Laboratori Tecnologie per IoT
a.a. 2019/20

Cutaia Angelo, s180368
Tancredi Claudio, s244809

Indice

Laboratorio SW

Capitolo 1

Premesse

Tutto il codice usa il typing per creare autodocumentazione per l'IDE tramite i docstrings, di conseguenza è richiesto **python>3.6+**. Inoltre, sono stati usati molti meccanismi interni forniti da Cherrypy per rendere più agevole la fase di sviluppo.

Capitolo 2

LAB Software Part 1

Il primo laboratorio software è basato sul prendere dimistichezza con il framework per lo sviluppo di servizi REST chiamato Cherrypy. Il laboratorio è composto da quattro esercizi il cui scopo è utilizzare le meccaniche base di Cherrypy per realizzare dei servizi per la conversione della temperatura e per l'esposizione ed il salvataggio di una dashboard basata su freeboard. Per cercare di riutilizzare il codice del converitore è stato deciso di sviluppare una classe chiamata "**ConverterUtils**" situata all'interno del package temperature, nel modulo utils, la quale usa attributi di classe (comuni a tutte le istanze) ed un class method chiamato conversion al fine di effettuare la conversione di temperatura, senza bisogno di instanziare una nuova istanza ad ogni richiesta. Questa **ConverterUtils** viene chiamata dalle API sviluppate all'interno del package temperature nel modulo api, per convertire ogni richiesta che contiene dati validi. Inoltre Cherrypy, in caso di errori da comunicare al client, generalmente li manda in formato HTML, per cui è stata sviluppata una funzione "**jsonify_error**" che converte le eccezioni da HTML a JSON. Tale funzione sarà presente in ogni laboratorio che offrirà almeno un'interfaccia REST, e si troverà sempre nel package app, all'interno del modulo utils. Nelle API sarà sempre presente il decoratore "**@cherrypy.tools.json_out()**" perchè di default trasforma qualsiasi risposta data nel return in un JSON, mentre il decoratore "**@cherrypy.tools.json_in()**" sarà presente solo negli end point che richiedono un body in ingresso in formato JSON, operando automaticamente la conversione in dizionario Python. Nel caso dell'esercizio quattro del laboratorio, incentrato sulla dashboard, non è presente la **jsonify_error** perchè la dashboard è servita tramite HTML usando una directory statica, di conseguenza ha senso restituire gli errori in formato HTML. Per testare il codice si è deciso di scrivere degli unittest usando il tool fornito internamente da Cherrypy, per mostrare una completa padronanza di Cherrypy acquisita durante il corso delle lezioni.

Capitolo 3

LAB Software Part 2

Il secondo laboratorio è incentrato sullo sviluppo di un'infrastruttura IoT per la gestione di utenti, servizi e smart devices in maniera distribuita usando i protocolli REST ed MQTT e sul successivo testing delle sue funzionalità, mediante sketch Arduino e fake devices Python, usando direttamente il protocollo REST (ed il tool Postman) e/o il protocollo MQTT.

3.1 Catalog

La funzione del catalog è di offrire delle API REST per permettere di aggiungere, in real-time, devices, utenti e servizi e di rimuovere, ad ogni minuto trascorso, tutti i devices e servizi che non hanno contattato il catalog negli ultimi due minuti. Il timestamp viene controllato e gestito internamente dal catalog e non vengono utilizzati timestamps ricevuti dai devices, questo principalmente per evitare di avere incongruenze e inconsistenze sui dati. Le informazioni sono gestite tutte in formato JSON e per memorizzarle è stato deciso di utilizzare sqlite3, poichè esso permette di parsificare in maniera efficiente grandi quantità di dati e, essendo un database che non supporta la concorrenza in scrittura, attenua molti problemi dovuti al fatto che Cherrypy è un'applicazione multi-thread che usa un thread diverso per ogni richiesta. In poche parole, tramite sqlite3, si è potuto evitare di preoccuparsi di gestire la concorrenza mettendo numerose lock all'interno del codice, perchè è una criticità risolta di default da sqlite3 (per maggiori informazioni: documentazione). Il database viene interrogato periodicamente per eliminare i devices ed i servizi che hanno un timestamp "vecchio" usando un BackgroundTask fornito direttamente da Cherrypy; tale BackgroundTask non è altro che una sottoclasse della generica classe Thread e ripete in maniera periodica la funzione a cui è stato associato. Usando direttamente questa utility si è evitato di gestire manualmente un thread periodico lasciando la sua completa gestione a Cherrypy. Il catalog effettua anche delle parsificazioni dei dati in ingresso per cercare di estrapolare informazioni e memorizzarle in un formato più comprensibile allo sviluppatore di servizi. Questo

