

Smart Meter IoT

Programmazione di Reti 2021

Andruccioli Manuel

Maggio 2021

Indice

1	Introduzione	2
2	Scelte Progettuali	3
2.1	Analisi del Dominio	3
2.2	Comunicazione	4
3	Design Dettagliato	5
3.1	Strutture dati e Librerie	5
3.1.1	Classe Message	5
3.1.2	Dizionario di Messaggi	5
3.1.3	Serializzazioni dei dati	6
3.2	Buffer e Tempi	6
3.2.1	Buffer	6
3.2.2	Tempi di Trasmissione	7
3.3	Network Time Protocol	7
3.4	WireShark	8
4	Guida Utente	9
4.1	Materiale Necessario	9
4.2	Guida Software	9
4.3	Guida Hardware	10
4.4	Esecuzione	11
5	Commenti Finali	12
5.1	Repository	12

Capitolo 1

Introduzione

Per il Progetto di Programmazione di Reti, ho deciso di affrontare la traccia 1, realizzando uno *Smart Meter IoT*. Le specifiche richiedevano:

- 4 *Devices* che leggono i dati e si collegano via Socket UDP al Gateway
- un *Gateway* che aspetta le 4 connessioni e, successivamente invia i dati ad un Server Centrale via Socket TCP
- un *Server Centrale* che aspetta i dati dalla Socket TCP per stamparli in output

Per rendere il progetto più interessante, in modo da approfondire i Socket ed una loro applicazione pratica, ho deciso di sviluppare sia una versione demo, utilizzabile da PC, che una versione legata direttamente al mondo IoT, basata su microcontrollore e sensore. Questo mi è stato possibile grazie alle mie esperienze pregresse, ma ha aggiunto anche nuove sfide: programmare un microcontrollore tramite *MicroPython*, un'implementazione software di Python 3, per microcontrollori.

Capitolo 2

Scelte Progettuali

2.1 Analisi del Dominio

Innanzitutto ho deciso di realizzare 3 script diversi per gestire i componenti, ognuno dei quali deve disporre di una connessione stabile verso gli altri apparati (successivamente illustreremo le varie connessioni). Questo è necessario perché, ipotizzando una situazione reale, non sarebbe consono mantenere dei dati che non possono essere inviati, nella memoria di un sensore, ma neanche di un Gateway.

Client

Il Client realizzato per i test (IoT Client), permette di essere eseguito da riga di comando passandogli un argomento, che successivamente sarà il numero identificativo dello stesso. Se la fase di setup finisce regolarmente, si entra all'interno del loop, dove, ad ogni ciclo, viene letta la misurazione ed inviata tramite una *Socket UDP*, su un indirizzo di classe *C*, appartenente al Gateway.

Gateway

Il Gateway è costituito da uno script che inizia mettendosi in ascolto su una *Socket UDP* con un indirizzo di classe *C* (da config: 192.168.1.20/24). Il Gateway aspetta che 4 *dispositivi diversi* gli inviino la loro misurazione, per poi procedere. Proprio grazie all'identificativo dato ai client può realizzare ciò. Se necessario, è possibile variare il numero di device di misura che deve attendere, mediante il parametro *number of clients* dalle impostazioni. Una volta che ha ricevuto tutte le misure, instaura una *Socket TCP* verso il Server.

Server

Il Server è realizzato da uno script che inizia ponendosi in ascolto su una *Socket TCP*. Dovrebbe utilizzare un indirizzo di classe *A* (e.g. 10.10.10.2/24), ma per motivi di test si è sfruttato il *localhost*. Una volta ricevuti i dati, li stampa a video e ritorna in ascolto.

2.2 Comunicazione

La parte più critica della comunicazione è data dalla *Socket UDP*: ipotizzando la situazione reale descritta, se i devices si collegassero una sola volta durante l'intero arco della giornata, potrebbe verificarsi un grave problema: se solo uno dei 4 pacchetti non arrivasse a destinazione, il Gateway non potrebbe procedere ad inviare i dati per un'intera giornata. Per ovviare a questo problema, ho pensato fosse corretto, far aspettare al Client un messaggio di risposta dal Gateway. Tutto questo protetto da un timeout della Socket, se si dovesse perdere il messaggio di ritorno. Nel caso in cui scatti il timeout della Socket, il client provvederà ad inviare nuovamente i dati. Se il Gateway avesse già ricevuto i dati dal Client, non andrà ad aggiornarli, ma invierà solamente il messaggio di check. Per quanto riguarda la connessione verso il Server, il tutto rimane più semplice, instaurando solo la Socket TCP.

