Corso di Sistemi Embedded e Internet of Things

Smart Temperature Monitoring System

Progettazione e costruzione di un *sistema di controllo di temperatura* ed implementazione di una Macchina a Stati Finiti

Luca Ponseggi Jacopo Turchi Luca Venturini
Federico Bagattoni

15 gennaio 2025

Indice

Introduzione

L'obbiettivo del progetto finale del corso è costruire ed implementare un sistema di monitoraggio della temperatura che cambia in autonomia l'apertura di una finestra controllata da servo motore. Il sistema consente all'utente di aprire manualmente la finestra e fornisce un interfacia web per la visualizzazione dei dati.

Per lo sviluppo verrà utilizzata la piattaforma **Arduino Uno** e **ESP-32** con sensore di temperatura. Per quanto riguarda il backend viene utilizzato il framework Vertx.io consigliato a lezione.

Capitolo 1

Unità di controllo

Il sistema di controllo, codificato in Java, è totalmente basato sul framework Vertx.io che consente la creazione di più unità di elaborazione dette *verticles* (*vertici*) che vengono eseguite ciascuna in un thread separato.
Per far comunicare ciascun vertice il framework fornisce l'*Event Bus*, un sistema di messaggistica che permette di scambiare messaggi tra vertici in diverse maniere (request-reply, publish-subscribe...). L'Event Bus viene usato per la trasmissione dei dati all'interno del modello.

Il vantaggio dei vertici è quello di permettere comunicazione asincrona tra il backend e le varie componenti, delegando al framework ed alla JVM lo scheduling, la concorrenza e la gestione delle comunicazioni.
Infatti ogni vertice rappresenta una task o un componente del modello.

Il modello conserva i dati in apposite classi ed un vertice implementato come una macchina a stati finiti prende decisioni riguardo al comportamento del sistema. Altri vertici servono solo per scambiare comunicazioni tra le altre unità del sistema ed il backend.

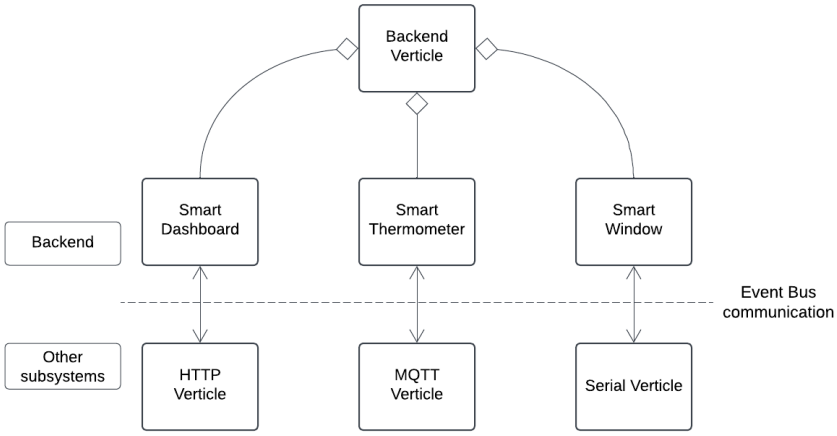

Figura 1.1: diagramma di funzionamento dell'unità di controllo con comunicazione con i componenti esterni.

1.1 Task del sistema

I *verticles* all'interno dell'unità di controllo possono essere visti come delle task indipendenti che vengono eseguite dal framework e che discorrono tra di loro.
Potrebbe sembrare esagerata la scelta di avere così tanti thread attivi ma è importante evidenziare come ciascun componente ha bisongo di una sua esecuzione indipendente al fine di potersi aggiornare dall'esterno e di isolare il comportamento di trasmissione e ricezione dall'unità principale.
Inoltre la comunicazione con *Event Bus* ferma l'esecuzione del thread quindi è necessario che il loop principale non venga mai fermato da comunicazioni in arrivo.
Questa soluzione permette isolamento e maggiore utilizzo di interfacce che contribuiscono a snellire il codice e renderlo maggiormente leggibile.

