UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Water Tanks Control with ESP8266

Replication guide

di Lauterio Davide

Data: 15/10/2022

Made by: Lauterio Davide

Contents

1 Introduzione

Lo scopo di questa guida è fornire istruzioni dettagliate su come replicare il mio progetto di laboratorio per il corso di Progettazione di Sistemi Embedded. Il progetto riguarda il controllo remoto e la notifica dello stato di un sistema di serbatoi d'acqua per l'irrigazione di un giardino.

Questo sistema è controllato con un ESP8266 WeMos Node MCU, una versatile scheda microcontrollore basata sul modulo Wi-Fi ESP8266. Offre connettività Wi-Fi, oltre ai classici pin GPIO per interfacciarsi con sensori e attuatori ed interfaccia seriale, ed è programmabile in C++ utilizzando l'IDE di Arduino.

L'ESP collegato in Wi-Fi invierà i cambiamenti nello stato codificati in un byte sia via seriale che tramite il protocollo MQTT ad un broker remoto.

Attualmente il broker è fornito da MyQTTHub [7.1], che offre un piano gratuito per effettuare testing in modo abbastanza rapido, su un server remoto.

1.1 Requisiti di progetto

Il sistema che si vuole controllare è quello in Fig. 1.1. In un primo serbatoio (T1) viene raccolta l'acqua piovana, questo semplicemente trabocca quando è pieno; quest'acqua viene riversata in un secondo serbatoio (T2), più piccolo, tramite una valvola elettromeccanica (V) che può essere impostata su due stati, aperta o chiusa. In T2 c'è una pompa sommersa (P) e due sensori di livello dell'acqua, uno per il serbatoio vuoto (T2E) e l'altro per il serbatoio pieno (T2F). Attivando la pompa, è possibile riempire il terzo serbatoio (T3), anche qui c'è un sensore di livello (T3F), che segnala quando il serbatoio è pieno.

La logica di controllo deve rispettare le seguenti considerazioni:

- L'obiettivo principale è mantenere T3 pieno.
- È necessario evitare che T2 e T3 strabordino.
- Tutti i sensori di livello forniscono un segnale HIGH quando l'acqua è sopra il loro livello, LOW altrimenti
- La pompa P non deve mai funzionare se non è sommersa, ossia quando T2E == HIGH.
- P non deve mai funzionare per più di 5 minuti continuativi e richiede un tempo di raffreddamento pari al tempo in cui è stata attiva.
- La valvola elettromeccanica viene aperta da un segnale HIGH.
- In caso T2 sia completamente vuoto, con la valvola aperta, ci vogliono 30 secondi perché l'acqua raggiunga il livello T2E.

- T3 verrà svuotato da un agente esterno e non va tenuto in considerazione questo evento se non per il fatto che qualcosa potrebbe far variare lo stato del sensore T3F.

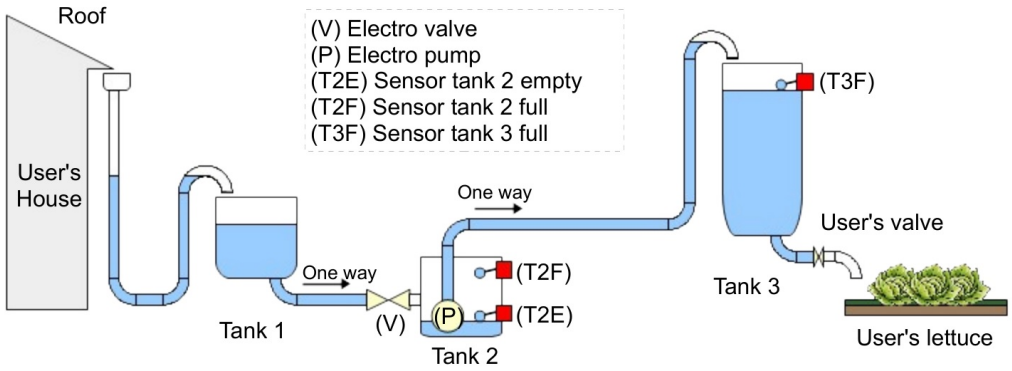

Figure 1.1: Schema del Sistema.

1.1.1 Connettività

La scheda microcontrollore utilizzata nel progetto dispone di un modulo Wi-Fi, il quale offre la possibilità di sfruttare la connessione wireless per inviare e ricevere messaggi dal broker MQTT, sia in locale che in remoto. Questo permette di avere un costante aggiornamento sullo stato del sistema e di comunicare con la scheda senza l'utilizzo di cavi di connessione. In questo modo, si potrebbe anche abilitare la possibilità di gestire il sistema anche da remoto, senza essere fisicamente presenti.

1.2 Struttura della guida

Questa guida è suddivisa in quattro capitoli, escludendo questa breve introduzione, ognuno dei quali descrive una fase specifica del processo di sviluppo del sistema.

Il primo capitolo, "Programmazione della scheda", illustra la definizione delle variabili, la costruzione della funzione Setup() e l'organizzazione del Loop() necessarie per modellare il sistema ed il suo funzionamento.