è stato fatto perchè i devices di tipo embedded hanno una memoria dalle dimensioni molto limitate, quindi si è preferito stabilire un formato comune ed il più leggero possibile per i devices così da avere il minor dispendio possibile di risorse hardware, senza rinunciare però ad una successiva astrazione dei dati. Questi discorsi di manipolazione dei dati valgono soltanto per i devices, di seguito saranno mostrati dei JSON in ingresso per dei devices REST e/o MQTT e la risposta del catalog quando verrà interrogato da un servizio sullo stato di questi devices.

ESEMPIO DEVICE IN INGRESSO CHE SUPPORTA SOLO MQTT

```
{
    "ID": "MQTT_DEVICE",
    "PROT": "MQTT",
    "IP": "test.mosquitto.org",
    "P": 1883,
    "ED": {
        "S": ["temp", "PIR", "noise", "SP"],
        "A": ["FAN", "led", "lcd"]
    },
    "AR": ["Temp", "FAN", "Led", "PIR", "noise", "SM",
        "Lcd"]
}
```

RISPOSTA DA PARTE DEL CATALOG AD UNA CURL

$ curl http://{ip_catalog}:{porta}/catalog/devices/MQTT_DEVICE

```
{
    "deviceID": "MQTT_DEVICE",
    "end_points": {
        "MQTT": {
            "ip": "test.mosquitto.org",
            "port": 1883,
            "end_points": {
                "subscribe": [
                    "temp/MQTT_DEVICE",
                    "PIR/MQTT_DEVICE",
                    "noise/MQTT_DEVICE",
                    "SP/MQTT_DEVICE"
                ],
                "publish": [
                    "FAN/MQTT_DEVICE",
                    "led/MQTT_DEVICE",
                    "lcd/MQTT_DEVICE"
```

```
                ]
            }
        }
    },
    "available_resources": {
        "MQTT": [
            "Temp",
            "FAN",
            "Led",
            "PIR",
            "noise",
            "SM",
            "Lcd"
        ]
    },
    "last_update": 1598521600
}
```

ESEMPIO DEVICE IN INGRESSO CHE SUPPORTA SOLO REST

```
{
    "ID": "REST_DEVICE",
    "PROT": "REST",
    "IP": "192.168.1.12",
    "P": 8000,
    "ED": {
        "S": ["temp", "PIR", "noise", "SP"],
        "A": ["FAN", "led", "lcd"]
    },
    "AR": ["Temp", "FAN", "Led", "PIR", "noise", "SM",
        "Lcd"]
}
```

RISPOSTA DA PARTE DEL CATALOG AD UNA CURL

$ curl http://{ip_catalog}:{porta}/catalog/devices/REST_DEVICE

```
{
    "deviceID": "REST_DEVICE",
    "end_points": {
        "REST": {
            "ip": "192.168.1.12",
            "port": 8000,
            "end_points": {
```

```
                "GET": [
                    "http://192.168.1.12:8000/temp",
                    "http://192.168.1.12:8000/PIR",
                    "http://192.168.1.12:8000/noise",
                    "http://192.168.1.12:8000/SP"
                ],
                "POST": [
                    "http://192.168.1.12:8000/FAN",
                    "http://192.168.1.12:8000/led",
                    "http://192.168.1.12:8000/lcd"
                ]
            }
        }
    },
    "available_resources": {
        "REST": [
            "Temp",
            "FAN",
            "Led",
            "PIR",
            "noise",
            "SM",
            "Lcd"
        ]
    },
    "last_update": 1598521842
}
```

ESEMPIO DEVICE IN INGRESSO CHE SUPPORTA MQTT E REST

```
{
    "ID": "MQTT_AND_REST_DEVICE",
    "PROT": "BOTH",
    "MQTT": {
        "IP": "test.mosquitto.org",
        "P": 1883,
        "ED": {
            "S": ["temp", "PIR", "noise", "SP"],
            "A": ["FAN", "led", "lcd"]
        },
        "AR": ["Temp", "FAN", "led", "PIR", "noise", "
           SM"]
    },
    "REST": {
```

```
        "IP": "192.168.1.12",
        "P": 8000,
        "ED":{
            "S": ["temp", "PIR", "noise", "SP"],
            "A": ["FAN", "led", "lcd"]
        },
        "AR": ["Temp", "FAN", "led", "PIR", "noise", "SM", "Lcd"]
    }
}
```