- **Backend verticle**: questo vertice è l'automa principale dell'unità di controllo. Esso prende decisioni e cambia il suo stato in base alla temperatura rilevata e modifica conseguentemente anche l'apertura della finestra. E' implementato come una macchina a stati finiti di cui si può vedere il comportamento dettagliato nell'immagine.

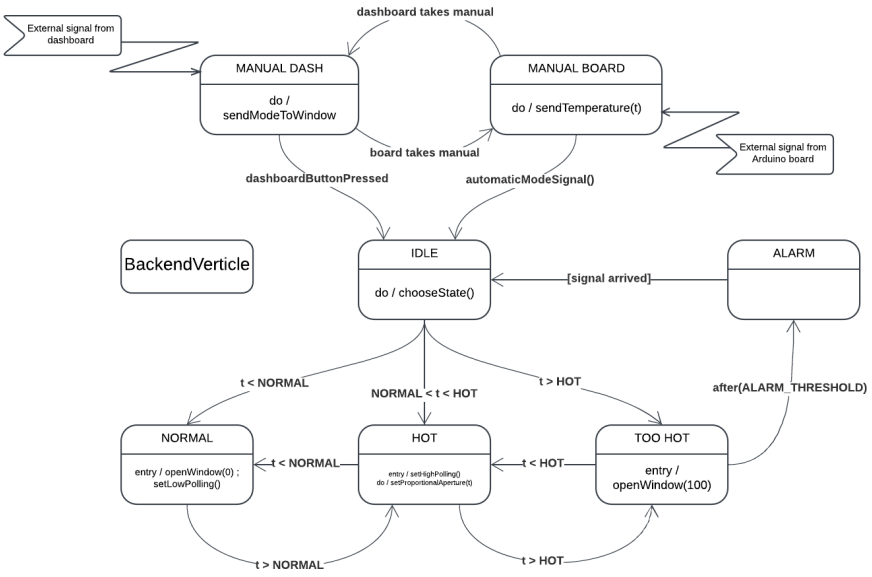

Figura 1.2: diagramma degli stati di *Backend verticle*

- **Smart Thermometer Verticle**: contenuto in BackendVerticle esso funge da endpoint per i valori di temperatura che arrivano, attraverso l'Event Bus, dal componente deputato alla comunicazione MQTT. Questa classe è l'astrazione locale del sistema di misurazione di temperatura ESP-32.
- **Smart Window Verticle**: questa classe è l'astrazione del sistema di controllo della finestra Arduino. Conserva ampiezza della finestra ed invia il nuovo parametro quando viene modificato. Questo vertice fa anch'esso parte di BackendVerticle.
- **Smart Dashboard Verticle**: rappresenta la dashboard web in cui vengono visualizzati i dati. Questa non è particolarmente complicata perché i dati vengono richiesti e forniti tramite l'utilizzo di funzionalità del framework quindi non vi è nessuna particolare astrazione.

Oltre ai vertici del backend sono stati implementati altri vertici che garantiscono la comunicazione con le unità esterne ed il cui funzionamento verrà spiegato nel dettaglio nei capitoli successivi:

- **MQTT Client Verticle**: questo verticle gestisce la connessione al broker MQTT e funge da ponte tra il sistema di monitoraggio della temperatura e l'unita di controllo. La sua funzione principale è inoltrare i messaggi di temperatura ricevuti dall'ESP32 al backend tramite l'Event Bus di Vert.x e inviare le frequenze di aggiornamento dal backend all'ESP32 tramite MQTT.

- **HTTP Verticle**: questo vertice è l'effettiva implementazione del web server che rende disponibile la dashboard. Comunica tramite event bus con *Smart Dashboard Verticle* richiedendo nuovi parametri da visualizzare e inviando i dati nella modalità manuale.
- **Serial Verticle**: comunica via seriale con Arduino e tramite event bus con *Smart Window Verticle* trasformando i messaggi da un formato ad un altro.