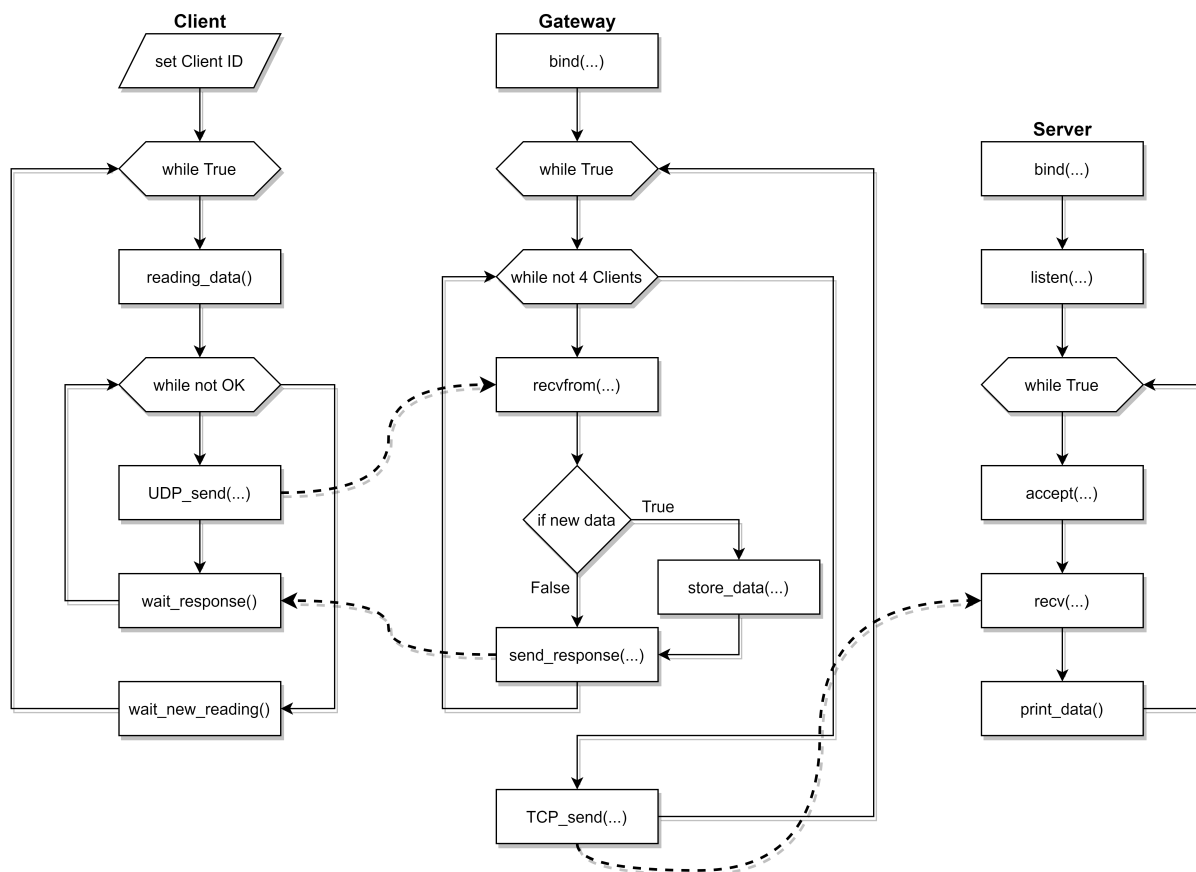


Figura 2.1: Schema delle componenti fondamentali della rete

Capitolo 3

Design Dettagliato

3.1 Strutture dati e Librerie

3.1.1 Classe Message

Per incapsulare il concetto di messaggio che viene scambiato all'interno della rete, ho deciso di implementare una classe apposita. All'interno del file *data message* possiamo trovare la classe *message*, utile al nostro scopo. Quando il Client misura i valori, creerà un'istanza della classe contenente i vari campi da trasmettere, arricchito con altri dati utili, che vedremo più avanti. Nel caso volessimo aggiungere responsabilità, cioè trasferire ulteriori dati, oppure modificarne la rappresentazione, basterà agire su questa classe e settare appositamente i nuovi campi. Tutto questo permette di non alterare il codice già scritto per l'invio tramite Socket. Ovviamente dovremo adottare appositi metodi di serializzazione, dato che su classe non potremmo richiamare il metodo *encode()*.