RISPOSTA DA PARTE DEL CATALOG AD UNA CURL

$ curl http://{ip_catalog}:{porta}/catalog/devices/MQTT_AND_REST_DEVICE

```
{
    "deviceID": "MQTT_AND_REST_DEVICE",
    "end_points": {
        "MQTT": {
            "ip": "test.mosquitto.org",
            "port": 1883,
            "end_points": {
                "subscribe": [
                    "temp/MQTT_AND_REST_DEVICE",
                    "PIR/MQTT_AND_REST_DEVICE",
                    "noise/MQTT_AND_REST_DEVICE",
                    "SP/MQTT_AND_REST_DEVICE"
                ],
                "publish": [
                    "FAN/MQTT_AND_REST_DEVICE",
                    "led/MQTT_AND_REST_DEVICE",
                    "lcd/MQTT_AND_REST_DEVICE"
                ]
            }
        },
        "REST": {
            "ip": "192.168.1.12",
            "port": 8000,
            "end_points": {
                "GET": [
                    "http://192.168.1.12:8000/temp",
                    "http://192.168.1.12:8000/PIR",
                    "http://192.168.1.12:8000/noise",
                    "http://192.168.1.12:8000/SP"
```

```
                ],
                "POST": [
                    "http://192.168.1.12:8000/FAN",
                    "http://192.168.1.12:8000/led",
                    "http://192.168.1.12:8000/lcd"
                ]
            }
        }
    },
    "available_resources": {
        "MQTT": [
            "Temp",
            "FAN",
            "led",
            "PIR",
            "noise",
            "SM",
                "Lcd"
        ],
        "REST": [
            "Temp",
            "FAN",
            "led",
            "PIR",
            "noise",
            "SM",
            "Lcd"
        ]
    },
    "last_update": 1598522057
}
```

3.2 Estensione MQTT del catalog

Per rendere il catalog compatibile con MQTT al fine di registrare devices che possano usare tale protocollo di comunicazione è stato deciso di sviluppare un plugin che facesse connettere il catalog, all'avvio, ad un broker MQTT (nel nostro caso "*test.mosquitto.org*") e lo facesse sottoscrivere al topic **"catalog/-devices"**, così da inoltrare nel bus interno di Cherrypy il payload dei messaggi ricevuti, permettendo a Cherrypy di gestire in maniera autonoma il passaggio di informazioni e memorizzare i dati dei devices seguendo gli stessi algoritmi di elaborazione di quelli ricevuti tramite REST. Sviluppando il plugin di MQTT in questo modo, avendo anche sqlite3 che gestisce con la sua lock interna la scrittura concorrente all'interno del database, si è ridotta notevolmente la complessità

delle operazioni da svolgere poiché tutta la comunicazione e sincronizzazione tra thread è internamente gestita da tutta l'infrastruttura multithread di Cherrypy e dal suo bus interno. Questa soluzione software permette di evitare di gestire direttamente thread vari e limitare al minimo indispensabile l'uso di lock, semafori e la gestione di problemi come la comunicazione efficiente tra thread e/o processi. In questo modo si è ottenuto un sistema molto stabile, cosa che verrà dimostrata nella pratica nella riproduzione della smart home e nei due servizi extra sviluppati per il laboratorio 4.

3.3 CLI Application

Era richiesto di sviluppare un client Python per testare le funzionalità REST del Catalog; a tale scopo è stata sviluppata una shell interattiva combinando l'uso del modulo della libreria standard cmd alla libreria requests. Si è deciso di svilupparla come se dovesse essere una utility per un admin per controllare lo stato del catalog, quindi si è cercato di rendere il più semplice possibile i comandi disponibili, inserendo tante descrizioni e dando la possibilità all'admin di inserire l'IP e la porta del catalog da testare, rendendo la shell totalmente indipendente dalla macchina su cui viene lanciata.

3.4 Fake devices

Per testare in maniera più accurata il catalog sono stati sviluppati due fake devices ed uno sketch Arduino per mandare sia dati reali sia fittizi, così da vedere la risposta effettiva della piattaforma.

3.4.1 Fake device REST

Questo fake device è stato totalmente sviluppato usando Cherrypy per esporre un end-point REST dove viene simulata una lettura di temperatura, restituendo un JSON a chi effettua la richiesta, proprio come se fosse un device con sensore di temperatura. Per eseguire la registrazione periodica al catalog si è usato il BackgroundTask fornito da Cherrypy, il quale ogni minuto effettua una POST.