Il secondo capitolo, "Integrazione nella simulazione in Processing", illustra come integrare la scheda programmata in una simulazione in Processing, utilizzando la connessione seriale per scambiare dati tra la scheda e il programma.

Il terzo capitolo, "Realizzazione del bot Telegram", spiega come creare un bot Telegram per ricevere notifiche sullo stato del sistema monitorato e potenzialmente interagirvi.

Infine, il quarto capitolo, "Conclusioni e migliorie apportabili", riassume il lavoro svolto e fornisce alcune idee per eventuali miglioramenti futuri del sistema.

2 Struttura preliminare

2.1 Macchina a stati

Assumendo che P e V siano indipendenti tra loro e che il sistema non modifichi il suo stato in modo istantaneo, possiamo rappresentarlo con le seguenti macchine a stato Fig. 2.1.

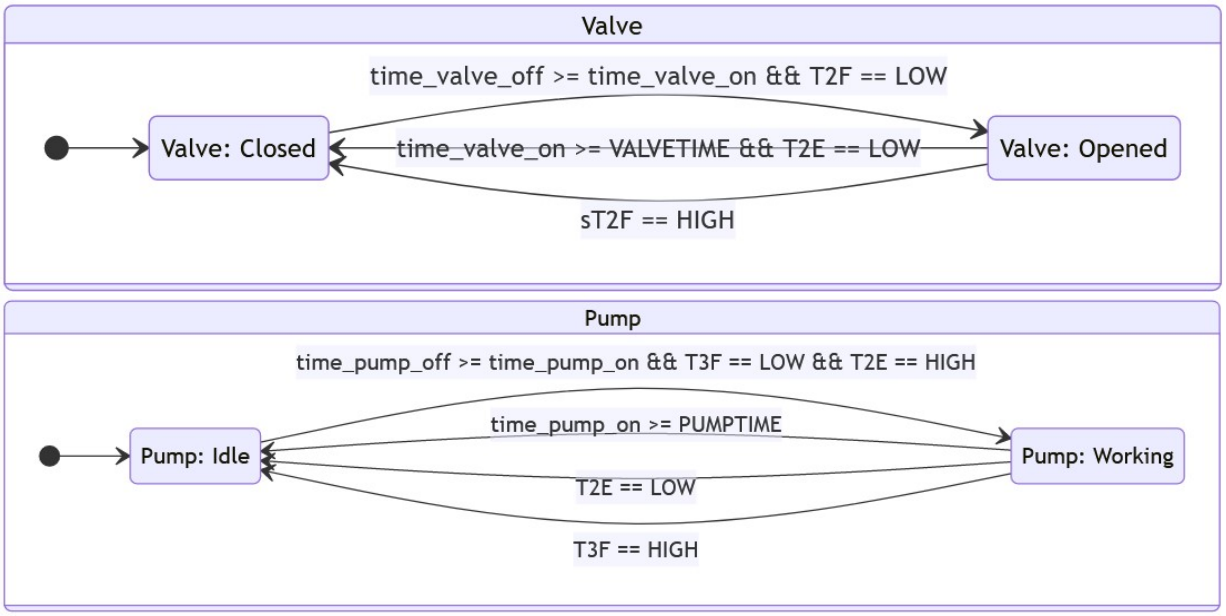

Figure 2.1: Schema del Sistema.

2.2 Tabella dei Segnali

I segnali in arrivo e in partenza verso il sistema sono riassunti nella tabella 2.1

Segnali	I/O	Pin	Dettagli
V	OUT	12	V=HIGH -> dispositivo attivo
P	OUT	5	P=HIGH -> dispositivo attivo
LED	OUT	13	LED=HIGH -> dispositivo attivo
T2E	IN	9	sensore sommerso -> T2E=HIGH
T2F	IN	10	sensore sommerso -> T2F=HIGH
T3F	IN	14	sensore sommerso -> T3F=HIGH

Table 2.1: Tabella dei segnali.

2.3 Preparazione della breadboard

3 Programmazione della scheda

Come detto nell'introduzione[1] è possibile programmare l'ESP8266 tramite l'IDE ufficiale Arduino in C++.

La struttura del codice è la seguente:

nella parte iniziale in cui vengono dichiarate le librerie necessarie e definite tutte le costanti e le variabili.

Seguono le implementazioni delle funzioni setup(), che inizializza lo stato di partenza della macchina e loop(), funzione centrale richiamata ciclicamente.

Infine vengono descritte delle altre funzioni ausiliarie che vanno a definire il comportamento del sistema.

Inoltre come vedremo nelle sezioni successive il codice è stato mantenuto il più possibile non bloccante, fatta eccezione per le funzioni relative alla connessione Wi-Fi e al broker MQTT. In caso questi servizi risultino disconnessi il dispositivo non sarà in grado di svolgere una delle sue funzioni principali ed andrà quindi a tentare la riconnessione per un numero indefinito di tentativi, bloccando tutte le altre funzioni.

Questo potrebbe presentare un problema per le condizioni di utilizzo della pompa P.

