Politecnico di Torino

Relazione del laboratorio software del corso “Tecnologie per IoT”

Pietro Macori s246163

Alessandro Versace s246056

Ferdinando Micco s245340

ANNO ACCADEMICO 2019-2020

Indice

I laboratori relativi alla parte software del corso sono stati strutturati in modo da permetterci di sviluppare un sistema IoT distribuito, composto da un Arduino Yun nel ruolo di Device, tre diversi client che offrono servizi all'utente, un message Broker MQTT e un Catalog, una componente software in grado di interconnettere i vari elementi precedentemente citati. In questa relazione illustreremo come sono stati sviluppati nel corso dei laboratori questi componenti, descrivendo le nostre scelte implementative, i problemi da noi riscontrati e le nostre riflessioni sull'utilizzo di una piattaforma distribuita rispetto ad un sistema a sensori in grado di effettuare le stesse operazioni in modo autonomo.

Catalog

Il Catalog è stato il primo elemento che abbiamo implementato a partire dal secondo laboratorio software. Il suo ruolo è quello di gestire diversi componenti e permettere ad essi di interfacciarsi in modo semplice ed intuitivo. Il Catalog è un webserver di tipo RESTful implementato tramite il framework Cherrypy che deve essere in grado di gestire la registrazione di device (Arduino nel nostro caso), di un message Broker MQTT, di servizi e di utenti. Il Catalog deve essere in grado di fornire informazioni riguardo ad ognuno degli elementi che si sono registrati, accettare nuove registrazioni e rimuovere tutte le componenti che dopo due minuti non hanno effettuato successive registrazioni.

La nostra scelta implementativa è stata quella di adottare diverse classi da esporre come servizi, ognuna su un differente path identificativo del nostro web Server. L'implementazione delle varie classi è molto simile tra di loro. Il funzionamento si basa sulla lettura e scrittura di un file in formato Json, contenente tutte le informazioni dei vari componenti registrati sul Catalog. La lettura e scrittura del file Json è stata protetta utilizzando due lock, in quanto azioni effettuate in parallelo avrebbero potuto portare ad un accesso non corretto alla sezione critica. Il formato utilizzato per la scrittura del file è il seguente

```
{
 "broker":{
        "name":"brokerName",
        "port":"brokerPort"
        },
 "devices":[],
 "services":[],
 "users":[]
}
```

Il formato utilizzato per la descrizione dei devices e dei servizi all'interno delle liste è riportato sulla sinistra con un esempio che mostra un device con due risorse e i rispettivi endpoints (ep), mentre il formato utilizzato per i nuovi utenti viene mostrato sulla destra.

```
{
 "id": 1,
 "ep": {
       "temp": "/iot/13/temp",
       "led": "/iot/13/led"
  },
  "t" = 156345
}
```

```
{
 "id":"UserID" ,
 "name":"UserName",
 "last":"UserSurname",
 "mail":"UserEmail",
}
```

È importante notare che il timestamp “t” non deve essere inserito dall’utente, ma viene gestito dal server stesso in modo che non vi siano errori di sincronizzazione causati da timer diversi. Un’altra scelta implementativa che abbiamo effettuato è stata quella di accettare gli endpoints in un formato dizionario. In questo modo il Catalog è in grado di creare un campo “resources” utilizzando le chiavi del dizionario “ep”. Questa scelta è stata effettuata poichè l’invio da parte di Arduino di stringhe molto lunghe tramite il metodo POST spesso creava problemi e malfunzionamenti. Abbiamo quindi optato per delegare più lavoro al server in modo da snellire le operazioni del device che ha una potenza computazionale ridotta.

Per non soffermarci sui dettagli implementativi relativi ai singoli elementi del nostro codice, che risultano spesso molto ripetitivi e abbastanza comprensibili, preferiamo esporre le funzionalità pratiche del nostro web server e analizzare in dettaglio solo le funzioni “core”. Di seguito verranno elencati i servizi offerti dal Catalog con il relativo metodo HTTP da utilizzare e il percorso su cui effettuare le richieste.

Azione	**Metodo**	**Location**
Ottenere nome broker MQTT	GET	http://ip_catalog:8080/broker/name
Ottenere porta broker MQTT	GET	http://ip_catalog:8080/broker/port
Registrare un device	POST	http://ip_catalog:8080/devices
Ottenere lista devices	GET	http://ip_catalog:8080/devices
Ottenere device dato ID	GET	http://ip_catalog:8080/devices?id=deviceID
Registrare un utente	POST	http://ip_catalog:8080/users
Ottenere lista utenti	GET	http://ip_catalog:8080/users
Ottenere utenti dato ID	GET	http://ip_catalog:8080/users?id=userID
Registrare un servizio	POST	http://ip_catalog:8080/services
Ottenere lista servizi	GET	http://ip_catalog:8080/services
Ottenere servizio dato ID	GET	http://ip_catalog:8080/services?id=serviceID

Le operazioni di GET sono molto semplici. Come prima operazione viene letto il file "information.json" tramite il metodo *readJson(),* utilizzando una lock in modo da non leggere il file durante una operazione di scrittura da parte di altri thread; viene quindi ritornato il valore desiderato a seconda della classe che chiama la funzione. L'unica complicanza è il ritorno di elementi dato l'identificativo. Questa parte viene gestita dal metodo *retrive_byId()* che in base al numero di parametri (1 parametro = richiesta in base all'ID, 0 parametri = richiesta di tutti gli elementi) ritorna il singolo elemento o la lista completa.