1.2 La Macchina a Stati Finiti

Come indicato nel capitolo precedente la Main task gestisce il funzionamento dell'intero sistema. Questa è una macchina a stati finiti che reagisce alla variazione dei parametri del sistema e cambia il suo stato di conseguenza.

La MSF ha 6 stati. All'avvio è posizionata nello stato **IDLE**, questo stato è usato come passaggio sia all'inizializzazione sia quando si passa da manuale ad automatico. Infatti questo stato permette di decidere in che stato posizionarsi.

1.2.1 La modalità automatica

Gli stati corrispondenti alla modalità automatica sono **NORMAL**, **HOT**, **TOO HOT**, **ALARM**. La macchina salta da uno stato all'altro in base alle soglie di temperatura in cui è configurata.
Nello stato **HOT** modifica l'apertura della finestra in base alla temperatura.
Rimane indefinitvamente nello stato **ALARM** di finché il sistema non viene ripristinato.

1.2.2 La modalità manuale

La modalità manuale consiste di due stati in quanto *una discrepanza con la descrizione delle specifiche creava un conflitto di interessi tra i componenti del sistema*. Gli stati sono **MANUAL ARDUINO** e **MANUAL DASHBOARD** che rappresentano la modalità manuale per ciascun componente che può abilitarla. Questo facilita la sincronizzazione ed evita che i sistemi, entrambi in manuale, abbiamo comportamenti non indicati nelle specifiche.

- Quando il sistema è in **MANUAL ARDUINO** il sotto-sistema Arduino è in controllo dell'apertura della finestra e la dashboard web si comporterà come fosse in modalità automatica (esattamente come prima che Arduino entrasse in modalità manuale)
- Quando il sistema è in **MANUAL DASHBOARD** il sotto-sistema della dashboard web è in controllo dell'apertura della finestra. Analogamente al punto precedente il sistema Arduino è ignaro della situazione e si comporta come se fosse in modalità automatica.

L'implementazione che abbiamo fornito è la nostra interpretazione delle specifiche permettendo la massima fedeltà per i due sotto-sistemi Web e Arduino aggiungendo un solo stato al sistema senza violare nessuna altra regola.

Capitolo 2

Dashboard

La Dashboard racchiude un backend in Java e un frontend dinamico che utilizza JavaScript. Il sistema è progettato per offrire un'interfaccia interattiva per il monitoraggio della temperatura in tempo reale ed il controllo da remoto.

2.1 Il Backend: Architettura in Vert.x

Il backend del progetto è stato implementato in Java utilizzando la libreria Vert.x.

2.1.1 Router e gestione delle rotte

L'oggetto `HTTPVerticle` definisce un Router che gestisce diverse richieste HTTP:

- **GET /api/system-state**: Fornisce lo stato attuale del sistema, in modo da aggiornare i dati nell'interfaccia.
- **POST /api/manual-mode**: Permette di impostare manualmente il livello di apertura della finestra.
- **POST /api/switch-mode**: Serve per cambiare la modalità operativa tra manuale e automatica.
- **POST /api/resolve-alarm**: Risolve lo stato di allarme del sistema.

Negli ultimi tre endpoint sono state utilizzate richieste POST e non GET in quanto gestiscono meglio l'invio di strutture dati al server (JSON) in quanto sono contenute nel body.

2.1.2 Gestione dello stato del sistema

L'oggetto `SystemState` rappresenta il cuore del backend, fungendo da modello dati. Include tutti i gli attributi che vengono visualizzati nella Dashboard, come

ad esempio la temperatura, la modalità operativa della finestra e la percentuale di apertura.

2.1.3 Comunicazione asincrona con l'event bus

Vert.x utilizza un Event Bus, un meccanismo di comunicazione per gestire eventi in modo asincrono. Nell'Event Bus il sistema riceve i messaggi dallo *Smart Dashboard Verticle* e ne aggiorna lo stato, i dati elaborati vengono poi inoltrati al frontend per l'aggiornamento in tempo reale.

