappo della tanica, per poter controllare il nuovo livello dell’acqua in seguito all’irrigazione. Il codice che viene utilizzato è quello visto nel paragrafo dedicato a tale sensore.

Si continua lo studio proseguendo sul ramo successivo allo stato “Irrigazione”.

Per prima cosa, terminato lo stato, si incrementa il contatore delle irrigazioni effettuate. Questa operazione è necessaria per regolare i cicli di fertilizzazione della pianta; si è scelto di fertilizzare il terreno ogni tre cicli di irrigazione. Si specifica, in ogni caso, che anche questo valore può essere facilmente modificato in base alle necessità.

Il nodo decisionale successivo analizza le alternative sulla base del valore del contatore appena aggiornato. In particolare:

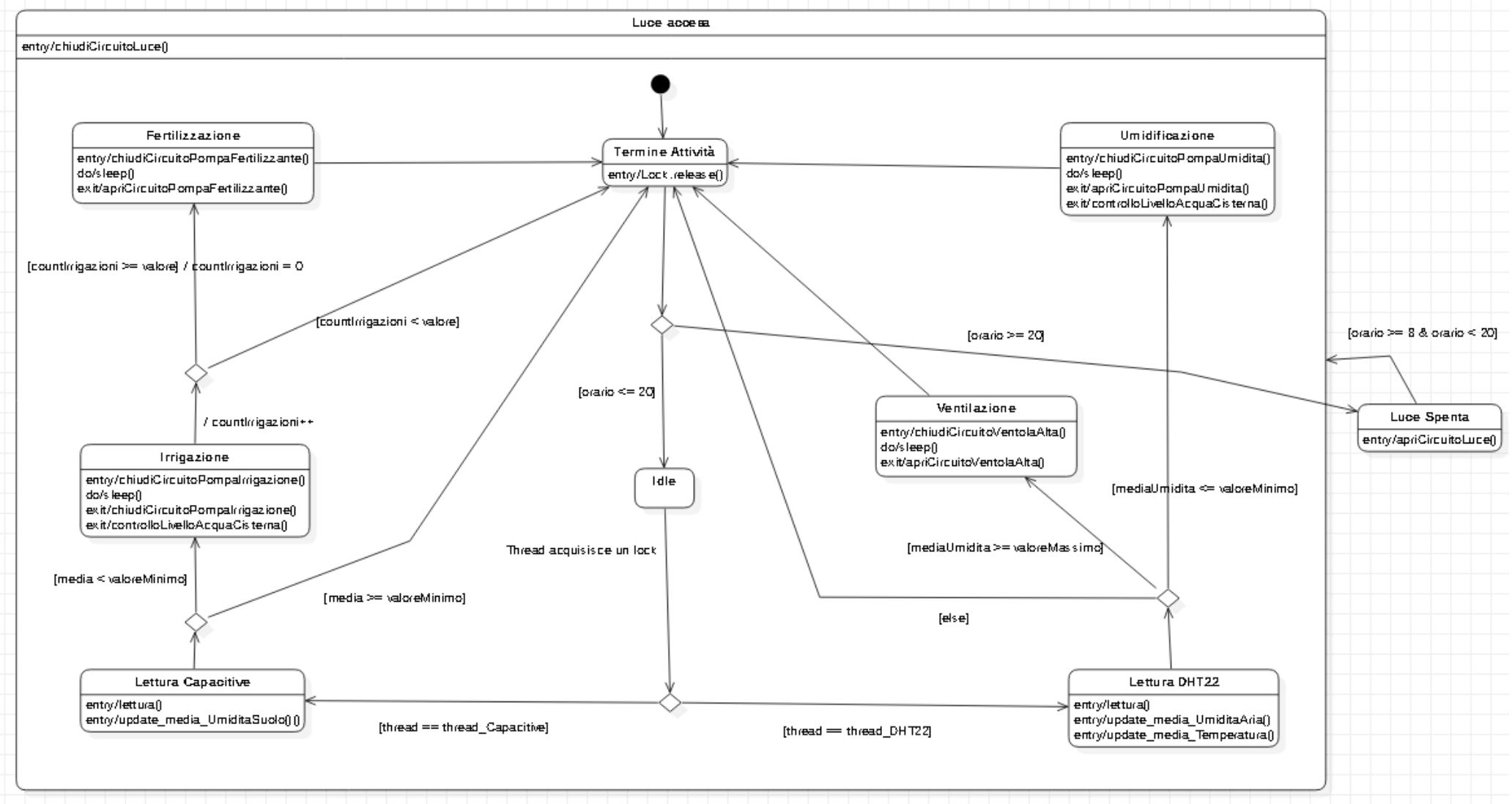
- se il valore del contatore è minore del valore di riferimento prestabilito (nel caso presentato tre), allora significa che non si è ancora arrivati al numero di irrigazioni consecutive dopo le quali è necessario introdurre del fertilizzante; pertanto si ritorna allo stato “Termine Attività”;
- qualora invece il valore del contatore sia uguale al valore di riferimento, si procede ad azzerare il contatore e successivamente si entra nello stato

“Fertilizzazione”. Tale stato si occupa di azionare e successivamente spegnere la pompa predisposta alla fertilizzazione del terreno. Conclusa questa fase si ritorna allo stato “Termine Attività”.

In generale, quindi, qualunque sia il percorso che si scelga di seguire all'interno dello stato “Luce Accesa”, si ritorna sempre allo stato “Termine Attività”.

In questo modo si rilascia il lock consentendo così all'altro thread di acquisirlo. Dunque i thread collegati ai due sensori si alternano in modo ciclico.

Lo statechart non presenta una fine poiché, teoricamente, il sistema è pensato per funzionare senza nessuna scadenza.



Statechart Diagram [Fig.13]

3.5. SCHEDULAZIONE THREAD

Dopo aver osservato il funzionamento generale del sistema Embedded attraverso l'utilizzo dello statechart diagram, si procede ora con uno studio più specifico del codice di controllo, attraverso il quale la scheda Raspberry gestisce ogni attività. E' necessario introdurre alcune premesse, al fine di comprendere al meglio come avviene la schedulazione dei compiti.

La gestione di attività parallele deve avvenire attraverso l'utilizzo dei thread.

All'interno del processo istanziato dal sistema operativo di Raspberry, che gestisce l'esecuzione del codice scritto in Python, vengono istanziati altri flussi, definiti thread. Questi sottoprocessi fanno parte dello stesso processo padre e possono essere eseguiti in parallelo. Ciò permette, quindi, associando una determinata attività ad ogni thread, di eseguire compiti anche in parallelo.

In alternativa, con l'utilizzo di altri meccanismi di sincronizzazione, come ad esempio i lock, è possibile andare a controllare il multithreading, gestendo le risorse condivise da più sottoprocessi in maniera sequenziale.

Nel progetto si è deciso di lavorare con tre diversi thread:

- un primo thread, che si occupa di gestire l'accensione e lo spegnimento delle luci led;
- altri due thread, che lavorano solo durante il ciclo 08:00-20:00, i quali si occupano rispettivamente delle misurazioni fatte dal DHT22 e di quelle fatte dal Capacitive Soil Moisture Sensor.

Si può osservare come il multithreading permetta quindi di svolgere le attività in parallelo. Ad esempio, in un orario compreso tra le 08:00 e le 20:00, si avranno attivi contemporaneamente sia il thread che gestisce le luci led, sia i due thread che gestiscono le attività dei sensori.

Per quanto riguarda questi ultimi due sottoprocessi, si è optato per una gestione seriale delle loro attività; pertanto solo uno dei due task può essere in esecuzione in un dato momento.

In particolare, come già osservato nel paragrafo precedente, l'attività dei due diversi sensori comporta anche azioni di accensione e spegnimento degli attuatori ad essi connessi, necessari per modificare lo stato interno della serra. Per questo motivo, per evitare che durante tali situazioni l'altro sensore continui ad effettuare misurazioni, si è preferito alternare serialmente i due thread.

Questa scelta permette, quindi, di non avere rilevazioni che possano essere affette da errori o che comunque dipendano fortemente dalla situazione particolare generata dal lavoro di un attuatore.

Ad esempio, se il DHT22 ha rilevato un'umidità dell'aria troppo bassa, procederà ad attivare la pompa preposta all'umidificazione; in tale situazione, però, è necessario bloccare le misurazioni dell'umidità del suolo svolte dal Capacitive Soil Moisture Sensor, in quanto, se così non fosse, il rischio di registrare nuovi valori di umidità del terreno molto più elevati rispetto a quelli reali sarebbe alto. E' necessario, quindi, attendere la fine del lavoro dell'attuatore, prima di poter riprendere con le misurazioni.