Le operazioni di POST sono leggermente più complesse. I valori da inserire sono passati tramite il body del messaggio HTTP, il valore viene convertito in dizionario, viene eseguita un'acquisizione su una seconda lock (in modo da poter leggere e scrivere in un unico ciclo, evitando che altri thread si intromettano tra la fase di lettura e quella di scrittura) e viene finalmente aggiunto il nuovo testo al file originale. In questa fase di scrittura i timestamp vengono salvati grazie alla funzione *time.time()*, e nel caso dei device viene aggiunto il campo "resources". La scrittura del file contenente le nuove informazioni viene effettuata dal metodo *writeJson().*

Soffermandoci sul modello implementativo con le due lock, abbiamo notato che il nostro caso rispecchiava il problema noto come "multiple readers - multiple writers". Abbiamo dunque deciso di gestire il problema in maniera semplificata permettendo l'accesso alla sezione critica non simultanea durante le operazioni, dato che nel caso di un numero eccessivamente grande di richieste in un lasso di tempo ridotto, le classi potrebbero ritrovarsi in busy waiting. Una situazione che però non definiremmo critica ma addirittura trascurabile, per via dell'utilizzo che verrà fatta di questa infrastruttura. Una delle due lock è utilizzata in caso di semplice apertura del file Json, evitando che contemporaneamente due funzioni vadano ad accedere alla risorsa, mentre una seconda lock accorpa in sezione critica il caso di lettura con l'aggiornamento del file, evitando che un reader possa leggere il Catalog non ancora aggiornato o peggio, che in caso di scritture simultanee qualche dato venga perso o l'output sia disordinato per via dell'ordine delle operazioni.

La parte che gestisce la rimozione di device e servizi obsoleti viene gestito da un thread indipendente. Questo thread è composto da un costrutto *while* infinito che ogni 5 secondi legge il file json, compara i timestamp dei vari elementi registrati con il tempo attuale utilizzando la funzione *time.time()*. Se la differenza tra questi due tempi è maggiore di 120 secondi allora le informazioni del servizio o del device verranno rimosse dal file.

All'interno del main abbiamo semplicemente lanciato il thread, configurato i valori del web server e montato le varie classi nei rispettivi path. Abbiamo infine esposto il Catalog sull'indirizzo IP della macchina che lo avrebbe fatto girare in modo da poter essere visibile e raggiungibile dalla rete locale, e abbiamo impostato la porta 8080 come endpoints del web server.

Device

All'interno della nostra piattaforma distribuita, il device è stato realizzato con un Arduino Yun, una scheda elettronica non solo in grado di interfacciarsi a vari sensori e attuatori tramite un classico MCU, ma anche in grado di svolgere operazioni più complesse grazie ad un secondo processore in grado di gestire una distribuzione Linux. A differenza del secondo laboratorio della parte hardware del corso, la scheda esegue operazioni molto semplici, in quanto essendo parte di una piattaforma distribuita, l'analisi dei dati raccolti dai sensori, il processing, i conseguenti comandi di attuazione e l'interfaccia con un utente vengono eseguite da servizi esterni appositi. Il device deve quindi solo effettuare una registrazione al Catalog, leggere i sensori di temperatura, movimento, rumore e inoltrarli tramite il protocollo MQTT ai servizi registrati come subscribers al message Broker. Oltre a ciò, il device stesso si deve sottoscrivere a determinati topic per poter ricevere i comandi di attuazione dai servizi. E' fondamentale notare che i vari topic utilizzati sono scelti dal device stesso, il quale manderà al Catalog tutte le informazioni sugli endpoints e saranno i servizi a dover recuperare i topic scelti dal device tramite il Catalog. Durante la fase di *setup()* viene chiamata la funzione *getBroker()* che tramite il componente Process della libreria Bridge (la quale permette di eseguire comandi della shell di Linux) permette di ottenere il nome e la porta del message Broker salvato nel Catalog. In questa funzione il comando eseguito sarà il comando *curl* che permette di trasferire dati utilizzando diversi protocolli tra cui HTTP. Nella nostra funzione il comando viene eseguito due volte, la prima per ottenere il nome del broker, la seconda per ottenere la porta utilizzata. I valori ottenuti vengono salvati in due variabili globali. Sempre nel *setup()* viene eseguita la subscribe a tutti i comandi di attuazione tramite il topic */iot/13/a/#* e viene associata la funzione *getMQQTdata()* alla ricezione di tali messaggi. La funzione *getMQQTdata()* gestisce i comandi ricevuti in base al topic e imposta gli attuatori in base ai valori ricevuti. Di seguito mostriamo una tabella riassuntiva del topic, del formato e della rispettiva azione della scheda.