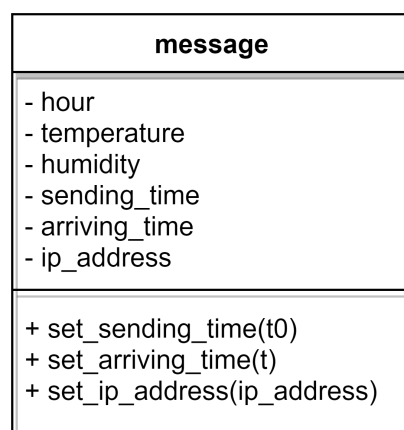


Figura 3.1: Classe message

3.1.2 Dizionario di Messaggi

All'interno del Gateway, per tenere traccia dei messaggi associati spediti dai vari Client, utilizzo un *Dict*, il quale associa il numero univoco del Client ai suoi dati. Questo mi permette di vedere se un Client ha già inviato i messaggi e, alla necessità, salvare i dati, oppure scartarli. Questo dizionario sarà successivamente inviato al Server Centrale, che potrà stampare i dati, associati ai relativi Client.

3.1.3 Serializzazioni dei dati

Come abbiamo visto più volte a lezione, i dati trasmessi mediante una Socket, sono delle Stringhe, alle quali viene applicato il metodo *encode()* e successivamente *decode()*. Dal momento che ho la necessità di trasferire la classe *message*, quest'ultima non dispone dei due metodi. Per ovviare a questo problema e poter trasmettere i dati via Socket, ho adottato due pacchetti disponibili per Python: *Pickle* e *JSON*.

Pickle

Questo modulo, mediante la funzione *dumps(...)*, mi permette di passargli un qualsiasi oggetto per ottenerne un altro, serializzato in bytes, pronto da spedire su una Socket. Nel momento della ricezione dei bytes dalla Socket, mi basterà passarli alla funzione *loads(...)*, che deserializzerà e restituirà l'oggetto, pronto da essere usato.

Grazie a questo modulo, posso trasferire facilmente gli oggetti mediante una Socket.

JSON

Durante la scrittura del codice demo, mi sono servito interamente di *Pickle*, proprio per la sua facilità. Nel momento in cui ho caricato il codice sul microcontrollore, ho riscontrato un problema: *MicroPython*, essendo un subset di *Python*, non dispone di un'implementazione di *Pickle*. Invece di modificare il sorgente *MicroPython*, ho deciso di sfruttare una libreria disponibile, per mandare i messaggi dai Client al Gateway: *JSON*.

Questo modulo, che viene sfruttato esattamente come *Pickle*, differisce solo per ciò che prende come input e output. Basta solamente passargli il *dict* della classe da serializzare.

3.2 Buffer e Tempi

3.2.1 Buffer

Ipotizzando la situazione reale, i Client inviano i dati una sola volta durante la giornata. Questo lascia supporre che non ci sarà congestione della rete e, nel caso peggiore in cui tutti i Client inviassero nello stesso momento, dovremmo gestire un buffer grande tanto quanto il totale dei bytes dei Client.

Detto ciò, ho assunto che ogni Client invii al massimo 1024 bytes. Così facendo imposto il buffer del Gateway della Socket UDP a $1024\text{bytes} \times 4\text{Clients} = 4096\text{bytes}$

Per la Socket TCP verso il server, ho impostato la grandezza del buffer sempre a 4096 bytes, perché dovranno essere inviati tutti i dati contemporaneamente, con la piccola aggiunta della struttura dati che li contiene, la quale non influirà più di tanto.

Nel caso in cui si vorranno aumentare i dati letti e trasferiti dai devices, andremo ad ampliare il buffer, sempre rispettando la proporzione, in base al numero dei Client.