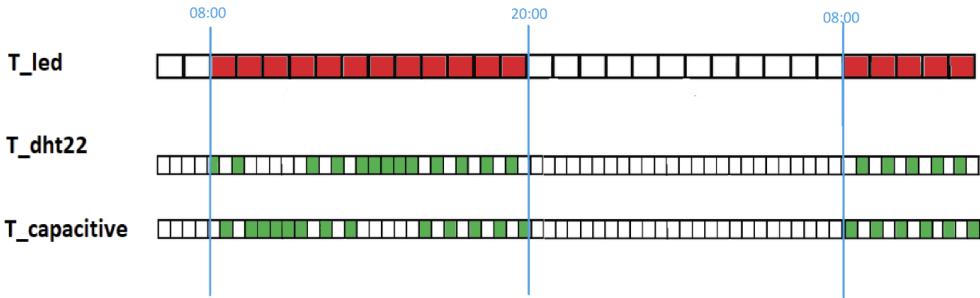
Questo giustifica la decisione di impiegare il meccanismo dei lock tra i due thread che si occupano delle rilevazioni. In questo modo, quando uno dei due possiede la risorsa condivisa, l'altro non può effettuare nessun rilevamento.

Inoltre, dato che le velocità di rilevazione dei due sensori non sono uguali, è necessario impedire che possa nascere la situazione in cui solo uno dei due sensori riesce ad acquisire il lock, a discapito dell'altro. Per questo motivo, si è optato per alternare sempre l'acquisizione del lock, impedendo quindi che uno stesso thread possa ottenerlo per due o più volte consecutivamente.

In seguito verrà mostrato il codice impiegato per ottenere questo effetto.

Viene riportata un'immagine [*Fig.14*] rappresentativa della schedulazione delle attività svolte dai vari thread:

Nome Thread	Tipologia attività	Periodo	Tempo Computazionale
Thread led	periodica	12 ore	12 ore
Thread DHT22	sporadica		Qualche ms- qualche minuto
Thread Capacitive	sporadica		Qualche ms- qualche minuto



Schedulazione attività dei tre diversi thread [Fig.14]

E' possibile quindi osservare come il thread che gestisce i led abbia un'attività periodica; infatti i led sono attivi tra le 08:00 e le 20:00 e spenti tra le 20:00 e le 08:00 del giorno seguente. L'attività di accensione delle luci è dunque periodica, ma il thread rimane sempre attivo. Nell'immagine i quadratini rossi del thread del led hanno durata di un'ora.

Gli altri due thread hanno invece delle attività sporadiche: sono infatti attivi solamente quando le luci sono accese, ovvero tra le 08:00 e le 20:00, ma svolgono i task in modo alternato, non per tutta la durata del loro periodo di attività. In questa situazione, un quadratino verde rappresenta qualche millisecondo.

Si osserva come, per quanto detto precedentemente, i quadratini verdi siano sempre alternati e mai in parallelo su entrambi i thread.

A volte sono presenti più quadratini in serie per lo stesso thread; questo serve a rappresentare il caso in cui, in seguito alla misurazione, è stato attivato anche qualche attuatore; di conseguenza il tempo durante il quale un thread è in possesso del lock è di qualche decina di secondi o anche di qualche minuto.

3.6. SPIEGAZIONE CODICE

Si procede quindi a presentare un possibile flow dell'esecuzione del codice, con lo scopo di mostrare al meglio il funzionamento.

Il file Python che viene eseguito per primo è “Controller.py”.

Prima di mostrare il funzionamento, è possibile osservare come, nello stesso file, sia presente anche il codice spiegato nel paragrafo due, per lo scambio dei dati tra Raspberry e architettura Cloud.

Come prima cosa vengono importate le librerie *[Fig.15]* necessarie; inoltre vengono importati anche due moduli sviluppati dai candidati.

```
1. from threading import Thread
2. import time
3. from datetime import datetime, timedelta
4.
5. import ThreadParallelisi as figli
6. import Relais as rel
7.
```

Librerie Controller.py [Fig.15]

Il modulo importato a riga 5 è quello sviluppato dai candidati e contenente le classi per i due thread che si occupano delle misurazioni.

Quello importato a riga 6 è il modulo sviluppato dai candidati, già affrontato nel paragrafo 3.3, e necessario per il controllo della scheda relè e degli attuatori ad essa collegati.

```
1. t = Thread_padre(identificativo = "padre" )
2. t.start()
3.
```

Creazione del thread [Fig.16]

Il codice inizia *[Fig.16]* andando ad istanziare una classe “Thread_padre” che estende la classe Thread e successivamente facendo partire tale sottoprocesso.

```

1. #thread padre
2. class Thread_padre(Thread):
3.     def __init__(self, identificativo):
4.         Thread.__init__(self)
5.         self.identificativo = identificativo
6.
7.     def run(self):
8.         print("I am the father")
9.
10.    while TRUE:
11.        t1 = figli.Thread_parallel(activity = figli.activity_DHT22,
12.                                     identificativo = "DHT22" )
13.        t2 = figli.Thread_parallel(activity =
14.                                     figli.activity_Capacitive, identificativo = "Capacitive" )
15.        t1.start()
16.        t2.start()
17.        #luci accese, T1 e T2 lavorano
18.        rel.relaies_ON_luce(GPIO_luci)
19.        orario_attuale = datetime.now()
20.        print(orario_attuale)
21.        confronto_sera = orario_attuale.replace(hour = 20, minute =
22.                                                    0, second = 0)
23.        #faccio dormire il thread luci, continuano T1 e T2 (sono tra
24.        #le 7 e le 20)
25.        while orario_attuale < confronto_sera:
26.            print("Sto dormendo...")
27.            time.sleep(30)
28.            #inviamo i dati al cloud
29.            chiamata_Http()
30.            orario_attuale = datetime.now()
31.            orario_attuale = datetime.now()
32.            #print(orario_attuale)
33.
34.            confronto_mattina = datetime.now()
35.            confronto_mattina += timedelta(days = 1)
36.            confronto_mattina = confronto_mattina.replace(hour = 8,
37.                                                       minute = 0, second = 0)
38.            # per sicurezza si aprono tutti i circuiti della scheda
39.            #relais per evitare che qualche attuatore rimanga attivo
40.            rel.relaies_OFF()
41.            #stoppo T1 e T2 e spengo luci (siamo tra le 20 e le 7)
42.            rel.relaies_OFF_luce(GPIO_luci)
43.            #faccio dormire il thread luci, T1 e T2 sono stoppati
44.            while orario_attuale < confronto_mattina:
45.                print("Sto dormendo...")
46.                time.sleep(10)
47.                orario_attuale = datetime.now()

```

Classe “Thread_padre”[Fig.17]

La classe “Thread_padre” [Fig.17] ha il suo costruttore e il metodo “run()” ereditato dalla superclasse “Thread”. Questo metodo viene richiamato quando il thread comincia.

Si entra subito in un ciclo infinito per via della presenza della condizione TRUE. Ciò mostra che il thread delle luci rimane sempre attivo e non termina mai la sua esecuzione.

All’interno del ciclo, dalla riga 11 alla 15, attraverso il modulo “Thread paralleli” (rinominato “figli”) vengono istanziati e fatti partire i due thread volti alle misurazioni.

Vengono quindi accese le luci (riga 18) attraverso il modulo della scheda relais.

Si passa poi a controllare l’orario durante il quale viene eseguita tale linea di codice (riga 19), settando inoltre il primo orario necessario per il ciclo luce/buio della serra, ovvero le 20:00 (riga 21).

Fin quando l’orario trovato rimane precedente alle ore 20:00, si resta nel ciclo di riga 24. All’interno di tale ciclo, con lo scopo di non sovraccaricare la scheda Raspberry, si addormenta il thread per qualche secondo attraverso il comando di sleep; successivamente viene effettuata la chiamata Http POST, come visto nel secondo paragrafo, e infine si aggiorna l’orario attuale.

Quando si otterrà un orario con valore superiore alle 20:00 si uscirà dal ciclo.

L’orario attuale viene dunque aggiornato (riga 31) e viene settata l’ora mattutina delle 08:00 del giorno successivo (riga da 34 a 36).

Vengono spenti per sicurezza tutti gli attuatori, luci led comprese (riga 41), andando ad aprire i circuiti associati ai relè (riga 39).

Infine, si entra in un ultimo ciclo, simile al precedente, in cui si continua ad aggiornare l’orario attuale fino a quando questo non è successivo alle ore 08:00 del giorno seguente (riga da 43 a 46). Questo ciclo, di fatto, rappresenta il periodo durante il quale tutte le attività della serra sono sospese.

Si affronta ora nello specifico il funzionamento dei due thread associati ai sensori, studiando il modulo “ThreadParallel.py”.

```
1. from threading import Thread, RLock
2. import time, queue, random
3. import time
4. from datetime import datetime, timedelta
5.
6. #import dei vari moduli presenti nella stessa cartella di Controller.py
7. import Coda as coda
8. import UpdateQueue as uq
9. import Circuits as cir
10. import Relais as rel
11.
12. Listdimension = 5
13.
14. #liste per fare una media di un certo numero di valori e non considerare
   solo il singolo
15. lista_valori_dht22_temperatura = queue.Queue(Listdimension)
16. lista_valori_dht22_umidita = queue.Queue(Listdimension)
17. lista_valori_capacitive = queue.Queue(Listdimension)
18.
19. #valori di riferimento per prendere decisioni
20. M_temperatura_aria = 0
21. M_umidita_aria = 0
22. M_umidita_suolo = 0
23.
24. # valori costanti
25. Max_umidita_aria = 60
26. Min_umidita_aria = 50
27. Min_umidita_suolo = 40
28. countIrrigazioni = 0
29. numeroIrrigazioni = 3 # dopo quante irrigazioni bisogna fertilizzare
30. altezzariferimento_vuoto = 16
31. altezza_vuoto = 0
32.
33. # numero di porte GPIO per gli attuatori
34. GPIO_pompa_irrigazione = 17
35. GPIO_pompa_umidificazione = 27
36. GPIO_pompa_fertilizzante = 22
37. GPIO_ventola = 24
38.
39. # tempi di attesa per il funzionamento dei vari attuatori
40. time_irrigazione = 10
41. time_umidificazione = 10
42. time_fertilizzante = 10
43. time_ventola = 10
44.
```

Librerie e variabili globali [Fig.18]

Il codice [Fig.18] comincia con l'importazione delle librerie e il settaggio di alcune variabili globali necessarie al funzionamento del sistema Embedded.