Topic	Formato messaggio	Valore	Azione eseguita
/iot/13/a/d	"testo da visualizzare"	Stringa	Visualizza sul display lcd il messaggio ricevuto
/iot/13/a/f	"velocità della ventola"	Intero (0-255)	Imposta sul PIN della ventola la velocità ricevuta
/iot/13/a/l	"luminosità del led"	Intero (0-255)	Imposta sul PIN del led il valore ricevuto
/iot/13/a/rgb	"red,green,blue"	3 Interi (0,255)	Imposta l'intensità luminosa dei 3 led rgb

Il formato scelto vuole essere il più snello possibile in modo da evitare complesse operazioni di decodifica le quali hanno provocato innumerevoli problemi durante lo sviluppo della piattaforma. Infatti, l'utilizzo di JsonDocument per la codifica/decodifica di messaggi Json si è rivelata essere estremamente problematica e non stabile per i nostri scopi. Abbiamo quindi definito un formato ad hoc per essere più efficaci nella ricezione dei messaggi MQTT.

All'interno del *loop()* viene effettuata la registrazione al Catalog ogni 30 secondi, tramite la funzione *postRequest()* che sfrutta nuovamente la libreria Bridge per effettuare una *curl* al Catalog, inviando tramite HTTP POST il nome del device e gli endpoints disponibili. Infine, vengono effettuate tre pubblicazioni MQTT per inviare i valori di temperatura, movimento e rumore ai rispettivi topic. Il formato utilizzato per i messaggi MQTT rispetta il dataformat SenML.

Un sommario dei topic e della loro funzionalità viene riportata di seguito.

Topic	Funzionalità
/iot/13/tmp	Invio del valore letto dal sensore di temperatura in formato SenML
/iot/13/mov	Invio del valore letto dal sensore di movimento in formato SenML
/iot/13/nos	Invio del valore letto dal sensore di rumore in formato SenML

Servizio 1: Smart Home Controller

Il primo servizio che abbiamo implementato è un client in grado di effettuare le medesime operazioni svolte da Arduino nel secondo laboratorio della parte hardware. Lo Smart Home Controller dovrà quindi gestire degli attuatori (una ventola, un led e un monitor lcd) in modo da simulare un impianto di climatizzazione intelligente, in grado di prendere decisioni a seconda della temperatura rilevata e dell'eventuale presenza di persone nell'ambiente circostante.

Il nostro servizio deve registrarsi al Catalog, ottenere gli endpoints del Device e successivamente ricevere i valori letti dai sensori di Arduino tramite MQTT, processare i dati ottenuti e inviare degli opportuni comandi di attuazione alla scheda Arduino. Dato che la maggior parte delle funzioni utilizzate nel client devono svolgere le medesime operazioni delle funzioni scritte in linguaggio Arduino nei precedenti laboratori, il team ha deciso di evitare di riscrivere le funzioni da zero, preferendo riadattarle in linguaggio Python. Questa scelta può sembrare poco elegante, poiché alcune funzioni avrebbero potuto essere implementate in maniera più consona al cambio di linguaggio, ma questa scelta ci ha garantito un codice funzionante, leggero (il client è composto da meno di 300 righe di codice) e

facilmente comparabile con quello scritto nei laboratori hardware. Data questa nostra scelta preferiamo non focalizzare troppo l'attenzione sulle funzioni che elaborano i dati (un'analisi più approfondita di esse può essere reperita nella relazione dei laboratori hardware) per poter essere più esaustivi sulle funzionalità aggiuntive che il client offre.

La registrazione al Catalog viene gestita in modo ripetitivo all'interno del main. Utilizzando la funzione *retrieve_broker()* viene effettuata una richiesta HTTP di tipo GET al Catalog che ritorna il nome e la porta del Broker MQTT. Successivamente, tramite la funzione *retrieve_sensors()* viene eseguita una richiesta al Catalog in modo da ottenere la lista dei device connessi. Un menù molto basilare permette all'utente di selezionare il device desiderato. Nel caso nessun device sia connesso, il programma verrà chiuso in quanto nessuna operazione sarà possibile. Un secondo loop, controllato dalla variabile *checkPresenceDevice,* che termina il while nel caso il device scelto non sia più connesso al Catalog, esegue ciclicamente ogni 5 secondi una POST al web server per registrarsi come servizio attivo e successivamente chiede all'utente di inserire dei nuovi range di temperatura (set-points). In questo modo la funzionalità che su Arduino veniva svolta dalla connessione seriale è stata implementata in remoto. All'interno del *main* viene infine creato un nuovo oggetto della classe *Client* che gestirà le connessioni MQTT e l'elaborazione dei dati.