3.2.2 Tempi di Trasmissione

Per il calcolo dei tempi di trasmissione, ho adottato due metodologie diverse, anche per la presenza del Client con microcontrollore. Sfortunatamente, essendo in *localhost*, alcune misure non sono apprezzabili, dato che i tempi di trasmissione sono troppo rapidi

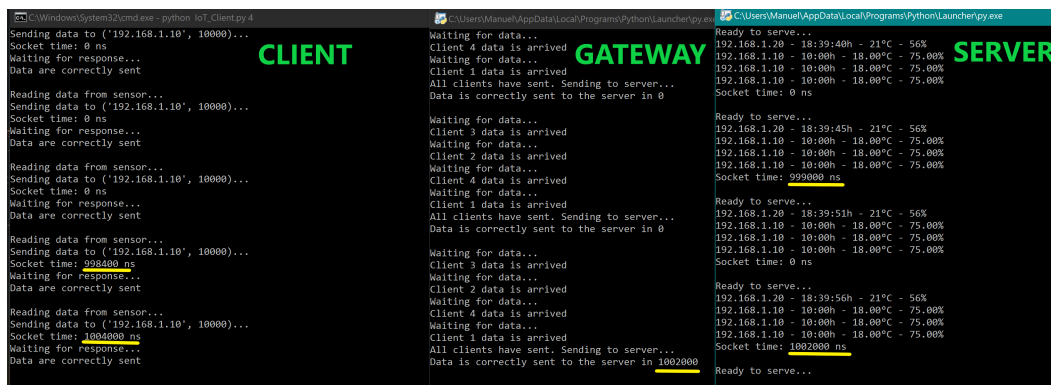
Client demo e Gateway-Server

Questo metodo di misura è stato possibile proprio perché client, Gateway e Server risiedono sulla stessa macchina e condividono lo stesso formato e misura di tempo.

Nel momento in cui viene trasferito il messaggio dal Client, quest'ultimo aggiunge il valore del tempo di invio (t_0) nella classe message. Quando il Gateway riceve i dati, va ad aggiungere al message il tempo di arrivo (t). Quando si vorrà conoscere il tempo che il pacchetto ha impiegato per il trasferimento nella Socket, basterà ricavarlo con $\Delta t = t - t_0$

Client microcontrollore

Nel caso in cui usassi il metodo visto precedentemente, risulterebbe errato con il microcontrollore, perché ha una rappresentazione diversa del tempo (ha meno cifre disponibili per rappresentarlo, quindi verrebbe errato Δt). Per questa ragione ho deciso di misurare il tempo con tramite una differenza tra inizio e terminazione dell'invio del messaggio nella Socket. La precisione di questa misura è sicuramente influenzata del numero ridotto di cifre disponibili al microcontrollore



```
CLIENT
Sending data to ('192.168.1.10', 10000)...
Socket time: 0 ns
Waiting for response...
Data are correctly sent
Reading data from sensor...
Sending data to ('192.168.1.10', 10000)...
Socket time: 0 ns
Waiting for response...
Data are correctly sent
Reading data from sensor...
Sending data to ('192.168.1.10', 10000)...
Socket time: 0 ns
Waiting for response...
Data are correctly sent
Reading data from sensor...
Sending data to ('192.168.1.10', 10000)...
Socket time: 1004000 ns
Waiting for response...
Data are correctly sent

GATEWAY
Waiting for data...
Client 4 data is arrived
Waiting for data...
Client 1 data is arrived
All clients have sent. Sending to server...
Data is correctly sent to the server in 0
Waiting for data...
Client 3 data is arrived
Waiting for data...
Client 2 data is arrived
Waiting for data...
Client 4 data is arrived
Waiting for data...
Client 1 data is arrived
All clients have sent. Sending to server...
Data is correctly sent to the server in 1002000

SERVER
Ready to serve...
192.168.1.20 - 18:39:40h - 21°C - 56%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
Socket time: 0 ns
Ready to serve...
192.168.1.20 - 18:39:45h - 21°C - 56%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
Socket time: 999000 ns
Ready to serve...
192.168.1.20 - 18:39:51h - 21°C - 56%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
Socket time: 0 ns
Ready to serve...
192.168.1.20 - 18:39:56h - 21°C - 56%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
192.168.1.10 - 10:00h - 18.00°C - 75.00%
Socket time: 1002000 ns
Ready to serve...
```

3.3 Network Time Protocol

Nel momento cui ho richiesto la data corrente al microcontrollore per salvarla nel messaggio da inviare, è sorto un problema: il device, non essendo dotato di una batteria tampone, quando viene a mancare l'energia elettrica ricomincia a contare il tempo dal 1° Gennaio 2000. Questo rende impossibile ottenere una misura consona.

Per ovviare a questo problema ho utilizzato il *Network Time Protocol* (NTP), così da ripristinare la data e l'ora corretta ad ogni interruzione di alimentazione.

3.4 Wireshark

Durante la realizzazione del sistema descritto, ho sfruttato Wireshark per testare il funzionamento. Come si nota in Figura 3.2, possiamo vedere i pacchetti scambiati tra il microcontrollore e il Gateway, mediante la Socket UDP.