In particolare sono presenti alcuni moduli come Coda e UpdateQueue (riga 7 e 8) che sono stati sviluppati dai due candidati e verranno analizzati nel dettaglio in seguito.

Dalla riga 15 alla 17 vengono create le tre diverse Queue per le misurazioni di temperatura aria, umidità aria e umidità suolo. Si è infatti già mostrato come le decisioni sull'eventuale accensione di un attuatore debbano essere prese sulla base della media delle ultime rilevazioni effettuate dal sensore. Per questo motivo si è scelto di utilizzare la Queue, ovvero una struttura dati con politica FIFO (First in, First out).

In questo modo, ogni misurazione deve essere inserita nella lista e, qualora quest'ultima dovesse essere piena, si elimina la misurazione più vecchia presente nella struttura dati.

Successivamente, si istanziano le variabili per le medie (riga da 20 a 22) e vengono salvati i valori massimi e minimi accettabili di temperatura e umidità per la serra (riga da 25 a 31).

Per concludere, si indicano i pin a cui ogni relè è collegato (riga da 34 a 37) e il tempo di attesa durante il quale l'attuatore corrispondente è in attività (riga da 40 a 43).

Si riprende quindi il flow di esecuzione del codice, ripartendo dal modulo osservato precedentemente, nel quale venivano creati i due diversi thread per le misurazioni (“Controller.py”). Successivamente si osserva la classe “Thread_parallel”, anch’essa sottoclasse di Thread .

Per poter presentare il corretto funzionamento del metodo “run()”, è necessario introdurre il modulo “Coda.py”[Fig.19].

```

1. #classe custom per gestire la queue dei thread che richiedono un lock
   condiviso
2. class coda:
3.     #la coda ha due campi
4.     #lista di thread che è un array chiamato Elementi
5.     #l'ultimo elemento che è stato buttato fuori dalla lista che
       all'inizio è 1 a caso
6.     def __init__(self):
7.         self.Elementi = []
8.         self.lastpop = 1
9.
10.    #la push inserisce in fondo alla lista l'elemento
11.    def push(self, element):
12.        self.Elementi.append(element)
13.
14.    #la pop aggiorna il nuovo ultimo elemento che viene buttato fuori e
       poi lo toglie
15.    def pop(self):
16.        self.lastpop = self.head()
17.        return self.Elementi.pop(0)
18.
19.    #head permette di vedere chi si trova in cima alla lista
20.    def head(self):
21.        return self.Elementi[0]
22.
23.    #stampa gli elementi della lista
24.    def print(self):
25.        for el in self.Elementi:
26.            print(el)
27.
28.    #serve per vedere quante istanze di un element sono contenute
       all'interno della lista
29.    def search(self, element):
30.        return self.Elementi.count(element)
31.
32.    #ritorna la lunghezza della lista
33.    def length(self):
34.        return len(self.Elementi)
35.

```

Classe “Coda”[Fig.19]

Come già anticipato in precedenza, si è deciso di alternare il lavoro dei due thread associati ai sensori, in modo tale da non permettere che un thread possa ottenere due volte consecutive il lock.

Per ottenere tale risultato, si è sfruttata una struttura dati con politica FIFO (First In, First Out), per simulare una lista di attesa.

Tale classe è composta da un array di elementi e una variabile che tiene conto dell'ultimo elemento che è stato eliminato dalla lista stessa (riga 7 e 8).

La classe Coda serve per simulare la lista di attesa dei thread che aspettano che il lock condiviso si liberi nuovamente. Sulla base dell'ordinamento dei thread salvati nella coda, si deciderà chi ha accesso al lock; ovviamente l'elemento presente da più tempo nella coda stessa ha maggiore priorità.

Tutti i metodi presenti nella classe saranno impiegati per il corretto funzionamento dell'alternanza dei due thread.

```
1. #variabili globali
2. mutex = RLock()
3. q = coda.coda()
4.
```

Creazione Lock e Coda [Fig.20]

Tornando al modulo “Thread_paralleli”, vengono creati il lock e la coda [Fig.20].

Ora invece si analizza la classe “Thread_paralleli”[Fig.21].

```
1. #classe Thread_paralleli
2. class Thread_paralleli(Thread):
3.     def __init__(self, activity, identificativo):
4.         Thread.__init__(self)
5.         self.activity = activity
6.         self.identificativo = identificativo
7.
8.     def run(self):
9.         global mutex
10.        global q
11.
12.        orario_attuale = datetime.now()
13.        confronto = orario_attuale.replace(hour = 19, minute = 55,
14.        second = 0)
15.        #il run del thread viene fatto solo quando l'orario è prima
16.        delle 20:00
17.        while orario_attuale < confronto:
18.            mutex.acquire()
19.
20.            #se siamo nella situazione di lista vuota dobbiamo inserire
21.            #il thread nella lista di attesa
22.            if(q.length() == 0):
23.                #se però il thread è già stato servito per ultimo
24.                #secondo la lista, allora non lo metteremo in attesa perchè vogliamo
25.                #evitare di servirlo due volte di fila
26.                if(q.lastpop != self.identificativo):
27.                    q.push(self.identificativo)
```

```

23.
24.         #queste attività possono essere fatte solo se la lista ha
   degli elementi
25.         if(q.length() != 0):
26.             #se il thread è in testa alla lista, allora eseguo le
   sue attività e lo tolgo dalla lista
27.             if(q.head() == self.identificativo):
28.                 self.activity()
29.                 q.pop()
30.             #se invece non è in testa, controllo quante istanze di
   lui ci sono nella lista, se il valore è 0 allora posso metterlo, se è
   diverso da 0 vuol dire che è 1
31.             #e significa che è già nella lista e non devo scriverlo
   ancora, se no lo ripeto nella lista
32.             else:
33.                 count = q.search(self.identificativo)
34.                 if(count == 0):
35.                     q.push(self.identificativo)
36.
37.             mutex.release()
38.             #lo sleep serve solo per rendere la simulazione più lenta,
   altrimenti non si vedrebbero le attività
39.             time.sleep(2)
40.             orario_attuale = datetime.now()
41.

```

Classe “Thread_parallel” [Fig.21]

Tale classe presenta un costruttore e il metodo “run()”.

Dal costruttore si può notare come ogni thread possieda:

- un’attività, ovvero una funzione con tutti i task che tale thread deve svolgere: le misurazioni con il sensore e i compiti di controllo. Tali funzioni verranno spiegate successivamente;
- un identificativo per i controlli da effettuare sulla coda.

Si analizza ora il codice del metodo “run()”.

Alla riga 12 viene estratto l’istante di tempo nel quale tale linea di codice viene eseguita. Successivamente si setta alla riga 13 l’orario a cui l’attività del sensore deve terminare, ovvero alle 19:55.

Si rimane quindi nel ciclo della riga 15 fino a che l’orario attuale non risulti essere successivo alle ore 19:55. In quest’ultima situazione, infatti, si esce dal ciclo e si procede con la terminazione del thread.

Una volta entrato nel ciclo il thread acquisisce il lock (riga 16).

Si considerano ora due diverse situazioni::

- se la coda è vuota, allora bisogna verificare che l'ultimo pop() della coda non coincide con il medesimo thread che esegue tale controllo. Si vuole infatti impedire ad un thread di effettuare le misurazioni per due volte di seguito. Quindi, come si può osservare a riga 21 e 22, se la lista è vuota si inserisce tale thread, a condizione che questo non sia stato l'ultimo pop() della lista di attesa;
- qualora la lista di attesa non fosse vuota, si va a controllare se tale thread sia nella testa della lista stessa (riga 27).
 - Se è così, significa che è il thread con maggiore priorità nella coda di attesa, quindi è il suo turno. Di conseguenza, si toglie il sottoprocesso dalla lista di attesa e si eseguono le sue attività (riga 28 e 29);
 - In caso contrario, allora si osserva se il thread è già presente nella lista di attesa; se non c'è lo si aggiunge (riga 35), se c'è non si fa nulla.

Infine, viene rilasciato il lock (riga 37) e si sospende tramite uno sleep di due secondi il thread (riga 39). In questo modo si rende la simulazione più lenta e si riescono ad osservare i comandi di “print()” più facilmente. Al termine, viene aggiornato l'orario attuale (riga 40).