Nella classe *Client* il metodo che permette la comunicazione tramite protocollo MQTT con il Device è il metodo *start().* Questo sottoscrive il client ai Topic relativi alla temperatura, al movimento e al rumore e successivamente lancia un thread denominato *Routine* che simula il funzionamento della funzione *loop()* di Arduino. In questo modo abbiamo nuovamente una simmetria con il codice scritto in precedenza. Inoltre, all'interno di questo thread viene gestito il controllo del Device: ogni 5 secondi viene verificato che il Device scelto in precedenza sia ancora connesso al Catalog.

Il metodo "core" della classe Client è il metodo *notify().* Questa callback viene richiamata ad ogni ricezione di messaggi MQTT. Dopo aver ricevuto i dati dall'Arduino e aver elaborato i rispettivi comandi di attuazione (tramite le funzioni *actualTemp(), regulate_fan_or_led()* e *checkPresenceSOUND()* che eseguono le identiche operazioni svolte dalle funzioni Arduino), vengono eseguiti i comandi di pubblicazione per la ventola, il led e il display lcd. Sottolineiamo il fatto che le pubblicazioni vengono effettuate all'interno del metodo *notify()* quindi l'invio di messaggi al Device avverrà solamente se il Device stesso invierà dati al nostro servizio. In questo modo evitiamo di inviare dati ad un dispositivo che non sta funzionando correttamente in quanto l'unica funzionalità della scheda Arduino è quella di inviare informazioni ai servizi. Un altro dettaglio molto importante è la frequenza con cui le pubblicazioni vengono effettuate. Infatti, la gestione di un impianto di condizionamento non richiede una risposta ai cambiamenti in tempo reale. Possiamo quindi evitare di "inondare" il Device con numerosi messaggi al secondo, riducendo i comandi inviati ad uno ogni 5-10 secondi. In questo modo la scheda Arduino risponderà leggermente in ritardo ai cambiamenti ma dovrà gestire un carico di lavoro nettamente più consono alle sue capacità computazioni.

Servizio 2: Grafico e statistiche sulla temperatura

Il secondo servizio che abbiamo sviluppato è un Web Server che viene eseguito in locale, in grado di mostrare a schermo il grafico e le varie statistiche relative alla temperatura. Per poter implementare queste funzionalità, ci siamo affidati alla libreria *plotly,* che in modo automatico è in grado di lanciare un server in locale e mostrare un grafico che si aggiorna in tempo reale.

Il codice che gestisce la connessione del servizio al Catalog è del tutto analogo a quello utilizzato nel Remote Smart Controller. Tramite la funzione *retrieve_broker()* viene effettuata una richiesta HTPP di tipo GET al Catalog in modo da ottenere le informazioni del message Broker, mentre la funzione *retrieve_sensors()* permette all'utente di selezionare il Device desiderato. Successivamente viene creata un'istanza della classe *Client()* che nuovamente ci permetterà di gestire la comunicazione MQTT. In questo servizio nessuna informazione viene mandata all'Arduino e solo le informazioni riguardanti la temperatura verranno ricevute. All'interno del metodo *notify()*, ogni 2 minuti viene salvata la temperatura ricevuta in una lista; in questo modo la lista di valori utilizzati per il grafico non comprenderà tutte le innumerevoli temperature ricevute tramite la subscribe.

L'ultima operazione effettuata nel *main()* prima di lanciare il web server è creare thread della classe Service, che semplicemente ogni minuto rinnova la propria registrazione presso il Catalog.