2.2 IL Frontend: Interfaccia Grafica con JavaScript

Il frontend è responsabile della visualizzazione dei dati e dell'interazione con l'utente. È costruito utilizzando **HTML** e **CSS** per la struttura e lo stile e **JavaScript** per la logica dinamica e l'aggiornamento del contenuto.

2.2.1 Aggiornamento Dinamico dello Stato

La funzione `fetchSystemState` nel file JavaScript invia una richiesta GET al backend per recuperare lo stato del sistema e aggiornare la dashboard

2.2.2 Grafico delle Temperature

Per rappresentare graficamente lo storico delle temperature, viene utilizzata la libreria Chart.js. Il grafico si aggiorna con le ultime 20 misurazioni ricevute dal backend.

2.2.3 Interazione con l'Utente

L'interfaccia consente di:

- Visualizzare i dati principali ed il grafico delle temperature
- Cambiare modalità operativa (manuale o automatica).
- Impostare manualmente l'apertura della finestra.
- Risolvere stati di allarme.

Capitolo 3

Window Controller Subsystem

Il Window Controller Subsystem è stato implementato su una scheda Arduino UNO. Rappresenta il cuore del controllo meccanico della finestra, consentendone l'apertura e la chiusura in due modalità: automatica e manuale. Garantisce il controllo da parte degli operatori tramite uno schermo LCD, un bottone che consente il passaggio di modalità e un potenziometro per regolare l'apertura della finestra.

3.1 Model

Sono presenti 4 oggetti:

- **HWPlatform**: incapsula e nasconde dettagli dell'implementazione hardware.
- **OperatorPanel**: permette l'uso dell'LCD e la rivelazione della pressione dei pulsanti.
- **Dashboard**: Permette la comunicazione tramite seriale con il Serial Verticle. E' usato per notificare il verticle di nuove informazioni e per processare i messaggi che riceve da esso.
- **WindowController**: Rappresenta il controller. Permette di controllare l'apertura della finestra agendo sul suo motore e di cambiare la modalità di operazione del controller (Automatica o Manuale).

3.2 Task del sistema

L'architettura prevede 2 tasks:

- **WindowControllingTask**:
  - Task per controllare la finestra in base alla modalità selezionata.
  - Usa il WindowController, l'OperatorPanel e la Dashboard.
  - Period: 100ms