Viene ora affrontato il codice relativo alle attività svolte dal DHT22. [Fig.22]

```
1. ## attività del sensore DTH22, per misura umidità aria e temperatura
2. def activity_DHT22():
3.     global M_temperatura_aria
4.     global M_umidita_aria
5.
6.     #potrebbe accadere che a volte la lettura del valore non vada a buon
    fine, nel caso tralasciamo tale misurazione
7.     try:
8.         #lettura valori
9.         h,t = cir.dht22()
10.
11.        #aggiorno la lista_valori_dht22_temperatura e la
    M_temperatura_aria
```

```

12.         #t = random.randint(0,22)
13.         M_temperatura_aria = uq.update_M(lista_valori_dht22_temperatura,
14.             M_temperatura_aria, t)
15.         M_umidita_aria = uq.update_M(lista_valori_dht22_umidita,
16.             M_umidita_aria, h)
17.         #controllo attività
18.         #le attività possono essere svolte solo se l'array dei valori è
19.         pieno
20.         if(lista_valori_dht22_umidita.full()):
21.             #Umidificazione
22.             if M_umidita_aria < Min_umidita_aria:
23.                 #accendiamo la scheda
24.                 rel.relaies_attuatori(GPIO_pompa_umidificazione,
25.                     time_umidificazione)
26.                 #azzeriamo le medie e svuotiamo gli array
27.                 uq.clearList(lista_valori_dht22_temperatura)
28.                 uq.clearList(lista_valori_dht22_umidita)
29.                 uq.clearList(lista_valori_capacitive)
30.                 M_temperatura_aria = 0
31.                 M_umidita_aria = 0
32.                 M_umidita_suolo = 0
33.             #Ventilazione
34.             elif M_umidita_aria > Max_umidita_aria:
35.                 #accendiamo la scheda
36.                 rel.relaies_attuatori(GPIO_ventola, time_ventola)
37.                 #azzeriamo le medie e svuotiamo gli array
38.                 uq.clearList(lista_valori_dht22_temperatura)
39.                 uq.clearList(lista_valori_dht22_umidita)
40.                 uq.clearList(lista_valori_capacitive)
41.                 M_temperatura_aria = 0
42.                 M_umidita_aria = 0
43.                 M_umidita_suolo = 0
44.         except:
45.             pass

```

Codice funzione “activity_DHT22()” [Fig.22]

Il codice comincia con la definizione della funzione “activity_DHT22()” che, come suggerisce il nome stesso, contiene tutte le attività svolte dall’omonimo sensore.

Alle righe 3 e 4 vengono recuperate le due variabili globali, già osservate precedentemente all’inizio dello studio del modulo “Thread_parallel.py”, necessarie per salvare le medie della temperatura e dell’umidità ambientale.

Si precisa che il blocco try-except (da riga 7) è stato inserito con lo scopo di arginare eventuali problematiche che sollevano delle eccezioni. Queste situazioni potrebbero presentarsi nel caso in cui alcune misurazioni effettuate dal sensore non vadano a buon fine. Qualora si dovessero riscontrare tali problematiche, il codice ignora l'eccezione sollevata, attraverso il blocco except, senza rimanere bloccato.

Come prima operazione del blocco try si leggono i valori di umidità dell'aria e della temperatura attraverso il DHT22 e si salvano i risultati registrati rispettivamente nelle variabili h e t (riga 9).

Successivamente si aggiornano le code contenenti le misurazioni rispettive e si aggiornano le medie di umidità dell'aria e della temperatura (riga 13 e 14).

A questo punto, si controlla che le code delle medie siano piene (riga 18); dato che il controllo dei parametri non è da fare sui singoli valori, ma solo sui valori medi, è necessario che, prima di poter effettuare azioni di controllo, la coda sia totalmente piena di rilevazioni registrate.

Nel caso in cui siano state raccolte sufficienti misurazioni si entra quindi nell'if (riga 18) e si passa subito alla valutazione di una nuova condizione riguardante il valore medio del livello di umidità ambientale registrato (riga 20):

- se il valore medio dell'umidità è al di sotto di un minimo prestabilito, allora è necessario procedere con l'umidificazione; dunque si attiva l'attuatore dedito all'attività di umidificazione (riga 22) e, successivamente, si svuotano le liste contenenti i valori registrati dal DHT22 e dal Capacitive, per poi azzerare anche i valori delle medie (riga da 25 a 30);
- qualora, invece, il valore medio di umidità misurata sia maggiore del massimo definito, significa che è necessario ridurre l'umidità ambientale; pertanto, si procede all'attivazione della ventola (riga 35) e successivamente si svuotano le liste contenenti i valori registrati dal

DHT22 e dal Capacitive, per poi azzerare anche i valori delle medie (riga da 38 a 43).

Infine, se il valore di umidità registrato è accettabile, cioè è compreso tra il minimo e il massimo impostati, il sistema non deve eseguire nessuna istruzione. Nel blocco except è presente unicamente il comando di pass; dunque, nel caso in cui si entri in questo blocco, il sistema non svolge nessuna attività, ma si limita ad ignorare questo caso.

Di seguito verrà analizzato il codice relativo alle attività svolte dal Capacitive Soil Moisture Sensor. [Fig.23]

```
1. ## funzione attività del sensore Capacitive Soil Moisture, per umidità suolo
2. def activity_Capacitive():
3.     global M_umidita_suolo
4.     global countIrrigazioni
5.     global altezza_vuoto
6.
7.     #try perchè potrebbe accadere che a volte la lettura del valore non vada a buon fine, e per evitare che il sistema si interrompa, tralasciamo tale misurazione
8.     try:
9.         #lettura del valore
10.        umidita = cir.capacitive()
11.        M_umidita_suolo = uq.update_M(lista_valori_capacitive,
12.        M_umidita_suolo, umidita)
13.        #controllo attività
14.        #le attività possono essere svolte solo se l'array dei valori è pieno
15.        if lista_valori_capacitive.full():
16.            #bisogna Irrigare perchè l'umidità del suolo è troppo bassa
17.            if M_umidita_suolo < Min_umidita_suolo:
18.                #accendiamo la scheda
19.                rel.relaies_attuatori(GPIO_pompa_irrigazione,
20.                time_irrigazione)
21.                #aggiorniamo il conteggio delle irrigazioni
22.                countIrrigazioni += 1
23.
24.                #azzeriamo le medie e svuotiamo gli array
25.                uq.clearList(lista_valori_capacitive)
26.                M_umidita_suolo = 0
27.                #se il numero di irrigazioni arriva a un tot allora si fertilizza e si azzerà
```

```

28.
29.         if countIrrigazioni >= numeroIrrigazioni:
30.             countIrrigazioni = 0
31.             #accendiamo la scheda
32.             rel.relaies_attuatori(GPIO_pompa_fertilizzante,
33.                                     time_fertilizzante)
34.             altezza_vuoto = cir.hcsr()
35.     except:
36.         pass

```

Codice funzione “activity_Capacitive()” [Fig.23]

Come per l’attività del DHT22, anche in questo caso, all’interno della funzione “activity Capacitive()”, si riprendono tre variabili globali (riga da 3 a 5):

- “M_umidita_suolo” contiene il valore medio dell’umidità del suolo, calcolato a partire dai valori contenuti nella lista denominata “lista_valori_capacitive”;
- “contIrrigazioni” è necessario per tenere il conteggio delle irrigazioni effettuate; in questo modo è possibile alternare un ciclo di fertilizzazione a quelli di irrigazione, secondo un rapporto prestabilito. Questo contatore viene azzerato ogni volta che si effettua una fertilizzazione;
- “altezza_vuoto” è la variabile che contiene il valore, rilevato dal sensore HC-SR04 collocato sul tappo della tanica, della distanza tra il sensore e il livello di acqua, cioè il livello di aria presente all’interno della tanica.

Per le stesse motivazioni presentate nelle attività del DHT22, anche in questa situazione è stato utilizzato un blocco try-except.

Alla riga 10, all’interno del blocco try, viene letto il valore di umidità del terreno registrato dal Capacitive Soil Moisture Sensor. Tale valore viene in seguito aggiunto alle altre misurazioni (riga 11).

A questo punto, come accadeva anche per il sensore DHT22 presentato precedentemente, si valuta la condizione per cui la lista delle medie dell’umidità del terreno sia piena (riga 15) e, in caso affermativo, si passa alla valutazione della condizione di un secondo if (riga 17).

Viene dunque valutato il valore medio di umidità del suolo registrato dal sensore:

- se tale valore risulta essere maggiore o uguale ad un determinato valore minimo, allora non sono richieste ulteriori attività;
- al contrario, qualora il valore registrato sia inferiore del minimo prestabilito, è richiesta l'attività di irrigazione; pertanto, alla riga 19, viene attivata, tramite la scheda relè, la pompa predisposta a questo scopo.

Successivamente si procede ad incrementare il contatore delle irrigazioni (riga 22), svuotando anche la lista di misurazioni effettuate dal sensore e azzerando la media dell'umidità (riga 25 e 26).

A seguito di ogni irrigazione è necessario valutare il contatore delle irrigazioni consecutive effettuate e, qualora richiesto, procedere con il ciclo di fertilizzazione. In tal caso, quindi, viene eseguita la riga 32 del codice, che attiva la pompa dedita alla fertilizzazione e il contatore delle irrigazioni consecutive viene posto nuovamente a zero (riga 30).