3.4.2 Fake device MQTT

Per questo fake device si è usata la sola libreria paho. All'avvio il fake device si connette al broker preimpostato, pubblica tramite MQTT nel topic "*catalog/devices*" le informazioni da memorizzare nel catalog e poi, ogni tre secondi, pubblica il valore della temperatura calcolata, usando sempre il formato JSON. Per svolgere i compiti periodici di registrazione al catalog ed invio di temperatura si è preferito evitare l'utilizzo della classe Thread del modulo threading della libreria standard, si è usata invece la classe Timer, sempre del modulo threading, così da avere una soluzione più pulita e più facile da gestire in caso di improvvisa interruzione del flusso di esecuzione del codice.

Capitolo 4

LAB Software Part 3

Il terzo laboratorio è incentrato sullo sviluppo di servizi basati principalmente sul protocollo MQTT, i quali interagendo con il catalog e con i vari devices registrati, dovranno essere in grado di ricreare in maniera distribuita la smart home sviluppata nei laboratori hardware, spostando tutte le logiche dall'Arduino al cloud.

4.1 Servizi distribuiti

La traccia chiedeva di sviluppare servizi specifici per Arduino Yún. Tutti i servizi sviluppati hanno preso in considerazione il fatto che ogni deviceID rappresentante un Arduino Yún avesse una parte dell'ID comune a tutti, **"YUN"**, ed il resto generato in maniera randomica a runtime. Per comodità abbiamo scelto di vedere l'Arduino come unico device e non come nodo gateway che serve per accedere a più sensori ed attuatori. Si tratta di un problema di pura astrazione architetturale, nella piattaforma c'è già un middleware, il catalog, ed i devices possono a loro volta essere visti come nodi gateway verso sensori e/o attuatori, esponendo ciascuno di essi come un "device", oppure essere visti nella loro interezza, come device nel vero senso della parola. La nostra scelta mira a minimizzare l'uso della memoria perchè, in caso contrario, si sarebbero dovuti generare tanti JSON quanti erano i sensori/attuatori connessi all'Arduino, e mira a semplificare la gestione dell'intero ecosistema IoT. Inoltre, è stato supposto che nel catalog possano esserci anche devices registrati su broker diversi da quello utilizzato dal catalog stesso, potenzialmente registratisi al catalog usando una POST ma che espongono end-points solo su MQTT. Per questa ragione si è deciso di realizzare, a livello software, un "ponte" tra broker diversi nel caso in cui fossero presenti, per rendere totalmente distribuita l'applicazione. Per dar prova del conseguimento dell'obiettivo sono stati realizzati anche dei fake devices appoggiati su un broker diverso da Mosquitto. Ogni servizio, nel caso in cui non trovi almeno un device compatibile, ovvero che offra le risorse richieste dal servizio per funzionare, riprova ad interrogare il catalog dopo un interval-

lo di tempo opportuno chiedendo tutti i devices o servizi attivi (**attenzione**, un servizio può basarsi su altri servizi e non necessariamente sui soli devices). Ottenuta la lista di informazioni, estrapola quelle utili alla propria attività. In caso non si fosse ottenuta nessuna informazione utile al proprio funzionamento, viene interrotto immediatamente l'update della propria registrazione al catalog, il quale se dovesse notare un'inattività superiore ai due minuti provvederà a cancellerà il servizio, perchè non più attivo.