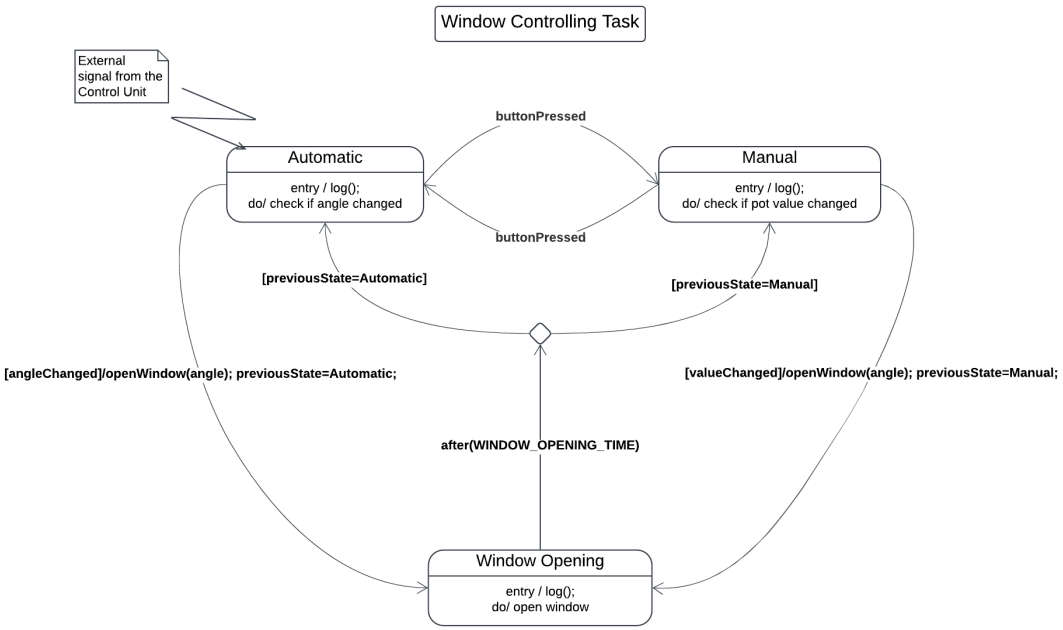

Figura 3.1: diagramma degli stati di *Window Controlling Task*

- **MessageTask**:
  - Task per regolare la ricezione e l'invio dei messaggi. Tiene inoltre traccia della modalità corrente del sistema.
  - Usa il WindowController e la Dashboard.
  - Period: 100ms