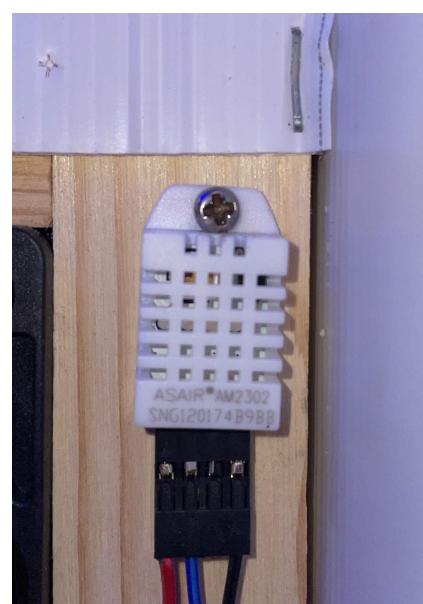
Al termine di ogni irrigazione, viene valutato il livello di acqua presente nella tanica (riga 34). Il valore registrato dal sensore HC-SR04 viene aggiornato per il file JSON. Tale file viene inviato alla tecnologia Cloud e sarà quindi visibile all'utente finale tramite l'applicativo.

3.7. IMMAGINI SERRA

In questo paragrafo verranno mostrate alcune immagini relative alla struttura della serra realizzata e ai componenti utilizzati.



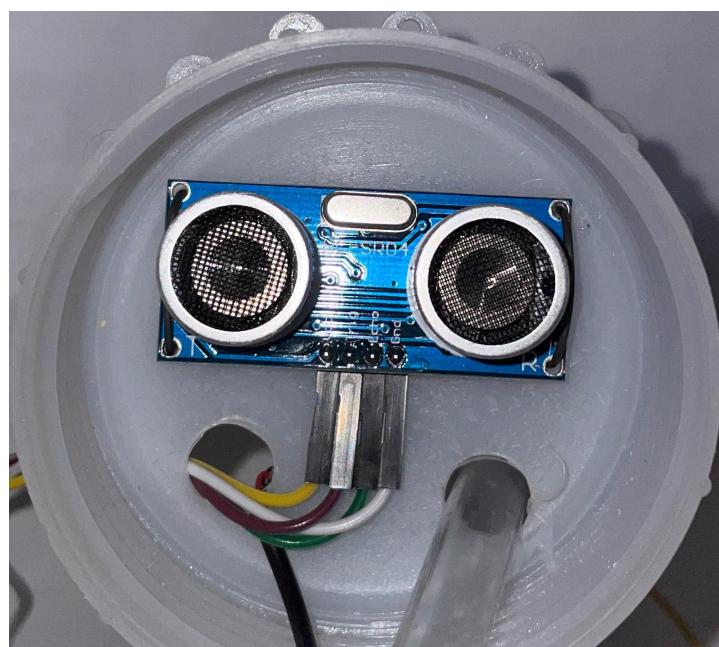
Struttura serra [Fig.24]



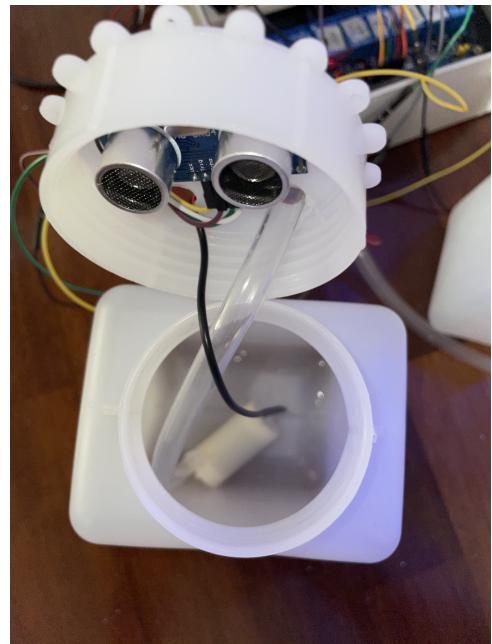
DHT22 [Fig.25]



Capacitive Soil Moisture Sensor [Fig.26]



HC-SR04 [Fig.27]



Pompa irrigazione e sensore HC-SR04 [Fig.28]



Tanica umidificazione [Fig.29]



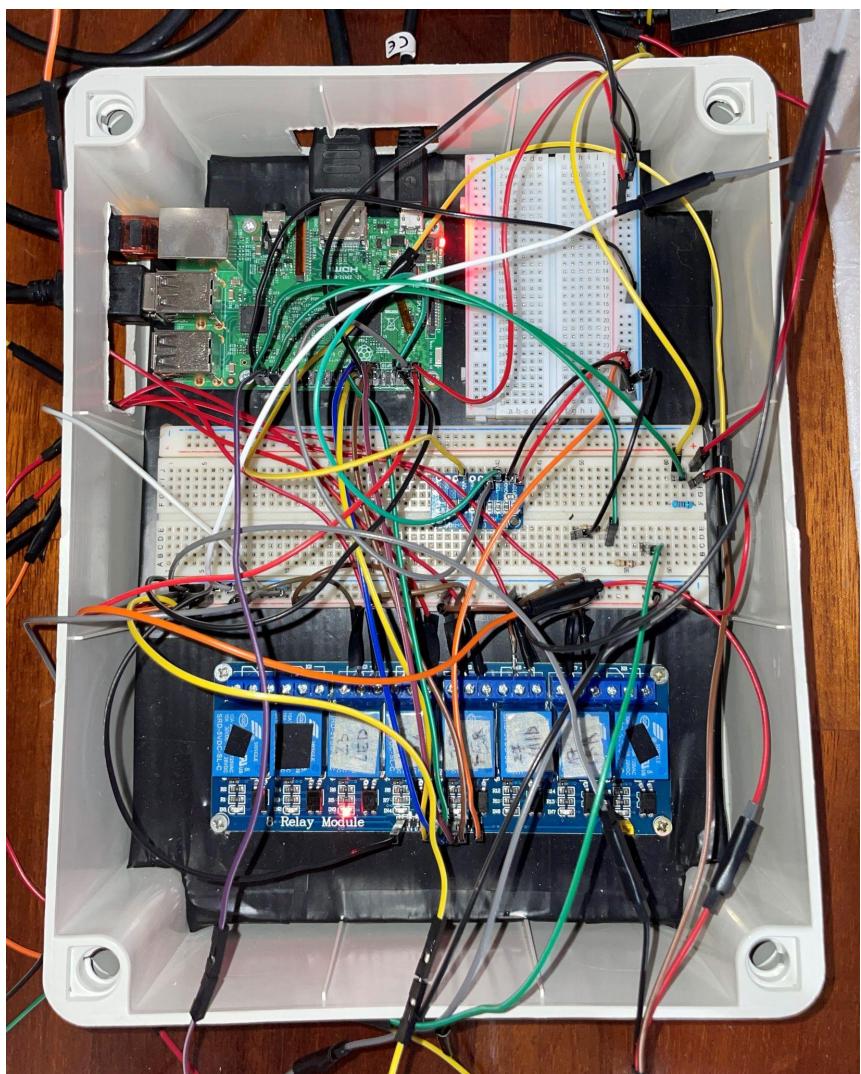
Tanica irrigazione e fertilizzazione [Fig.30]



Striscia led [Fig.31]



Ventola [Fig.32]



Scheda relè e circuiti [Fig.33]