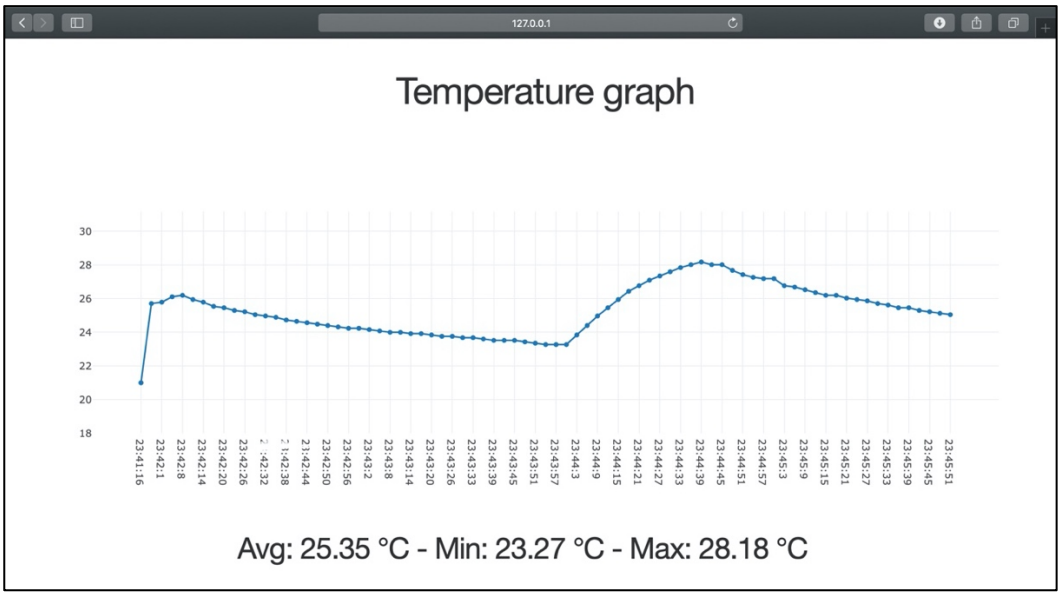
Il plot della temperatura viene effettuato e aggiornato tramite una *app* creata con *Dash* in formato html, a cui abbiamo aggiunto come da documentazione un css per un leggero abbellimento estetico. La parte principale è gestita tramite un *dash-core-component* di tipo *Graph*, il quale è in grado di gestire un grafico realizzato con *plotly* e aggiornato tramite una funzione di *callback update_graph()*, che fa l'upgrade del componente con id "graph" con un intervallo prefissato all'interno della stessa *app* in un secondo componente *Interval*.

Tramite questa funzione, gestiamo il grafico calcolando il momento attuale di "campionamento" della temperatura in formato temporale tramite la funzione ad hoc *actualTime()*, che verrà aggiunto alla coda rappresentante l'asse delle ascisse X, mentre l'ultima temperatura ricevuta è stata inserita in precedenza durante la ricezione di un messaggio MQTT nella coda Y. Infine, ci occupiamo di aggiornare i dati del grafico rifornendo le due serie aggiornate e ritornando il plot con il corretto layout dei due assi.

Lo stesso procedimento è eseguito con la seconda funzione di callback *update_output_div()*, la quale si occupa dell'aggiornamento del div inserito nella *app* successivamente al *Graph* come componente html, facendo semplicemente la return dei nuovi parametri di media, minima e massima calcolati all'ultimo campione ricevuto.

Sulla funzione per il calcolo del tempo per l'asse delle ascisse è importante precisare l'utilizzo della libreria *datetime* per la sua capacità di recuperare informazioni sul momento attuale e la facile conversione di questi valori in un formato ben comprensibile alla libreria *plotly*.

Un esempio del risultato finale viene mostrato di seguito. Si noti come per poter visualizzare il grafico l'utente si debba connettere all'interfaccia di loopback (*localhost* o 127.0.0.1) alla porta 8050 tramite un comunissimo browser web.



Servizio 3: Smart Light

Il terzo e ultimo servizio che abbiamo implementato è una lampadina smart, in grado di accendersi in base all'orario del crepuscolo e in grado di cambiare colore a seconda delle condizioni atmosferiche rilevate. Non essendo in grado di utilizzare una lampadina vera e propria, abbiamo utilizzato un led RGB presente in ogni kit base di Arduino per poter testare il nostro software. Per poter accedere alle informazioni riguardanti l'orario del crepuscolo e il tempo atmosferico di una data località, abbiamo sfruttato due API gratuite di tipo RESTfull. Per poter inviare i comandi di attuazione alla scheda Arduino abbiamo utilizzato il protocollo MQTT. La parte di codice utilizzato per ottenere le informazioni sul broker e il thread che rinnova la registrazione del servizio al Catalog verrà omessa in quanto è la medesima dei servizi visti precedentemente.

All'avvio del servizio, viene chiesto all'utente di scegliere una tra le cinque città italiane possibili. Infatti, la API che restituisce il meteo (http://www.metaweather.com/api) ha nel proprio database le informazioni riguardanti Torino, Milano, Venezia, Napoli e Roma. Una volta che l'utente ha selezionato la città di riferimento viene effettuata una richiesta HTPP di tipo GET all'indirizzo:

https://www.metaweather.com/api/location/search/?query=cittàSelezionata

Questa richiesta serve per poter ottenere una serie di informazioni indispensabili per poter richiedere il meteo al web server. Un esempio del testo ritornato dal server viene riportato per la città di Torino.

[{"title":"Torino","location_type":"City","woeid":725003,"latt_long":"45.070290,7.667680"}]

Nel nostro servizio estraiamo le informazioni riguardanti il campo “woeid”, che identifica in modo univoco la città all’interno del sistema gestito da metaweather, e il campo “latt_long” che rappresenta le coordinate della città. Questa informazione è importante per poter utilizzare la seconda API.