Nel codice sono state usate delle direttive al preprocessore in modo da rendere lo script modulare e poter scegliere di abilitare o disabilitare funzioni o comportamenti andando a commentare o meno solo la direttiva di definizione corrispondente.

3.1 Definizione delle variabili

In prima istanza andiamo a definire le costanti temporali necessarie e i pin ai quali associare ingressi ed uscite secondo la tabella definita nell'introduzione (Fig. 2.1).

```
// Time constants define
#define PUMPTIME 300000  // 5m pump max time of continuous operation
#define VALVETIME 30000  // 30s valve continuous time of operation with T2E = 1
#define VALVEOFF 3600000 // 1h valve cooldown in case T2E remains 0

// IN defines
#define pinT3F 14 // GPIO14 <-> D5  (safe)
#define pinT2F 10 // GPIO10 <-> SD3 (note:1 at boot)
#define pinT2E 9  // GPIO9  <-> SD2 (note:1 at boot)

// OUT defines
#define pinV 5    // GPIO12 <-> D6  (safe)
#define pinP 12   // GPIO5  <-> D1  (safe)
#define pinLED 13 // GPIO13 <-> D7  (safe)
```

Seguono le variabili di stato degli input e output e le variabili che faranno da contatore temporale durante l'esecuzione.

Viene poi definita la struttura del byte di stato del sistema, in modo da avere una rappresentazione sintetica di cosa è attivo e cosa è spento.

```
uint8_t prev_machine_state = 0;
uint8_t machine_state = 0;
// N.A.| N.A. | N.A. | sT3F | sT2F | sT2E | sP | sV
```

3.1.1 La funzione update_counter()

```
void update_counter(uint32_t &counter)
{
  counter += (millis() - last_time);
}
```

La funzione update_counter() è una funzione ausiliaria che prende come parametro un riferimento ad una variabile intera a 32 bit "counter". La funzione è utilizzata per aggiornare il valore della variabile "counter" ogni volta che viene richiamata, incrementandola del tempo trascorso dalla precedente chiamata.

Per ottenere il tempo trascorso tra le chiamate, viene sottratto il valore di last_time (variabile di stato aggiornata ad ogni chiamata) dal valore restituito dalla funzione "millis()", che restituisce il tempo in millisecondi trascorso dall'avvio del microcontrollore.

In questo modo, la funzione update_counter() permette di tenere traccia del tempo trascorso per eseguire determinate operazioni, come ad esempio l'accensione di una pompa per un tempo massimo prestabilito.

3.2 Costruzione della funzione Setup()

Nella funzione Setup vengono effettuate le impostazioni del sistema, come ad esempio le modalità con cui i pin della board vengono utilizzati e la velocità di trasmissione della porta seriale, inizializzando quest'ultima per la comunicazione.

Contestualmente vengono anche inviati via seriale dei feedback all'utente in modo da avere traccia di cosa sta succedendo.

É importante in questa fase ricordarsi di associare alle variabili che fanno da contatore temporale il valore della funzione millis().

3.3 Organizzazione del Loop()

Il metodo Loop è il seguente:

```
void loop()
{
  // put your main code here, to run repeatedly:
  // reading inputs
```

```
5   read_inputs();
6
7   // do some stuff
8   check_behaviour();
9
10  update_state();
11
12  // applying outputs to pins and computing led state
13  compute_led();
14  apply_outputs();
15
16  last_time = millis();
17
18  if (prev_machine_state != machine_state)
19  {
20    prev_machine_state = machine_state;
21  }
22 }
```

Si inizia con la lettura degli input, che siano ottenuti via seriale, via hardware tramite i pin o tramite altri metodi di trasmissione in questa istanza non è rilevante, tali dettagli sono descritti nella funzione stessa read_inputs();

La funzione check_behaviour(); va ad analizzare lo stato del sistema a partire dagli input ottenuti e dai contatori che vengono aggiornati ogni ciclo dalle funzioni ausiliarie chiamate durante l'esecuzione di quest'ultima.

Ottenuta la situazione attuale del sistema viene aggiornato il byte dello stato.

Tramite lo stato attuale del sistema si aggiorna lo stato del led, utilizzando una apposita funzione (vedi 3.3.1) è possibile gestire ogni led con un contatore ed una variabile di stato.

Vengono poi aggiornate le uscite del sistema che queste siano pin hardware, uscite seriali o altro tipo di connessione. Come detto poc'anzi per mantenere il codice il più modulare possibile sono aggiunte funzioni ausiliarie e direttive al preprocessore in modo da avere un singolo script facilmente testabile e modificabile per ogni necessità.

3.3.1 Gestione dei led

La funzione LED_Blink() è una funzione ausiliaria utilizzata per gestire il blink di un LED. In particolare, è stata progettata per essere configurabile, in modo da poter essere utilizzata in contesti diversi.

La funzione prende in ingresso il numero del pin del LED, il suo stato attuale, un contatore che viene aggiornato ad ogni chiamata alla funzione (in millisecondi), un moltiplicatore di periodo e una frazione di ciclo di lavoro. Lo stato ed il contatore sono passati per riferimento.