4. TECNOLOGIA CLOUD

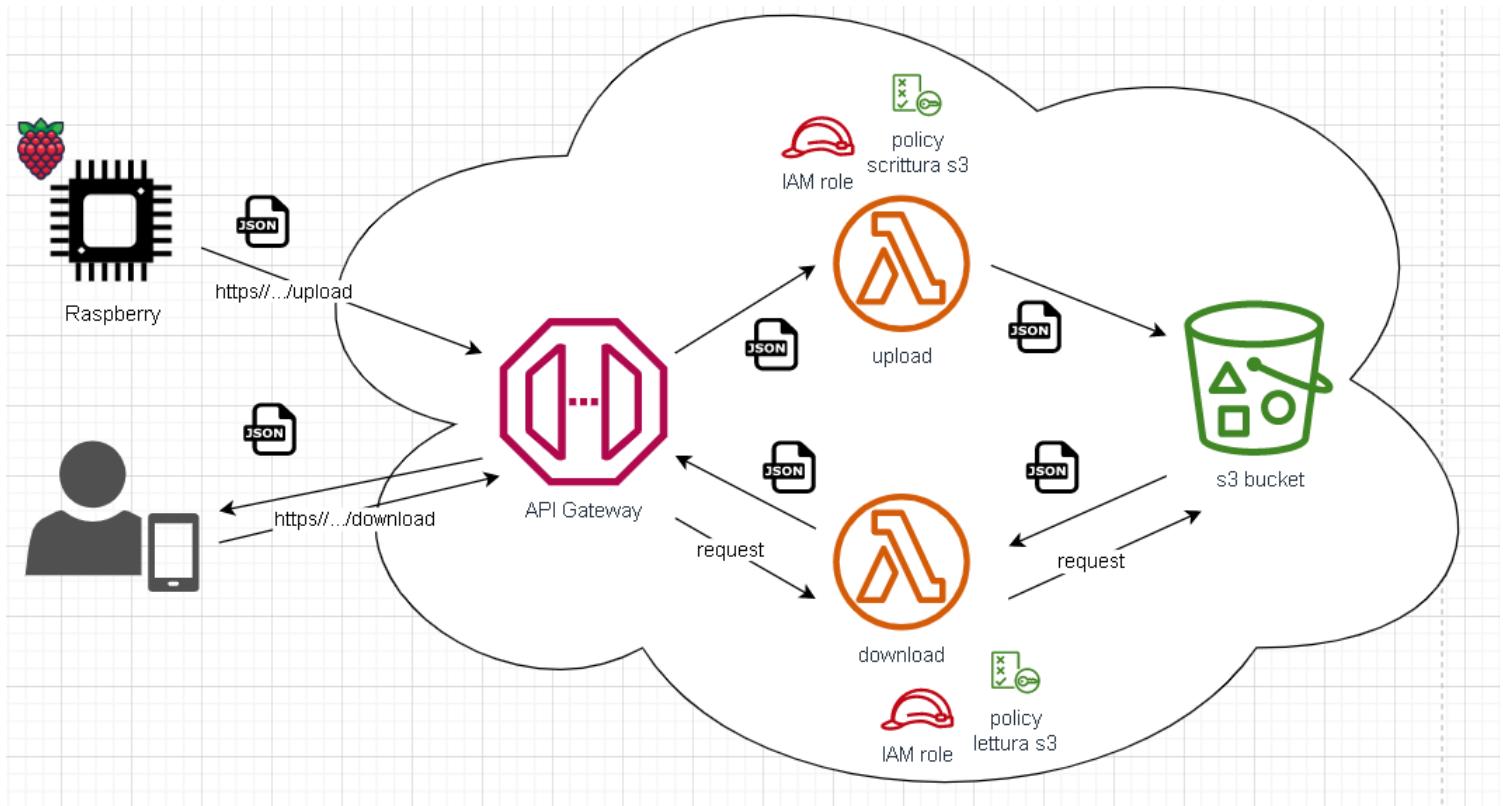
Dopo aver studiato nel dettaglio il funzionamento e la composizione del sistema Embedded, si procede ad analizzare l’architettura Cloud sviluppata attraverso le risorse fornite da Amazon Web Services.

Quando si parla di Cloud, si intende il Cloud computing, ovvero un metodo particolare di erogazione di servizi da parte di un fornitore. Con tale architettura è possibile andare ad ottenere le stesse risorse salvate nel Cloud da qualsiasi dispositivo e in qualunque parte del mondo, in quanto l’archiviazione dei dati avviene su un server remoto e non localmente nel dispositivo dell’utente. In una visione molto più semplicistica, invece di sfruttare, ad esempio, le risorse di archiviazione messe a disposizione dai nostri dispositivi, utilizziamo quelle di altri, ovvero quelle di un server.

Quindi, nel caso del progetto, si utilizza il Cloud per salvare i dati processati da Raspberry in remoto, ovvero su un server, per permettere all’applicazione mobile di ottenerli in qualsiasi momento e da qualsiasi luogo.

Inoltre il progetto sfrutta i servizi messi a disposizione da Amazon Web Services, quindi è possibile osservare ancora meglio come l’archiviazione dei dati processati dal sistema Embedded avvenga sfruttando tecnologie messe a disposizione da dispositivi esterni rispetto a quelli dei due candidati.

Di seguito, si riporta un’immagine dell’architettura Cloud che verrà sfruttata per descrivere al meglio il suo funzionamento [*Fig.34*].



Schema architettura Cloud [Fig.34]

Osservando lo schema, è possibile riconoscere i tre principali componenti del sistema:

- il sistema Embedded, in alto a sinistra rappresentato da Raspberry;
- l'utente con l'applicazione Mobile rappresentato in basso a sinistra;
- l'architettura Cloud, ovvero tutta la nuvola al centro.

Prima di proseguire con la spiegazione del flow dei dati e delle situazioni che possono verificarsi all'interno dell'architettura, si osservano in dettaglio i componenti dell'architettura Cloud:

- l'API Gateway serve per intercettare le chiamate in entrata (nel progetto in esame sono chiamate Http POST). Tali chiamate vengono poi inoltrate al sistema interno, il quale permette l'erogazione delle funzionalità richieste. L'interfaccia dell'architettura Cloud protegge tutti i componenti di back-end e controlla l'accesso al sistema;

- la funzione lambda (rappresentata in arancione) è una porzione di codice che viene eseguita per poter ottenere funzionalità specifiche;
- il bucket è un database, ovvero un sistema di archiviazione dei dati online.

Esistono inoltre all'interno dello schema altri simboli che non sono componenti dell'architettura Cloud ma, in qualche modo, ne fanno comunque parte:

- il foglio con la scritta JSON, indica un file che viene scambiato con standard JSON tra due componenti;
- l'IAM role in rosso indica il ruolo che una funzione lambda ha all'interno dell'architettura.
- la policy in verde è un file che contiene i permessi associati al ruolo di una funzione lambda. Tali permessi coincidono con le azioni che le lambda sono autorizzate ad effettuare sul bucket.

Si procede quindi a spiegare il funzionamento dell'architettura Cloud, seguendo la direzione delle frecce presenti nello schema.

La scheda Raspberry, come si è già osservato nel secondo paragrafo, processa dei dati che devono essere salvati in remoto; si esegue quindi una chiamata Http di tipo POST all'URL dell'API Gateway per ottenere tale risultato.

Il termine Http indica la tipologia di protocollo che si sta impiegando per permettere lo scambio di dati. Serve sia al client che al server per stabilire le leggi che ne regolano la comunicazione.

La chiamata è di tipo POST: essa infatti è una chiamata di scrittura; inoltre permette di proteggere i dati relativi alla serra, non rendendoli visibili dall'esterno.

Infine la chiamata viene fatta a un URL specifico, ovvero a un indirizzo che rappresenta univocamente l'API Gateway dell'architettura Cloud nella rete Internet.

Oltre all'URL, nella chiamata è necessario specificare anche l'indirizzamento “/upload”.

Si sono introdotti gli indirizzamenti per distinguere con maggiore facilità le due funzionalità che l’API Gateway deve eseguire:

- se si effettua la chiamata seguendo l’indirizzamento “/upload”, l’API capisce che l’intenzione è quella di salvare dei dati nel database remoto;
- in caso contrario, con l’indirizzamento “/download”, l’API comprende che l’intenzione è quella di ottenere dati dal server.

Nella situazione in esame, essendo l’indirizzamento uguale a “/upload”, l’API Gateway attiva la lambda upload, inoltrandole il contenuto del corpo della chiamata Http, ovvero il file JSON contenente i dati processati dal Raspberry.

Si riporta ora il codice della funzione lambda upload [Fig.35]:

```
1. import json
2. import boto3
3. import uuid
4.
5. def lambda_handler(event, context):
6.
7.     content = event['body']
8.
9.     s3 = boto3.resource("s3")
10.    s3.Bucket("raspberry-dati").put_object(Key = "dati.json", Body =
    content)
11.
12.    # TODO implement
13.    return {
14.        'statusCode': 200,
15.        'body': json.dumps('Dati salvati correttamente')
16.    }
17.
```

Codice funzione Lambda upload [Fig.35]

Nelle righe da 1 a 3 si importano le librerie necessarie e successivamente si definisce la funzione “lambda_handler()”, che viene richiamata dall’API Gateway.

Si estrae il file JSON inviato da Raspberry (riga 7) e successivamente si instaura un collegamento con il bucket s3 (riga 9). Dopodichè si inserisce il contenuto nel database, specificando il nome del file e il bucket impiegato (riga 10).

Infine, qualora il salvataggio sia stato effettuato correttamente, si ritorna al Raspberry un messaggio di corretta riuscita dell'operazione.

Nella parte inferiore dello schema, è l'utente utilizzatore dell'applicazione a richiedere i dati al server.

Nel paragrafo successivo, verrà analizzato come all'interno di un file .dart si possa effettuare una chiamata Http di tipo GET.

In questo paragrafo, invece, si tratterà di come venga gestita tale chiamata dall'architettura Cloud.

Il client effettua una chiamata Http di tipo GET (in quanto deve ottenere dei dati dal server), riferendosi all'URL dell'API Gateway con instradamento “/download”. L'API riconosce tale instradamento e attiva la funzione lambda download, andando ad eseguire il seguente codice [Fig.36]:

```
1. import json
2. import boto3
3.
4. def lambda_handler(event, context):
5.
6.     s3 = boto3.client("s3")
7.
8.     try:
9.         response = s3.get_object(Bucket = "raspberry-dati", Key =
10.             "dati.json")
11.     except:
12.         return{
13.             'statusCode':400,
14.             'body':json.dumps("Non esistono dati salvati sul bucket")
15.         }
16.
17.     # TODO implement
18.     return {
19.         'statusCode': 200,
20.         'body': response['Body'].read()
21.     }
22.
```

Codice funzione lambda download [Fig.36]

Per primo, vengono importate alcune librerie e, successivamente (riga 4), viene definita la funzione che l'API Gateway richiama.