Possiamo notare il Gateway con IP *192.168.1.10* e il Client con IP *192.168.1.20*

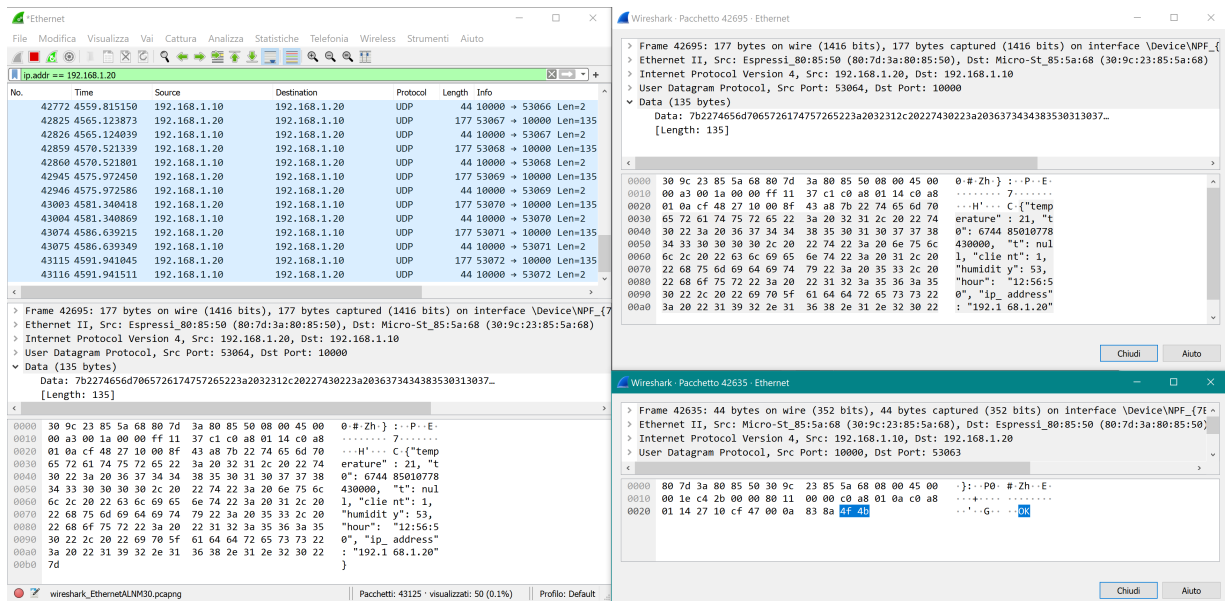


Figura 3.2: Scambio di pacchetti da microcontrollore a Gateway

Capitolo 4

Guida Utente

Se si volesse utilizzare il sistema che ho realizzato, si può operare in modalità demo, nel caso in cui non si dispone di tutti gli strumenti necessari.

4.1 Materiale Necessario

Avremo bisogno di:

- **ESP32** (o ESP8266) da utilizzare come Client
- **DHT11** sensore di temperatura e umidità
- Jumper per i collegamenti
- Cavo USB micro B per connettere ESP32 al PC

4.2 Guida Software

Bisogna installare 2 moduli Python per configurare il Client:

- **esptool.py**
- **ampy.py**

Il primo modulo permette di flashare la memoria di ESP32 con MicroPython. Collegarsi al sito di MicroPython (<https://micropython.org/download/esp32/>) e seguire la procedura.

Ora con il modulo *ampy* sarà possibile eseguire script Python su ESP32 (seguire la documentazione).

4.3 Guida Hardware

Effettuare i collegamenti come riportato in figura 4.1:

- un collegamento su +5V
- un collegamento su GND
- un collegamento su PIN 4 di ESP32