All'interno della funzione viene chiamata la funzione ausiliaria update_counter(), che viene utilizzata per aggiornare il contatore.

Successivamente, viene effettuato un controllo sullo stato della macchina per verificare se questa è in uno stato diverso rispetto alla chiamata precedente. In caso affermativo, il LED viene spento e

il ciclo di lavoro viene reimpostato al valore di default.

Successivamente, viene effettuato un altro controllo sullo stato del LED. Se lo stato del LED è 0 e il ciclo di lavoro rimanente è maggiore di 0, viene verificato se il tempo trascorso è maggiore o uguale al periodo di blink del LED moltiplicato per il moltiplicatore di periodo. In caso affermativo, il LED viene attivato, il ciclo di lavoro rimanente viene ridotto di 1 e il tempo trascorso viene reimpostato a 0.

In caso contrario, se lo stato del LED è 1 o il ciclo di lavoro rimanente è 0, viene effettuato un altro controllo sul tempo trascorso. In particolare, se il tempo trascorso è maggiore o uguale al periodo di blink del LED diviso per la frazione di ciclo di lavoro, il LED viene spento, il tempo trascorso viene reimpostato a 0 e, se lo stato del LED è 0, il ciclo di lavoro rimanente viene ridotto di 1.

In questo modo, la funzione LED_Blink() permette di gestire in modo preciso e flessibile il blink di un LED, adattandosi a diverse situazioni in cui può essere utilizzato.

4 Integrazione nella simulazione in Processing

Sono partito da una simulazione in processing fornitami dal docente, questa era già in grado di ricevere ed inviare dati tramite seriale.

Il sistema è rappresentato in forma grafica nella Fig. 4.1

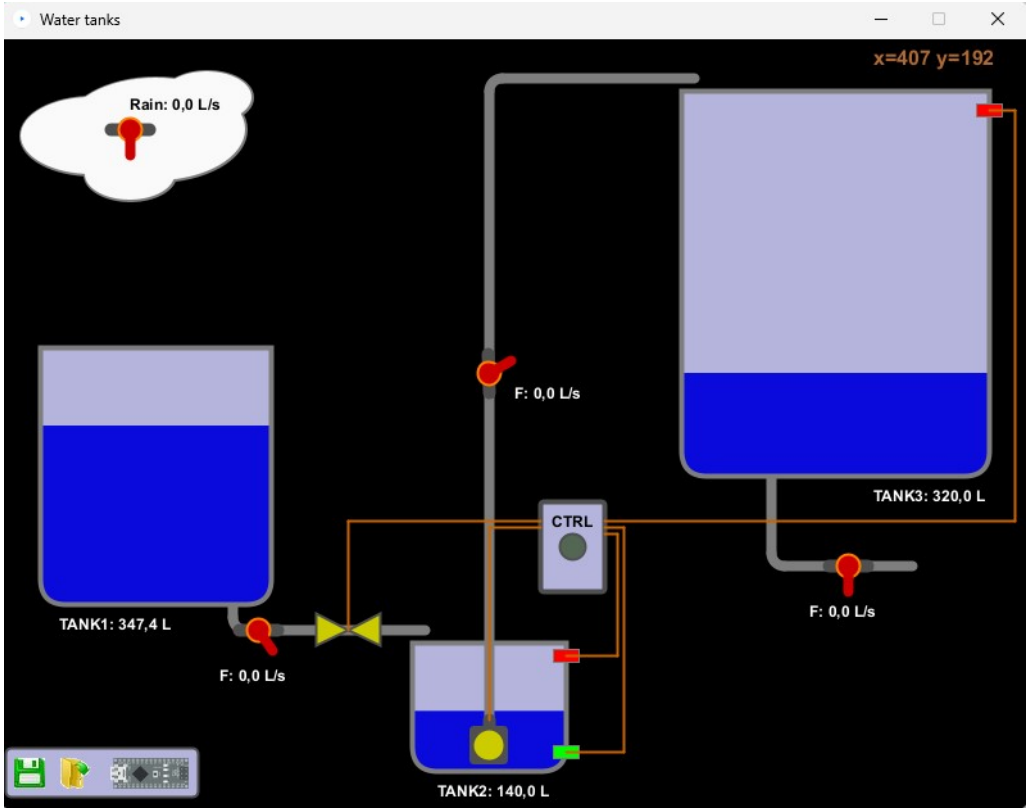

Figure 4.1: Rappresentazione del Sistema.

È previsto sia che l'utente vi interagisca tramite input diretto, quindi modificando i vari livelli utilizzando il mouse, sia che venga abilitata la comunicazione ed il controllo diretto tramite una scheda esterna collegata via seriale.

4.1 Processing?

Processing è un ambiente di sviluppo integrato (IDE) e un linguaggio di programmazione basato su Java, progettato specificamente per la creazione di immagini, animazioni e grafica interattiva. Il suo scopo principale è quello di semplificare la creazione di progetti grafici e interattivi, fornendo un'interfaccia intuitiva e una libreria grafica predefinita.