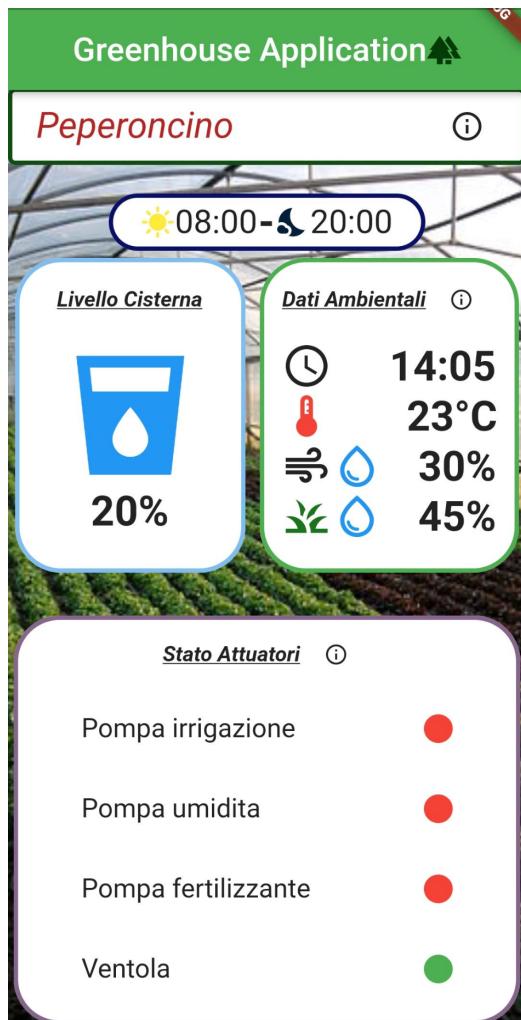
Si instaura una connessione con il bucket s3 (riga 6) e, successivamente, con un blocco try-except, si cerca di ottenere il file salvato nel bucket (riga 9).

Qualora il file non sia ancora stato creato, viene sollevata un'eccezione catturata dal blocco except (riga da 10 a 14), il quale ritorna un messaggio di errore al client.

In alternativa, se non ci sono errori nell'esecuzione del codice, viene restituito al client un messaggio avente per corpo il file JSON richiesto.

5. APPLICATIVO FLUTTER

L'applicativo sviluppato con Flutter presenta una schermata iniziale [Fig.37] che mostra, oltre alla tipologia di pianta coltivata, nel caso in esempio il peperoncino, tutte le informazioni processate dalla scheda Raspberry e riguardanti ogni aspetto della serra.



Schermata iniziale Applicazione Mobile [Fig.37]

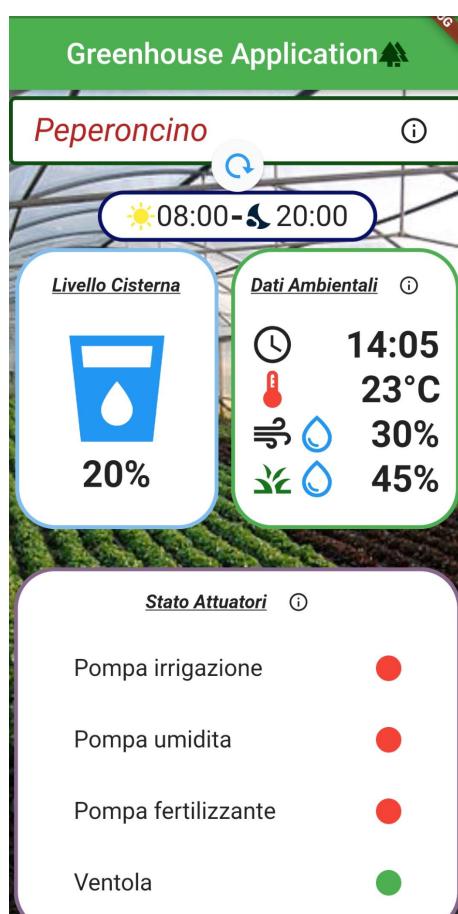
La prima informazione riportata dall'applicativo è l'orario di attività scelto per la serra; in tale situazione si è deciso, come già annunciato nei paragrafi precedenti, di utilizzare l'intervallo orario 08:00-20:00.

Viene poi riportato, sulla sinistra, la percentuale di riempimento d'acqua della tanica di irrigazione, mentre, a destra, è presente il riquadro “Dati ambientali”, all'interno del quale vengono presentati in ordine:

- l'orario in cui sono state fatte le rilevazioni mostrate;
- la temperatura dell'aria all'interno della serra, espressa in gradi centigradi;
- la percentuale di umidità presente nell'aria;
- la percentuale di umidità misurata nel terreno.

Infine, nel riquadro intitolato “Stato Attuatori”, si mostrano tutti gli attuatori utilizzati nel progetto. Questi sono affiancati da un pallino: verde nel caso in cui siano attivi, rosso in caso contrario.

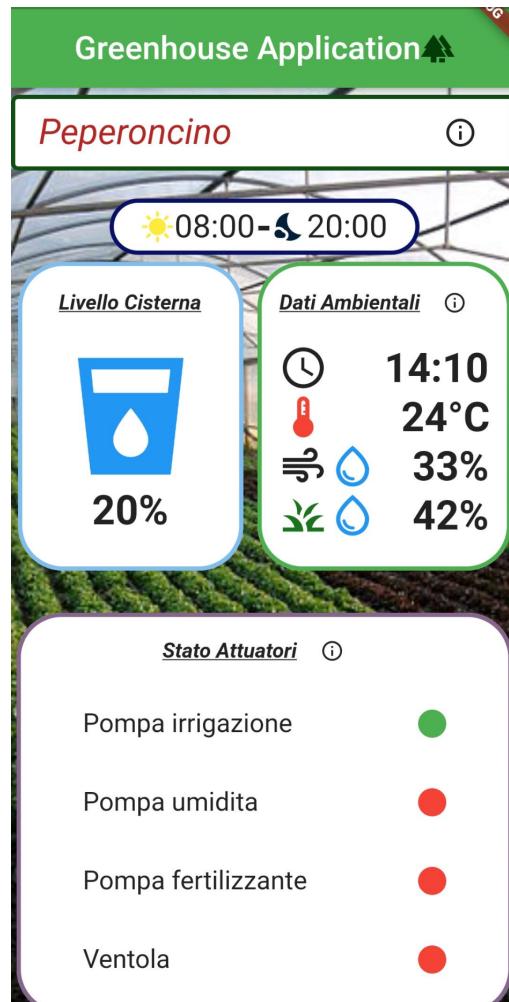
Per la schermata iniziale è stata implementata la funzione di refresh [Fig.38].



Azione di scroll sulla schermata principale [Fig.38]

Attraverso uno scroll verso il basso dello schermo, infatti, si effettua una nuova chiamata Http GET alla tecnologia Cloud, per ottenere eventuali nuovi dati caricati dalla scheda Raspberry sul bucket s3.

Nell'applicativo vengono quindi ora mostrati i dati più recenti salvati sul bucket [Fig.39].



Schermata iniziale aggiornata [Fig.39]

Si nota che, nella schermata aggiornata [Fig.39], l'orario è di cinque minuti più avanti rispetto all'immagine della schermata più vecchia [Fig.37].

Inoltre, alcuni parametri sono cambiati, così come lo stato di alcuni attuatori.

Di seguito viene mostrato il codice sviluppato per consentire all'applicativo Flutter di effettuare le chiamate Http e di salvare i dati ottenuti dalla tecnologia Cloud [Fig.40]. In particolare, si è deciso di presentare nel dettaglio solamente la funzione che si occupa dell'aggiornamento dei dati. La descrizione del codice riguardante l'implementazione dell'interfaccia grafica dell'applicazione mobile verrà invece omessa.

```
1. import 'package:http/http.dart' as http;
2.
3. const api = 'https://a9qj196ajf.execute-api.us-east-1.amazonaws.com';
4.
5. Future<List> getData() async {
6.   final response = await http.get(Uri.parse(api + '/download'));
7.
8.   if (response.statusCode == 200) {
9.     //get the array of values from api/list_races
10.    return Map<String,
11.      dynamic>.from(jsonDecode(response.body))['dati'];
12.  } else {
13.    List listavuota = [];
14.    return listavuota;
15.  }
16.}
```

Codice chiamata http applicativo Flutter [Fig.40]

Dopo aver importato il package necessario per effettuare delle chiamate Http e averlo rinominato per comodità “http” (riga 1), si passa alla definizione dell’api a cui collegarsi e verso il quale rivolgere le chiamate Http di GET (riga 3).

Successivamente, viene definita una funzione “getData()” che ritorna una Future<List> in modalità asincrona. Ciò è necessario per poter renderizzare in qualsiasi caso le componenti grafiche dell'applicazione, anche qualora i dati dovessero tardare ad arrivare. L'ottenimento dei dati dall'architettura Cloud e la renderizzazione dell'applicativo avvengono quindi in maniera asincrona.