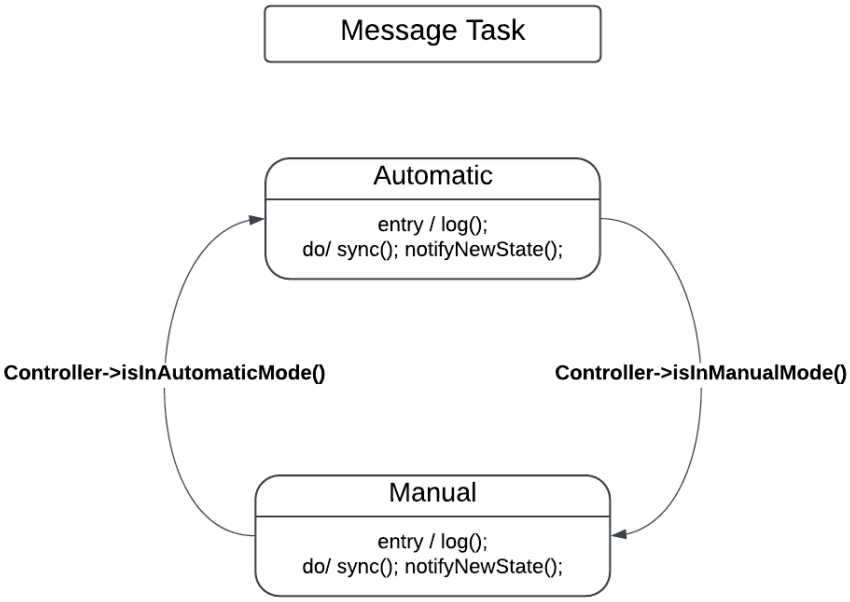

Figura 3.2: diagramma degli stati di *Message Task*

3.3 Circuito

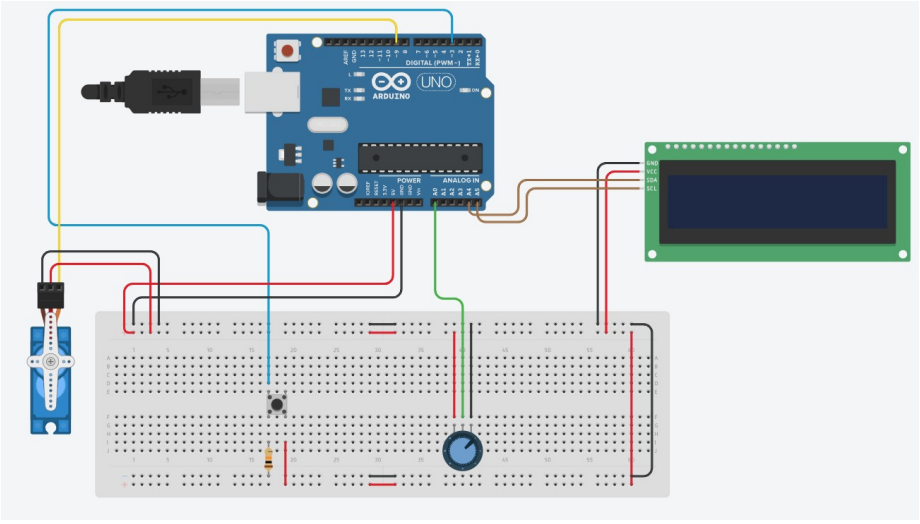

Figura 3.3: Circuito dell'*Arduino UNO*

Capitolo 4

Sistema di monitoraggio della temperatura

Questa unità consiste in un sistema di monitoraggio della temperatura basato su un'architettura di comunicazione MQTT e su un design software strutturato come una macchina a stati finiti. L'obiettivo principale è quello di acquisire dati da un sensore di temperatura ed inviarli a un broker MQTT per l'elaborazione remota.

4.1 Architettura

Il sistema si basa su una architettura **task based** ovvero che il lavoro viene suddiviso in due task **TemperatureTask** e **ObserverTask**, entrambe implementano una **final state machine** per gestire le diverse condizioni operative e le transizioni tra stati.

4.1.1 Temperature Task

Questa task si occupa di leggere i dati dal sensore ed inviarli ad un broker Mqtt ad una frequenza configurabile tramite l'inserimento di dati all'interno di una coda che permette la comunicazione tra i task. Questa separazione logica consente una maggiore modularità e una gestione indipendente delle funzionalità.

Stati:

- **IDLE**: Stato di attesa, verifica il tempo trascorso rispetto alla frequenza configurata.
- **SENDING**: Invia il valore di temperatura corrente al broker MQTT.
- **CONNECTING**: Aspetta che la connessione al broker venga ristabilita.

Transizioni :

- Da **IDLE** a **SENDING** se è trascorso il tempo configurato.
- Da **IDLE** a **CONNECTING** se il broker MQTT non è connesso.
- Da **CONNECTING** a **IDLE** se la connessione viene ristabilita.

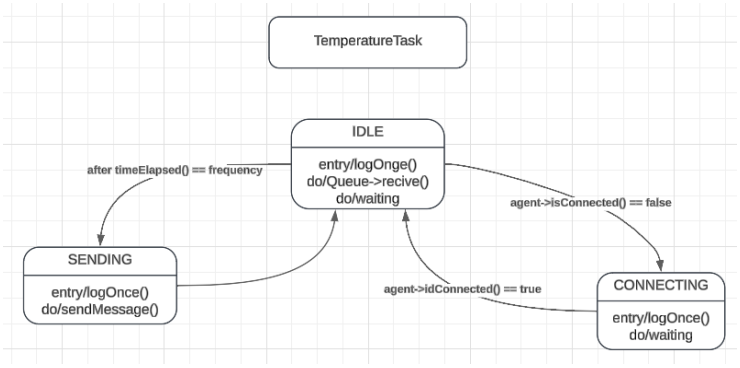

Figura 4.1: Diagramma di temperature task

4.1.2 Observer Task

Questa task ha la funzione di ricevere i messaggi contenenti la frequenza con cui viene richiesta la temperatura, nterpretarli e comunicarli alla Temperature Task tramite una coda condivisa. .

Stati:

- **IDLE**:Stato di attesa, verifica la connessione al broker MQTT e la presenza di nuovi messaggi
- **COMPUTIG**: Riceve il messaggio, lo elabora e inserisce la frequenza ricevuta nella coda condivisa.
- **RECONNECTING**: Tenta di ristabilire la connessione al broker Mqtt.

Transizioni :

- Da **IDLE** a **COMPUTING** se è stato ricevuto un messaggio.
- Da **IDLE** a **RECONNECTING** se il broker MQTT non è connesso.
- Da **RECONNECTING** a **IDLE** se la connessione viene ristabilita.