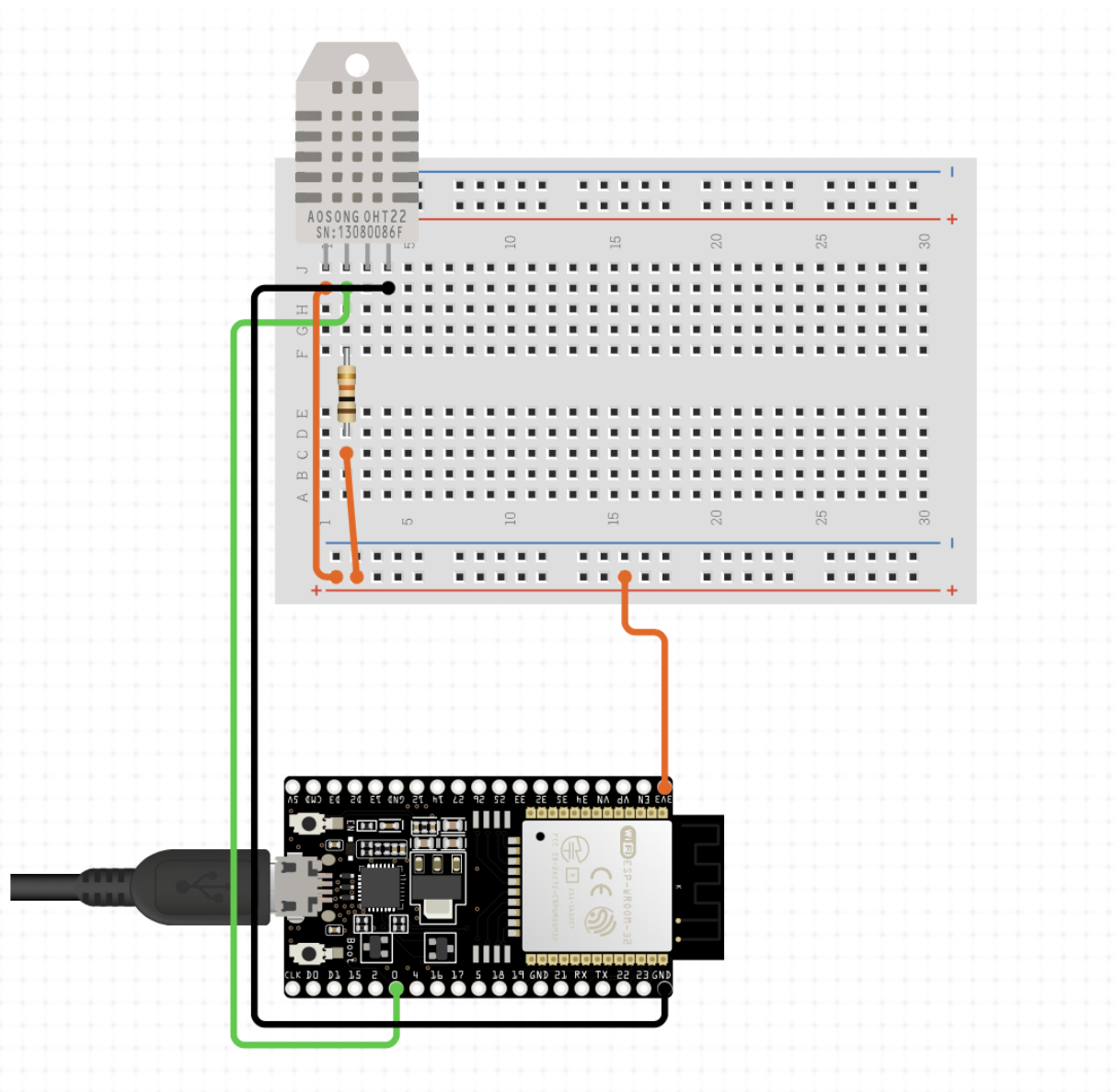


Figura 4.1: Cablaggio di microcontrollore e sensore

4.4 Esecuzione

Per far funzionare il sistema, collegarsi alla *repository* e scaricare il codice. Successivamente basterà eseguire:

1. modificare, se necessario, il file **config.py**
2. eseguire **Server.py** e **Gateway.py** (non importa l'ordine)
3. eseguire **IoT Client.py** da riga di comando, passandogli il numero identificativo

Se si volesse utilizzare il client con microcontrollore, bisognerà andare nella directory *sensor version* ed eseguire gli script sul microcontrollore (ho lasciato un setup.bat, utilizzabile da Windows per installare direttamente tutto il necessario. Bisogna modificarlo selezionando la porta USB dove è collegato ESP32).

Capitolo 5

Commenti Finali

Sono soddisfatto di questo progetto, anche perché ho potuto toccare con mano i problemi che possono incorrere utilizzando tecnologie diverse.

Grazie per la lettura!

5.1 Repository

Per collaborazioni e/o migliorie lascio il repository GitHub:

<https://www.github.com/manuandru/Reti2021-Smart-Meter-IoT>

Manuel Andruccioli, Maggio 2021