Processing è focalizzato sulla creazione di grafica e interattività, quindi fornisce una serie di funzioni per la gestione dell'input utente, la creazione di forme, la manipolazione delle immagini e l'animazione.

La libreria grafica di Processing offre una vasta gamma di funzionalità per la creazione di grafica 2D e 3D, compresi la creazione di forme, l'applicazione di effetti, l'utilizzo di texture e molto altro ancora. Processing è stato utilizzato in molti campi, tra cui la grafica generativa, la visualizzazione dei dati, l'arte digitale, il design interattivo, etc...

Inoltre, Processing supporta l'utilizzo di molte librerie esterne sviluppate dalla comunità, che forniscono funzionalità aggiuntive, come la gestione del suono, la comunicazione tramite protocolli web, la creazione di mappe e così via.

4.2 Scambio dati via connessione seriale

Tramite connessione seriale la scheda riceverá un byte che codifica lo stato attuale del sistema in processing. Tramite shifting, gli stati dei sensori vengono letti dalla scheda che elaborerá lo stato successivo tenendo conto dei contatori oltre che degli input così ricevuti.

Gli output sarannò applicati allo stesso modo, ma questa volta sarà la scheda ad inviare le informazioni sullo stato di P e V.

4.2.1 Modifiche allo script C++

Per rendere lo script realizzato nel capitolo scorso compatibile con quanto dello fin ora su processing abbiamo bisogno di aggiungere alcuni blocchi di codice che effettuino lo scambio dei dati e lo applichino adeguatamente.

È consigliabile incapsulare questi blocchi di codice in blocchi condizionali con delle direttive al preprocessore, in questo modo nel caso volessimo modificare la tipologia di comunicazione sarà molto più semplice abilitare o disabilitare la compilazione di queste, in modo da caricare sul µC solo ed esclusivamente le parti di codice rilevanti e mantenere il file compilato il più leggero possibile.

É necessario aggiungere nel metodo che esegue la lettura degli input il seguente blocco di codice che farà da ricevitore degli input da seriale e li applicherà alle variabili di stato già definite per gli input hardware.

```
#if defined(PROCESSING)

  if (Serial.available() > 0)
  {
    ds = (Serial.readStringUntil('\n').toInt());
  }

  binaryData = String(ds, BIN);

  // update inputs
  sT2E = (binaryData[0] == '1');
  sT2F = (binaryData[1] == '1');
  sT3F = (binaryData[2] == '1');

#else
```

Ricevuti gli stati in Input la procedura per il calcolo dei segnali da applicare in uscita è la medesima, ma cambia appunto l'applicazione di questi ultimi, che sarà fatta nel seguente modo:

```
#if defined(PROCESSING)
  devState = (pump_is_on << 4) | (valve_is_on << 3);
  Serial.println(devState);
  delay(500); // consigliato in fase di testing per diminuire la frequenza di aggiornamento e visualizzare
     al meglio il comportamento del sistema in relazione ai cambi di stato, in caso questi siano forzati
    dall'utente.
#else
```

La variabile devState, definita a monte dello script, va a fare da variabile d'appoggio per gli stati in ingresso e viene poi anche utilizzata come veicolo per i segnali d'uscita.

5 Accesso remoto via MQTT e Telegram

Realizzata l'implementazione di base del controllo del distema di serbatoi e testata in ambiente simulativo adesso prima di collegare la sensoristica reale è possibile realizzare la componente che permetterà al sistema di essere visualizzato e potenzialmente controllato da un client remoto.

Questo avverrà tramite l'invio di dati verso un broker MQTT. Questo potrà essere hostato in remoto su un server di terze parti, come fatto per questa mia esperienza, oppure direttamente in loco, in quanto questo è un servizio estremamente leggero ed eseguibile su un enorme numero di dispositivi. L'obiettivo iniziale era infatti quello di eseguirlo su una Raspberry Pi, che in ambito domotico potrebbe essere una soluzione estremamente applicabile anche in campo hobbystico quindi non professionale.

Una volta che le rilevazioni dei sensori e lo stato degli output saranno inviati via MQTT al broker scelto, dovrà esistere un applicativo che faccia da ricevitore in modo da rendere i dati disponibili all'utente finale.

Questa parte "front end" è stata realizzata in python sfruttando la possibilità di realizzare un bot telegram, che fornisce di base la possibilità di comunicare bidirezionalmente con un utente.

Per comodità lo stesso script python responsabile del bot è anche responsabile della comunicazione con il broker per ricevere i messaggi inviati dal sistema di controllo via MQTT.

5.1 MQTT

MQTT (Message Queuing Telemetry Transport) è un protocollo di messaggistica molto leggero ed efficiente progettato per le comunicazioni tra dispositivi IoT (Internet of Things) e applicazioni distribuite. É stato sviluppato per permettere la comunicazione tra sensori e dispositivi a basso consumo energetico, in modo da minimizzare il traffico di rete ed il consumo di batteria.