4.2 Smart Home

L'applicazione smart home distribuita è stata sviluppata usando lo stesso approccio dei servizi menzionati prima. Il risultato è stata un'applicazione in grado di gestire più smart home in real-time. La differenza principale tra l'approccio embedded e quello distribuito è che nel caso di quello embedded vi è un minore costo delle risorse, ma l'applicazione risultante può essere usata solo in un environment ristretto, mentre l'approccio distribuito garantisce maggiore scalabilità, affidabilità e possibilità di integrare nuove funzionalità (potenzialmente si possono sviluppare infiniti nuovi servizi) a costi ovviamente maggiori. Nel caso di un'applicazione distribuita che possieda al proprio interno una struttura come il catalog sviluppato nello scorso laboratorio, il middleware permette di effettuare vari gradi di astrazione suddividendo più servizi sul cloud, e quindi di modificare piccole porzioni di codice ottenendo logiche riutilizzabili ed astrazione pura dai sensori ed attuatori. A livello di sicurezza ed affidabilità è anche superiore l'approccio distribuito, perchè in caso di guasti o presenza di nuovi devices o servizi utili il sistema si riconfigura in maniera autonoma e smart. Infatti, se si dovesse o si volesse cambiare un dispositivo (perché malfunzionante, perché si intende fare un upgrade con tecnologie recenti, ecc.), questo potrebbe essere fatto a runtime, a patto che il nuovo dispositivo sia conforme agli standard della piattaforma in uso ed alle API. Un altro vantaggio, poi, è sicuramente quello di poter far coesistere nell'ecosistema tutto un insieme di protocolli di comunicazione trasparenti, quali Bluetooth, ZigBee, ecc., per l'interazione con i devices. Sarebbe possibile, inoltre, dockerizzare l'architettura distribuita al fine di favorire la distribuzione ed installazione di nuovi servizi su macchine diverse, andando ad aumentare la sicurezza dell'infrastruttura in caso di attacchi informatici e suddividendo l'insieme dei servizi sviluppati in microservizi. Un contro dell'aproccio distribuito è il fatto che più si scala in parallelo la capacità dell'applicazione (ad esempio affittando altri "cloud" su cui far girare l'applicazione), più aumentano i costi, quindi è necessario organizzare bene l'infrastruttura perché in applicazioni reali i dati dei device dell'IoT sono così numerosi che senza un'ottima architettura per astrarre e collezionare dati non si è in grado di realizzare applicazioni stabili. Proprio la grande mole di dati a disposizione permette di realizzare algoritmi dotati di un'intelligenza "collettiva", che possano essere utili nella gestione di interi edifici, estrapolando informazioni sulle abitudini degli utenti al fine di migliorare la qualità di vita degli stessi. Altro vantaggio dell'approccio distribuito è sicuramente la possibi-

lità di gestione remota di dispositivi (ad esempio macchinari aziendali), la quale permette di soddisfare esigenze particolari degli utenti finali.

Capitolo 5

LAB Software Part 4

Lo scopo del quarto laboratorio è espandere la piattaforma distribuita, realizzando almeno due servizi che possano essere utili all'utente. Pertanto si è deciso di sviluppare dei servizi che segnalino agli utenti, uno tramite e-mail e l'altro tramite messaggio su Telegram, il fatto che la temperatura degli Arduino registrati al catalog sia fuori dal range di buon funzionamento che di default è stato impostato tra 0 e 30 gradi Celsius. A tal scopo, torna utile il servizio 3 sviluppato nel laboratorio 3, poiché gestiva i led degli Arduino a seconda della temperatura e segnalava su un topic specifico lo stato del dispositivo (allarme true = fuori range, allarme false = ok). In questa maniera si è potuto semplificare lo sviluppo del servizio e-mail e del servizio Telegram, perchè così avrebbero semplicemente raccolto le informazioni da questo servizio comune di allarme ed effettuato le opportune operazioni. Oltre a questi servizi, sono stati riutilizzati il fake device del servizio 3 del laboratorio 3 e l'annesso sketch Arduino, per cercare di rendere il più realistico possibile lo scenario. Per inserire gli utenti e gli id delle chat Telegram in maniera user-friendly è stato deciso di sviluppare anche due Terminal User Interfaces, una per ogni servizio, usate come utility.

5.1 TUI

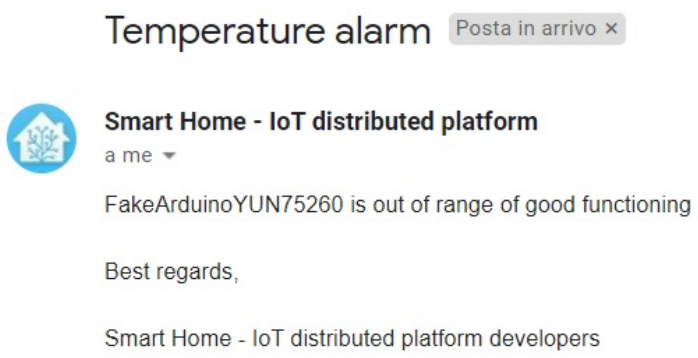
Le interfaccie grafiche da terminale sono state sviluppate usando la libreria python-prompt-toolkit. Tramite questo modulo esterno è stato molto semplice ottenere un'interfaccia carina e semplice da usare per poter inserire utenti nei due servizi aggiuntivi citati in precedenza. Inoltre è compatibile anche con il terminale di Windows, cosa non affatto scontata, in quanto generalmente le librerie per le TUI non sono multipiattaforma.