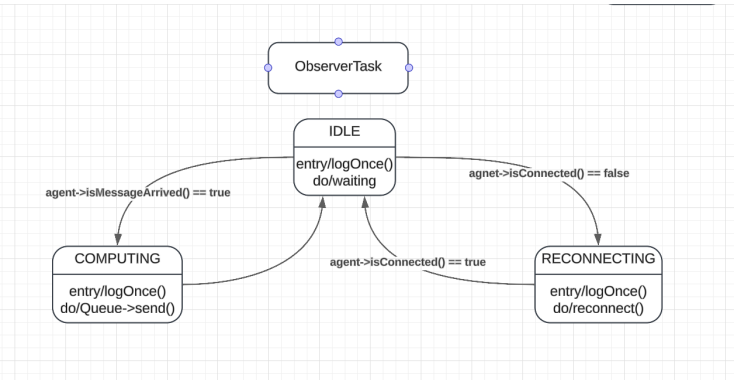

Figura 4.2: diagramma di Observer task

4.2 Funzionamento specifico

Il sistema opera acquisendo i dati di temperatura da un sensore tramite la **Temperature Task**, che si occupa anche di inviare i valori letti al broker MQTT con una frequenza configurabile. Questa frequenza viene dinamicamente aggiornata dalla **Observer Task**, che monitora un topic MQTT specifico per ricevere istruzioni. In caso di perdita di connessione, sarà la **Observer Task** che gestirà la riconnessione con il server mentre la **Temperature Task** rimarrà in uno stato di attesa.

4.3 Implementazione

4.3.1 Task

Le nostre task sono state sviluppate a partire dal funzionamento delle task in FreeRTOS che permette di gestire diverse attività in modo concorrente, dandole anche la possibilita di farle comunicare tra di loro. Ogni task infatti presenta un metodo statico **tick** il quale viene inserito in una xTaskCreate che permette di registrare questa funzione come una task di FreeRTOS, assegnandole uno stack, una priorità e un handle per gestirla, questo metodo è implementato attraverso un loop infinito contente il funzionamento della task stessa. Il passaggio tra le task viene gestito automaticamente dallo scheduler di FreeRTOS in base alle priorità delle task e ai ritardi definiti con vTaskDelay.

4.3.2 Connessione Mqtt

Per la connessione al broker Mqtt invece è stata utilizzat la libreria **PubSubClient** la quale permette di implementare il protocollo Mqtt grazie ad un client che da la possibilita di connettersi ad un server, sottoscriversi ad uno o più topic, gestire messaggi pubblicati su questi e pubblicare messaggi.Nel nostro caso

utilizziamo due topic, uno dove questo sottosistema riceve la frequenza che deve utilizzare ed uno dove viene pubblicata la temperatura rilevata.

4.4 Circuito

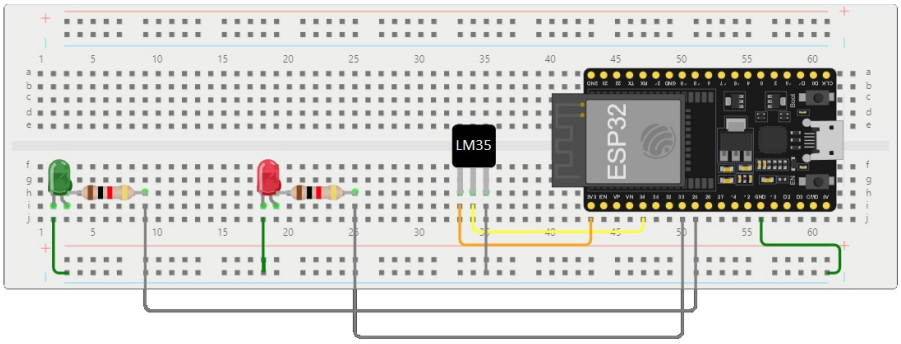

Figura 4.3: Circuito dell'*ESP32*