Il protocollo MQTT è basato sul modello publish/subscribe, dove i dispositivi pubblicano messaggi su un topic specifico e le applicazioni interessate si iscrivono a quel topic per ricevere i messaggi.

Il broker è il server centrale che funge da intermediario tra i dispositivi che pubblicano messaggi (publishers) e quelli che li ricevono (subscribers).

Il broker gestisce i canali di comunicazione tra i dispositivi, in modo da garantire che i messaggi siano correttamente inviati e ricevuti. In particolare, il publisher invia il messaggio al broker, che lo conserva fino a quando il subscriber richiede di riceverlo. In questo modo, il subscriber può ricevere solo i messaggi che sono rilevanti per la sua specifica richiesta.

Inoltre, MQTT offre la possibilità di utilizzare diversi livelli di qualità del servizio (Quality of Service - QoS) per garantire che i messaggi siano consegnati con la giusta affidabilità e tempestività. Ad esempio, il QoS 0 garantisce la consegna almeno una volta, mentre il QoS 2 garantisce la consegna esattamente una volta.

In questo progetto è stato utilizzato il QoS 0.

In sintesi, il broker MQTT è il componente centrale della comunicazione tra dispositivi IoT, che consente di garantire l'affidabilità e la scalabilità del sistema.

Come detto precedentemente MQTT è molto flessibile e può essere implementato su una vasta gamma di dispositivi, dal più semplice sensore al server più potente.

5.2 Modifiche allo script C++

5.2.1 Wi-Fi

Per poter implementare l'invio dei dati ad un servizio esterno raggiungibile con una connessione ad internet è stato necessario sfruttare il modulo Wi-Fi della scheda utilizzata.

La libreria $ESP8266WiFi.h$ permette di sfruttare il modulo per la connessione ad una rete Wi-Fi.

La memorizzazione dei parametri di connessione è stata accodata tramite costanti vista la natura embedded del sistema, l'oggetto WiFiClient fornito dalla libreria è l'oggetto che manterrà le informazioni relative alla connessione durante l'esecuzione del codice sulla scheda. La seguente funzione "connect_wifi()" implementa tale connessione.

```
void connect_wifi()
{
  if (WiFi.status() != WL_CONNECTED)
  {
    delay(10);
    // We start by connecting to a WiFi network
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(ssid);

    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);

    while (WiFi.status() != WL_CONNECTED)
    {
      delay(500);
      Serial.print(".");
    }

    randomSeed(micros());

    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
  }
}
```

Questa funzione viene chiamata alla fine del metodo Setup() ed ogni volta che il client MQTT non riesce a stabilire una connessione.

5.2.2 MQTT

Parlando del client MQTT la libreria utilizzata in questo caso è $PubSubClient.h$. In questo caso oltre ai parametri di connessione quali: server di destinazione, porta, username e password, sono stati definiti due topic, uno al quale effettuare subscribe ed uno sul quale pubblicare. In modo da garantire la possibilità di inviare, ma anche di ricevere dati.

L'oggetto responsabile della connessione ad un client MQTT è PubSubClient, che dovrà necessariamente essere a conoscenza dell'oggetto che rappresenta la connessione alla rete internet.

La seguente funzione si occupa di realizzare la connessione al broker MQTT, si accerta della presenza di una connessione ad internet e tenta ad intervalli regolari di connettersi al servizio dichiarato. Una volta stabilita la connessione effettua una subscribe al topic designato.

```
void reconnect_mqtt_broker()
{
  // Loop until we're reconnected
  while (!client.connected())
  {
    connect_wifi();

    Serial.print("Attempting MQTT connection...");
    // Create a random client ID
    String clientId = "ESP8266Client-";
    // clientId += String(random(0xffff), HEX);                    // COMMENT TO USE ONLINE BROKER
    // Attempt to connect
    // if (client.connect(clientId.c_str()) {                      // COMMENT TO USE ONLINE BROKER
    if (client.connect(clientId.c_str(), mqtt_user, mqtt_pwd))
    { // UNCOMMENT TO USE ONLINE BROKER
      Serial.println("connected");
      // Once connected, publish an announcement...
      // client.publish("TEST", "hello world");
      // ... or resubscribe
      client.subscribe(mqtt_channel_sub);
    }
    else
    {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}
```

Viene chiamata alla fine del metodo Setup() ed ogni chiamata della funzione Loop() seguita dal metodo client.loop(), quest'ultima è la funzione che si occupa di leggere i buffer del broker per verificare la presenza di messaggi non letti ed eventualmente chiamare la funzione di callback.

É stata quindi definita una funzione di callback da associare al comportamento dello script in caso di ricezione di un qualsiasi messaggio sul topic a cui ci si è iscritti. Questa funzione per il momento si limita a riportare sulla seriale il messaggio ricevuto ed invia sul canale di pubblicazione una conferma di ricezione.

```
void callback(char *topic, byte *payload, unsigned int length)
{
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  char *text;
  for (int i = 0; i < length; i++)
  {
    Serial.print((char)payload[i]);
    text += (char)payload[i];
  }
  Serial.println();
  client.publish(mqtt_channel_pub, text);
}
```

5.3 Realizzazione del servizio python

Stabilito il corretto funzionamento dello script caricato On-Board è possibile realizzare un servizio web, in questo caso realizzato in python, che prelevi tali informazioni e le restituisca all'utente.