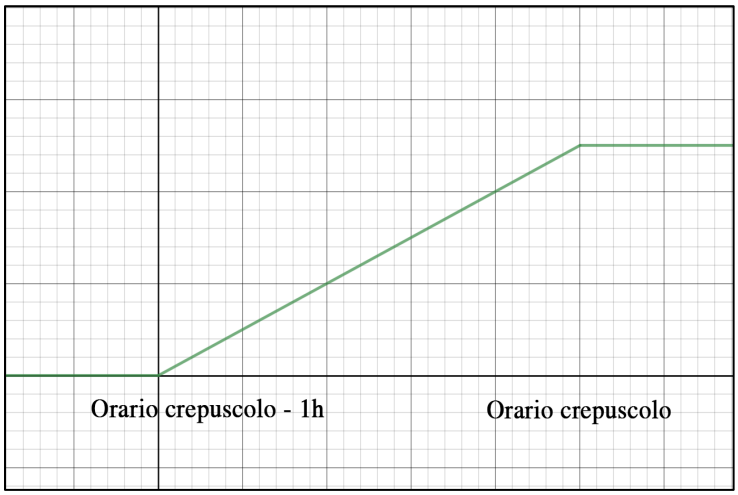
Dopo aver lanciato il thread che gestisce la registrazione del servizio, un secondo thread della classe *weatherThread* viene eseguito. Questo gestirà l’invio dei comandi ad Arduino tramite MQTT publish e avrà il compito di ottenere i dati relativi all’orario del crepuscolo e relativi al tempo atmosferico. La nostra idea è di permettere alla lampadina di accendersi in modo autonomo un’ora prima dell’orario di fine crepuscolo (ovvero quando si passa dal giorno alla notte), in modo da poter modulare l’intensità della luce gradualmente e non eseguire una semplice accensione programmata. Abbiamo anche deciso in maniera arbitraria che la luce si sarebbe spenta alle 2:00 del mattino in modo automatizzato. Abbiamo inoltre associato ad ogni tempo atmosferico un colore, creando un’analogia tra bel tempo/colore caldo e cattivo tempo/colore freddo. Per questo abbiamo suddiviso le varie condizioni metereologiche in tre diversi gruppi (bel tempo, meteo variabile, cattivo tempo) in modo da non dover gestire una decina di colori diversi. Ogni condizione atmosferica è stata mappata a un numero che identifica il gruppo di appartenenza all’interno del dizionario *map*. Di seguito viene riportata una tabella che mostra le nostre scelte.

Gruppo	**ID gruppo**	**Meteo**	**RGB**	**Colore**
Bel tempo	0	Clear Light clouds	R:240 G:60 B:0	
Meteo variabile	1	Showers Heavy clounds Light rain	R:60 G:240 B:180	
Cattivo tempo	2	Snow Sleet Hail Thunderstorm Heavy rain	R:0 G:60 B:240	

All'interno del metodo *run()* della classe *weatherThread* abbiamo eseguito una richiesta GET alla API *sunrise-sunset.org* (https://sunrise-sunset.org/api) che permette di ottenere informazioni riguardanti l'orario a cui sorge e tramonta il sole in una data località. L'informazione che ci interessa è l'ora di fine del crepuscolo che indica il momento della giornata in cui il cielo diventa completamente buio. La richiesta GET viene eseguita col seguente formato:

https://api.sunrise-sunset.org/json?lat=latitudine&lng=longitudine

Il testo che viene ritornato dal server è un json, nella quale è presente uno svariato numero di informazioni relative agli orari solari. Il nostro servizio estrapola il dato relativo alla fine del crepuscolo civile ("civil_twilight_end") tramite la funzione *strptime()* della libreria *datatime* e successivamente salva un secondo orario che si riferisce all'ora di accensione della lampadina, ovvero un'ora prima del crepuscolo. In questo modo abbiamo l'orario in cui la lampadina dovrà accendersi (inizialmente con luce molto fioca) e l'orario in cui la lampadina dovrà raggiungere la massima luminosità (al crepuscolo). Queste operazioni appena descritte vengono ripetute ogni 10800 secondi ovvero ogni 3 ore. Non avrebbe senso aggiornare dei dati che cambiano una sola volta al giorno con una frequenza maggiore. Un grafico della procedura che segue la lampadina viene mostrato di seguito.