L'esito della chiamata Http di tipo GET, effettuata all'instradamento “/download” dell'API, viene salvato all'interno della variabile “response” definita alla riga 6 del codice.

A questo punto viene effettuato un controllo sulla response:

- se lo statusCode della risposta è 200, significa che la chiamata Http ha esito positivo, dunque è possibile restituire i dati ottenuti attraverso una Map;
- qualora invece lo statusCode della response sia diverso da 200, significa che la chiamata Http non è terminata correttamente, dunque non sarà possibile restituire i dati; per questa ragione verrà restituita una lista vuota (righe da 11 a 13).

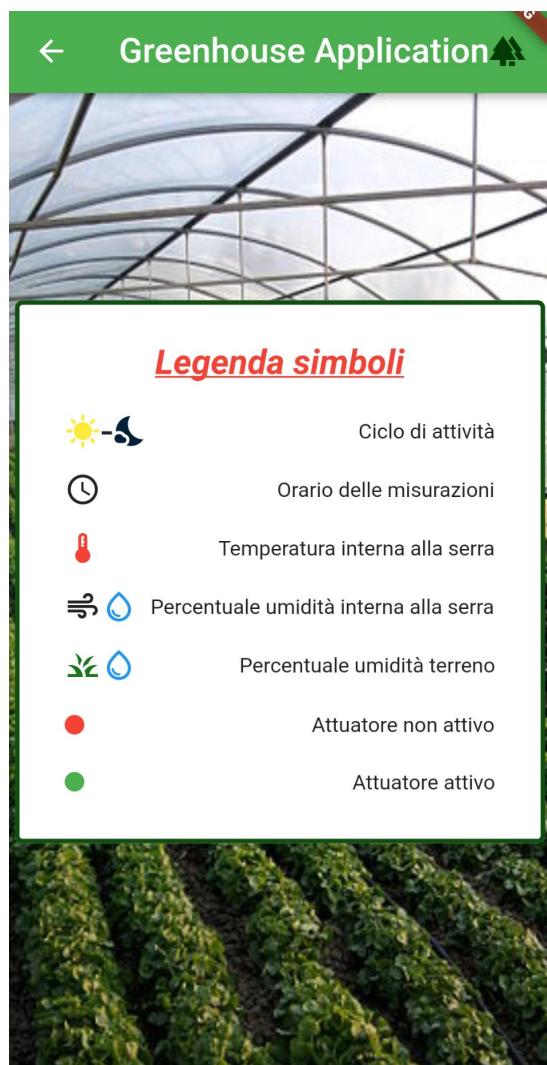
Accanto alla scritta “Peperoncino” della schermata iniziale è stata collocata una piccola icona che, se cliccata, porta ad una schermata secondaria contenente maggiori informazioni sulla tipologia di pianta coltivata nella serra [Fig.41].



Informazioni pianta [Fig.41]

Similmente, anche accanto al riquadro intitolato “Dati Ambientali” e a quello intitolato “Stato Attuatori” sono state inserite le medesime icone.

Queste ultime, se cliccate, aprono una nuova schermata secondaria, che riporta una breve legenda dei simboli utilizzati nell’applicativo appena mostrato [Fig.42].



Legenda simboli applicazione Mobile [Fig.42]

In ogni momento è possibile uscire dalle schermate secondarie e tornare in quella principale, attraverso un semplice clic sulla freccia posta in alto a sinistra dello schermo.

6. BIBLIOGRAFIA E SITOGRAFIA

Link alla repository di Github dove sono presenti tutti i codici e i modelli utilizzati per il progetto di tesi.

Il file readme permette di orientarsi al meglio all'interno della repository condivisa.

<https://github.com/loretor/Tesi>

Riferimenti bibliografici e sitografia citati nella tesi:

- *Raspberry Pi Foundation, Raspberry Pi Foundation Strategy 2016-2018*, (2016/2018).
- devAcademy, <https://devacademy.it/raspberry-pi/>.
- Wikipedia, https://it.wikipedia.org/wiki/Umidit%C3%A0_relativa
- Wikipedia, <https://it.wikipedia.org/wiki/I%C2%B2C>
- Wikipedia, <https://it.wikipedia.org/wiki/Attuatore>

Sitografia utilizzata per la documentazione relativa alla scheda Raspberry e ai componenti ad essa collegati:

- Pin scheda Raspberry: <https://pinout.xyz/>
- Sensore HC-SR04:
<https://theiphut.com/blogs/raspberry-pi-tutorials/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi>
- Sensore DHT22:
<https://www.meccanismocomplesso.org/dht11-e-dht22-misurare-umidita-e-temperatura-con-raspberry/>
- Capacitive Soil Moisture Sensor:
<https://www.switchdoc.com/2020/06/tutorial-capacitive-moisture-sensor-grove/>

- Sensori analogici:
<https://it.jf-paredes.pt/types-analog-digital-sensors-with-applications>
- ADS1115:
 - <https://www.adriirobot.it/convertitore-adc-16-bit-ads1115/>
 - <https://www.giuseppecaccavale.it/raspberry/utilizzare-un-adc-ads1115-con-raspberry/>
- I²C: <https://it.wikipedia.org/wiki/I%C2%B2C>
- Scheda relè:
 - <https://www.html.it/pag/367603/rele-e-libreria-gpiozero>
 - <https://elettricistalodi.net/come-funziona-rele/>
- Utilizzo dei thread: [https://it.wikipedia.org/wiki/Thread_\(informatica\)](https://it.wikipedia.org/wiki/Thread_(informatica))
- Tecnologia cloud:
<https://www.cloudflare.com/it-it/learning/cloud/what-is-the-cloud/>
- API gateway:
<https://www.redhat.com/it/topics/api/what-does-an-api-gateway-do>

Librerie impiegate nel codice del sistema Embedded:

- <https://github.com/adafruit/DHT-sensor-library> libreria dht22
- https://github.com/adafruit/Adafruit_ADS1X15 libreria ads1115
- https://github.com/adafruit/Adafruit_Blinka librerie board e busio
- <https://docs.python.org/3/library/time.html> time library
- <https://pypi.org/project/RPi.GPIO/> Rpi.GPIO

Librerie impiegate nel codice delle funzioni AWS:

- <https://docs.python.org/3/library/json.html> json library
- <https://pypi.org/project/boto3/> boto3 library
- <https://docs.python.org/3/library/uuid.html> uuid library

Librerie impiegate nel codice Dart di Flutter:

- <https://pub.dev/packages/http> http library

7. RINGRAZIAMENTI

In conclusione di questa relazione vogliamo cogliere l'occasione per esprimere la nostra gratitudine nei confronti del professore Davide Brugali, relatore di questa tesi, ringraziandolo per la disponibilità mostrata e per i preziosi consigli forniti.

Un comune ringraziamento è destinato all'amico Andrea Ferrari, per la disponibilità e l'apprezzatissimo aiuto fornito durante la realizzazione strutturale della serra oltre che per le conoscenze botaniche gentilmente condivise; senza di lui non sarebbe stato possibile realizzare il progetto.

Approfitto di questo momento per ringraziare:

La mia famiglia, per avermi sempre sostenuto nei momenti di difficoltà e nell'intero percorso di studi, dai primi anni fino ad arrivare a quelli universitari.

I colleghi conosciuti in questi tre anni, con i quali ho avuto il piacere non solo di studiare e di collaborare in progetti curricolari ma anche di condividere esperienze al di fuori del contesto didattico.

Gli amici che mi sono stati vicini e che hanno concesso momenti di spensieratezza, fondamentali nelle parentesi più critiche e di maggiore pressione.

Infine, un doveroso e sincero ringraziamento spetta al collega, nonché amico di lunga data, Lorenzo Torri, con il quale ho condiviso con gioia questi anni universitari e con cui ho potuto realizzare con entusiasmo il progetto di tesi presentato.

*Grazie infinite a tutti voi,
Matteo Carminati*

In conclusione di tale progetto, è doveroso ringraziare tutti coloro che mi hanno accompagnato durante questi tre anni presso l'Università degli Studi di Bergamo.

In primis, dedico questo traguardo ai miei genitori, Lia e Cesare, e a mio fratello Alessandro, che, con il loro supporto e affetto quotidiano, mi hanno sempre sostenuto durante tutti i miei studi e nella vita extra scolastica.

Estendo i miei ringraziamenti anche agli altri componenti della mia famiglia, che mi hanno sempre fatto sentire amato e benvoluto. Ringrazio quindi i miei nonni, i miei zii ed i miei unici due cugini, Chiara e Marco.

Ci tengo inoltre a ringraziare la mia fidanzata, Anna, per tutto il tempo dedicatomi in questi ultimi anni e per il suo continuo amore e sostegno.

Grazie anche ai miei amici di sempre, per tutte le avventure vissute ed i bei ricordi che porterò ovunque con me.

Sarò sempre grato all'Università degli Studi di Bergamo, che mi ha permesso di maturare e crescere e di appassionarmi a tutte le discipline legate all'informatica.

Infine, una dedica speciale a mio nonno Franco, venuto a mancare durante il mio percorso di studi.

*Grazie di cuore
Lorenzo Torri*