Iniziamo con la stesura della base, un servizio che farà da client MQTT in ricezione sul topic verso il quale la scheda pubblica messaggi e viceversa sul topic al quale la scheda è iscritta.

```
import paho.mqtt.client as mqtt

### MQTT
mqtt_server = "node02.myqtthub.com"
mqtt_user = "receiverLautz"
mqtt_Id = "Receiver"
mqtt_pwd = "HTU7UXbZ-HtGj3qlr"

mqtt_server_port = 1883
mqtt_channel_sub = "prj_upstream"
mqtt_channel_pub = "prj_dwnstream"

# MQTT client callbacks
def on_connect(client, userdata, flags, rc):
    print("Connected to MQTT broker with result code " + str(rc))
    client.subscribe(mqtt_channel_sub)


def on_message(client, userdata, msg):
    print(
        "MQTT message received: " + msg.topic + " " + str(msg.payload.decode("utf-8"))
    )

def main():
    print("Starting...")

    mqtt_client.username_pw_set(mqtt_user, mqtt_pwd)
    mqtt_client.on_connect = on_connect
    mqtt_client.on_message = on_message
    mqtt_client.connect(mqtt_server, mqtt_server_port, 60)

    mqtt_client.loop_forever()
```

```
33
34
35 if __name__ == "__main__":
36     main()
```

Oltre alla semplice definizione dei parametri di connessione è evidente la presenza di due funzioni che faranno da callback, una ad ogni connessione effettuata con successo, che effettua la sottoscrizione al topic designato, ed una ogni volta che viene ricevuto un messaggio sul topic a cui ci si è iscritti.

Effettuato un rapido test per verificare l'avvenuta connessione occorre spostarsi momentaneamente su un client telegram per richiedere la creazione di un bot. Di questo si occupa a sua volta un bot raggiungibile al link "https://t.me/BotFather".

Per ottenere il token unico del nostro bot basterà seguire le istruzioni riportate a schermo, come in Fig. 5.1.