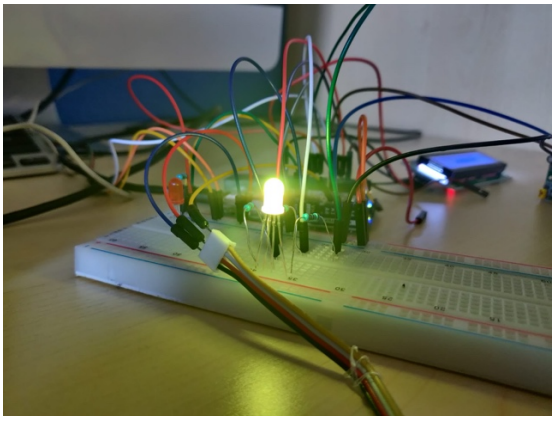


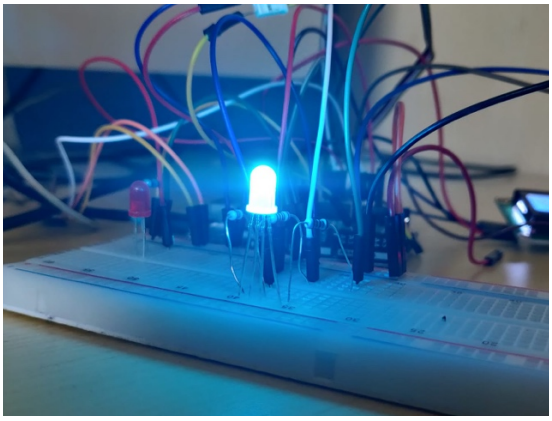
La lettura del tempo atmosferico avviene ogni due ore, tramite una GET alla API di metaweather. Utilizzando il codice della città, il server ritorna una lunga serie di informazioni sul meteo della località indicata. Un esempio di richiesta:

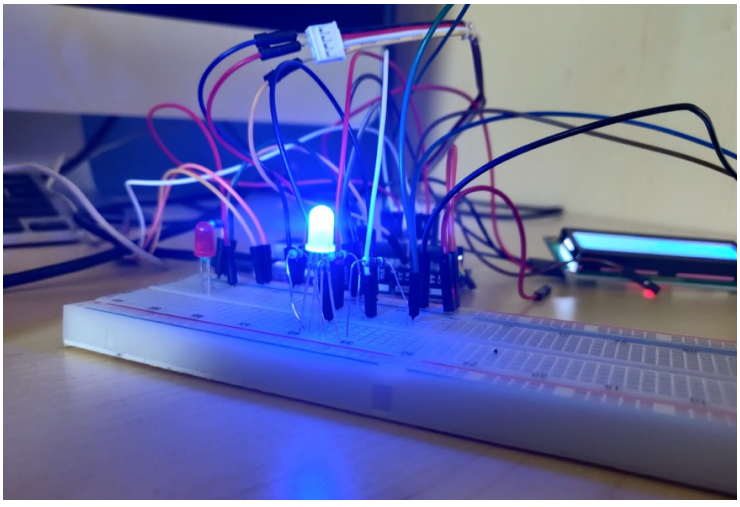
https://www.metaweather.com/api/location/cityCode

Dal testo in formato json ritornato, possiamo ottenere le attuali informazioni metereologiche all'interno del campo "weather_state_name".

Il metodo *updateRGB()* viene chiamato una volta al minuto e dopo aver processato le informazioni invia i comandi di attuazione al device. Il metodo confronta l'attuale orario con i due orari salvati in precedenza. Se il servizio si trova temporalmente dopo l'orario di fine crepuscolo, allora invierà un messaggio MQTT contenente i valori di fondo scala del relativo tempo atmosferico (per esempio 250,50,0 nel caso di bel tempo). Nel caso invece si trovasse all'interno dell'intervallo di crescita lineare della luminosità, allora aumenterebbe il valore di illuminazione di partenza (R:0 G:0 B:0) di un valore pari al corrispettivo valore di fondo scala diviso per 60. In questo modo, dopo un'ora di transizione, i valori raggiunti sarebbero equivalenti ai valori massimi (si noti che i valori di fondo scala sono tutti multipli di 60 in modo da non avere divisioni che ritornano numeri decimali). L'ultimo controllo verifica che siano superate le due del mattino e che l'ultimo valore inviato al device non sia quello settato per lo spegnimento del led ovvero 000. In questo modo la condizione non viene rispettata durante l'orario di transizione e di accensione e il led non verrà spento accidentalmente.







Considerazioni sull'utilizzo di una piattaforma distribuita

Avendo implementato lo Smart Controller in due diverse versioni, locale e distribuita dello stesso software, possiamo esprimere le nostre considerazioni su questi due diversi approcci di sviluppo.