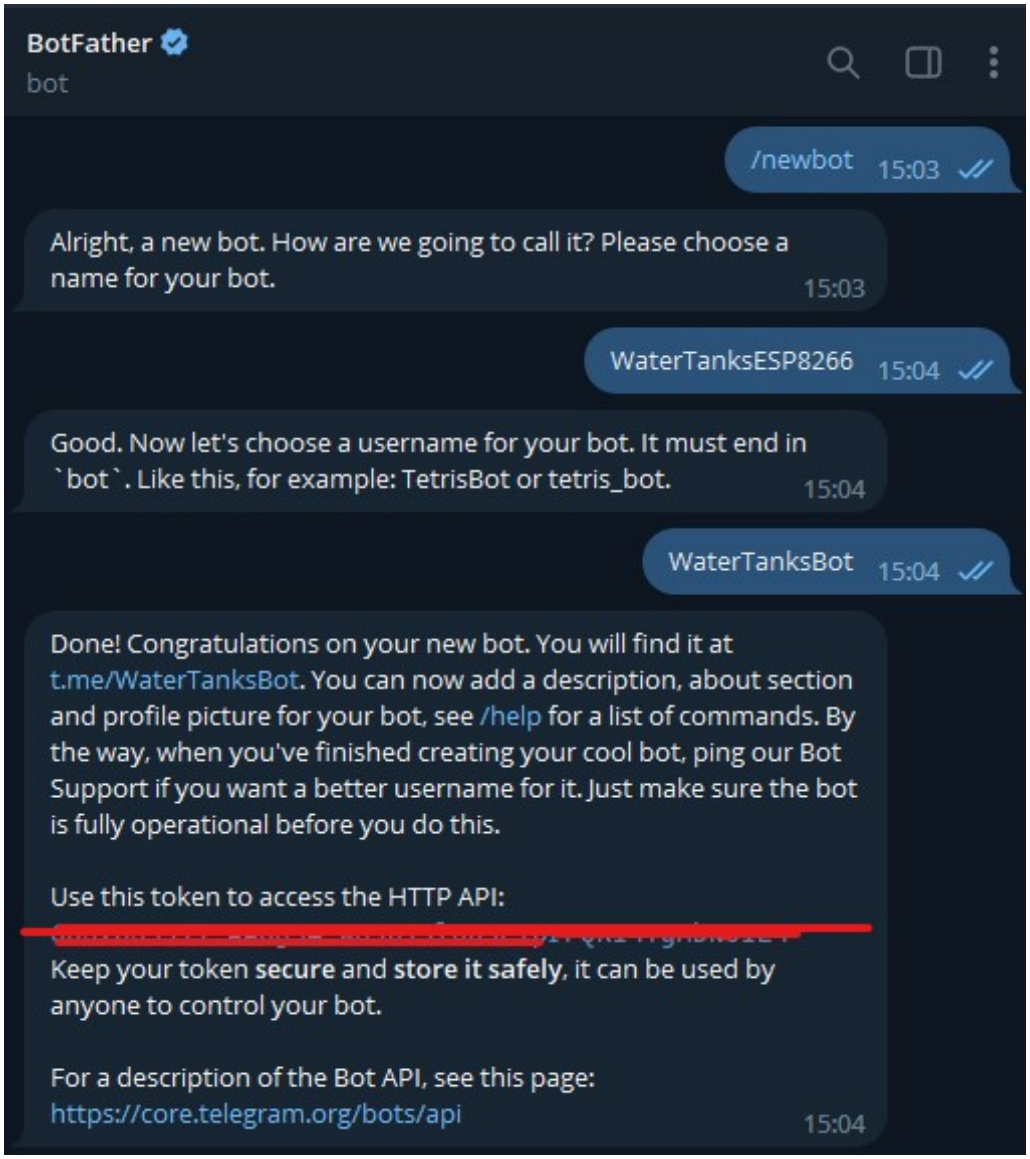

Figure 5.1: Richiesta token per bot telegram.

Una volta ottenuto il Token, sfruttando la libreria "telegram" per python è possibile integrare il codice esistente in modo che si colleghi a tale token ed agisca da backend.

É possibile quindi programmaticamente settare tutto ciò che riguarda il bot, comandi, foto profilo, descrizione, etc...

Per attivare questo servizio è necessario solamente fornire il token e chiamare le funzioni relative ai "listener" dichiarati nella libreria.

Le funzioni più importanti in questo caso sono le seguenti:

```
def update_me(update, context):
    """Send a message when the command /start is issued."""
    if idListToUpdate.count(update.message.chat.id) > 0:
        update.message.reply_text("You are already in the list of subscribers!")
    else:
        while idListToUpdate.count(update.message.chat.id) <= 0:
            idListToUpdate.append(update.message.chat.id)
            update.message.reply_text("You have been added to the list of subscribers!")

    print(idListToUpdate)


def remove_me(update, context):
    """Send a message when the command /start is issued."""
    if idListToUpdate.count(update.message.chat.id) <= 0:
        update.message.reply_text("You were not in the list of subscribers!")
    else:
        while idListToUpdate.count(update.message.chat.id) > 0:
            idListToUpdate.remove(update.message.chat.id)
            update.message.reply_text(
                "You have been removed from the list of subscribers!"
            )
    print(idListToUpdate)
```

come anche una modifica della funzione "on_message()":

```
def on_message(client, userdata, msg):
    print(
        "MQTT message received: " + msg.topic + " " + str(msg.payload.decode("utf-8"))
    )
    # Forward the message to Telegram
    for id in idListToUpdate:
        bot.send_message(
            chat_id=id,
            text="MQTT message received: "
            + msg.topic
            + " "
            + str(msg.payload.decode("utf-8")),
        )
```

I due comandi update_me e remove_me permettono all'utente di abilitare o disabilitare la ricezione dei messaggi MQTT per la sua istanza del bot, identificata dal suo chat Id che viene inserito o rimosso da un vettore locale dello script python. Queste inforazioni al momento non vengono memorizzate da nessuna parte, quindi allo spegnimento del bot tutti i client id vengono dimenticati.

La funzione on_message inoltra quindi a tutti i chat id raccolti i messagi ricevuti dal broker.

6 Conclusioni e migliorie apportabili

In conclusione, è stato realizzato con successo un sistema di controllo di serbatoi implementando tecniche di programmazione avanzate e protocolli di comunicazione moderni. Il sistema è stato testato in ambiente simulato e la sua applicabilità in campo reale è stata dimostrata.

Il sistema permette di monitorare e controllare i serbatoi in modo semplice e intuitivo tramite un bot Telegram, offrendo all'utente finale la possibilità di controllare il sistema a distanza. Inoltre, grazie all'utilizzo del protocollo MQTT, il sistema risulta altamente scalabile e facilmente integrabile con altre soluzioni IoT.

Tuttavia, il sistema è ancora migliorabile e offre spazio per ulteriori sviluppi, ad esempio, in questo caso si è scelto di realizzare un bot con singolo collegamento, senza nessun tipo di autenticazione o possibilità di modificare nè server nè topic da parte utente. Questo potrebbe essere modificabile in una eventuale futura rielaborazione.

Oppure si potrebbero staticizzare i dati relativi agli utenti telegram che hanno effettuato sottoscrizione ad un dato canale, in modo da avere persistenza in caso di spegnimenti o crash.

Si potrebbe migliorare il sistema di "visualizzazione" dello stato direttamente sulla scheda, che al momento è realizzato tramite un singolo led quindi poco significativo e poco visibile.

É inoltre possibile con pochissime aggiunte al codice on-board aggiungere anche un sistema di controllo esterno che modifica i tempi attualmente definiti come costanti, in modo da aggiungere flessibilità.

7 Appendice

7.1 Bibliografia/Linkografia

MyQTTHub