La prima soluzione, essendo gestita solamente da Arduino, ci ha dato soprattutto problemi relativi alle scarse possibilità implementative del software e dell'hardware. Non abbiamo potuto usare più di una ISR poiché solo pochi GPIO supportavano tale funzionalità e talvolta alcuni dovevano essere utilizzati per altre funzioni (per esempio i PIN digitali 1 e 2 dovevano essere liberi per la comunicazione con il processore Linux). Dunque abbiamo optato per soluzioni più complesse da gestire come ad esempio i controlli dei tempi tramite la funzione *millis().* Inoltre, il debug è stato molto complesso e spesso abbiamo dovuto utilizzare delle "ineleganti" *Serial.print()* per poter verificare il corretto funzionamento delle varie componenti software. Vogliamo comunque sottolineare che nel complesso l'implementazione delle specifiche in linguaggio Arduino non è stato così complesso. Una volta terminata l'implementazione, le performance del dispositivo si sono dimostrate del tutto appropriate per un utilizzo in tempo reale dell'impianto di condizionamento. Il movimento e il rumore venivano rilevati in modo preciso e immediato e in pochissimi istanti i comandi di attuazione reagivano ai cambiamenti dell'ambiente.

La versione distribuita del Remote Smart Controller ci ha dato molti più problemi a livello di sviluppo. Non avendo basi di *Software Engineering* e non avendo mai dovuto programmare un insieme di componenti che avrebbero dovuto interagire, abbiamo avuto un approccio troppo empirico, provando a scrivere codice senza avere una visione di insieme. A livello di pura implementazione, è sicuramente stato più complesso per noi da gestire rispetto a scrivere un unico programma per Arduino anche se alla fine siamo riusciti a far lavorare il sistema in modo adeguato.

Un altro problema che abbiamo riscontrato nell'utilizzo di una piattaforma distribuita sono i tempi di reazione del device. In questo caso, infatti, Arduino mandava dati al client che dopo averli elaborati inviava comandi di attuazione. Per poter far si che il device mandasse messaggi MQTT, sono state sfruttate delle librerie Arduino che permettevano l'utilizzo del processore Linux. Queste librerie eseguivano comandi che si sono dimostrati abbastanza pesanti da svolgere per una scheda con potenza computazionale limitata. Di conseguenza si è perso il senso di "real time" che avevamo provato con l'implementazione in locale. Ovviamente un impianto di condizionamento non ha bisogno di tempi di risposta nell'ordine dei decimi di secondo ma a livello pratico le operazioni ci sono sembrate più macchinose nella versione distribuita. D'altro canto, le possibilità di sviluppo che questo approccio permette sono pressoché illimitate e funzionalità che su Arduino sarebbero impensabili, possono essere sviluppate abbastanza semplicemente in moltissimi modi diversi. Ad esempio, il servizio che gestisce il grafico della temperatura non sarebbe implementabile in una "versione Arduino", mentre noi abbiamo potuto scegliere tra molte librerie già esistenti e sviluppare il codice in modo semplice e in poco tempo.

Inoltre, non è da sottovalutare la possibilità di poter gestire un'innumerevole quantità di risorse, allargando la prospettiva da un sistema di gestione di una semplice casa a un palazzo intero, rendendo quindi incredibilmente flessibile sia la comunicazione dei dati raccolti tramite sensori che dei comandi di attuazione, con una semplice organizzazione gerarchica dei topics MQTT. In questa maniera sarebbero oltretutto supportabili in contemporanea una grande mole di utenti e di informazioni grazie ad un sistema altamente scalabile, anche per via della semplicità di soluzioni cloud disponibili, senza sottovalutare la possibilità di calcolo e di analisi che possono essere effettuate contemporaneamente alla raccolta dei dati.

Conclusioni

La soluzione dei laboratori sulla parte software è stata sicuramente più complessa rispetto al lavoro svolto su Arduino. La gestione di un sistema distribuito è stata complicata non solo per l'elevato numero di elementi diversi da dover gestire, ma anche perché le libertà implementative non hanno permesso di aver delle "linee guida" da seguire in modo agevole. Sicuramente questa esperienza è stata stimolante e formativa sotto molti punti di vista. Ci siamo dovuti interfacciare per la prima volta alla complessità di situazioni reali e abbiamo dovuto affrontare delle problematiche più simili a quelle della vita lavorativa che a quelle di un'esercitazione scolastica.

Un altro aspetto che non va sottovalutato è l'importanza della collaborazione necessaria per lo sviluppo in team, un concetto che nel corso di tre anni accademici non abbiamo mai potuto sperimentare concretamente. Lo svolgimento di questi laboratori sarebbe stato molto più complicato per una singola persona, mentre il collaborare è stato fondamentale per riuscire a ottenere un risultato soddisfacente.

Infatti, nonostante i vari ostacoli incontrati durante lo svolgimento, il sistema finale può essere considerato concluso con successo. Il software non è probabilmente utilizzabile da parte di un utente vero e proprio, considerate alcune scelte poco raffinate e sicuramente la presenza di alcuni bug. Tuttavia, le funzionalità richieste vengono eseguite correttamente e in modo stabile e il comportamento del software è sempre quello desiderato. Infatti, abbiamo eseguito dei test sia sui singoli componenti, con vari metodi di debug, sia sul sistema in toto e non abbiamo rilevato problematiche.