5.2 Servizio e-mail

Per realizzare il servizo di e-mail si è attuata la medesima astrazione degli altri servizi già sviluppati (bridge tra broker, ecc.) e si è deciso di usare il proto-

collo SMTP e il modulo standard di Python smtplib per mandare e-mail agli utenti (anche perché, in quanto modulo standard, favorisce il raggiungimento dell'obiettivo prefissato dal nostro gruppo, che era quello di approfondire quanto più possibile la conoscenza di tale linguaggio). Il catalog inizialmente non avrà utenti registrati, quindi per inserirli si può ricorrere a Postman o, per semplicità, alla shell utility del servizio e-mail.

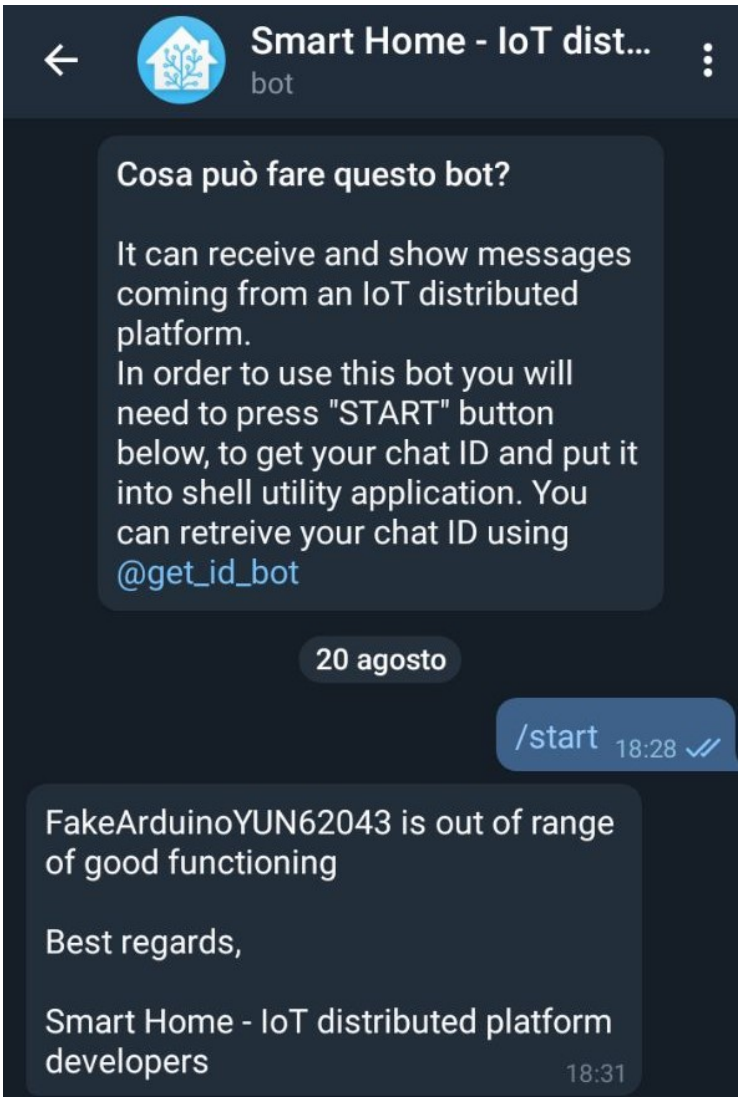
Il risultato ottenuto è il seguente:



5.3 Servizio Telegram

Il servizio Telegram per mandare messaggi agli utenti prevede che questi abbiano preventivamente effettuato delle azioni: avviare il Bot; ottenere il proprio chat id e fornirlo al servizio tramite TUI (e publish su topic specifico). L'interfaccia grafica da terminale è fondamentale in tal senso, perché spiega bene come ottenere tali informazioni e si occupa di contattare l'end-point del servizio Telegram per memorizzare i dati. Per realizzare questo servizio abbiamo usato la libreria telegram-python-bot.

Il risultato ottenuto è il seguente:


Smart Home - IoT dist...
bot
Cosa può fare questo bot?
It can receive and show messages coming from an IoT distributed platform.
In order to use this bot you will need to press "START" button below, to get your chat ID and put it into shell utility application. You can retreive your chat ID using @get_id_bot
20 agosto
/start 18:28
FakeArduinoYUN62043 is out of range of good functioning
Best regards,
Smart Home - IoT distributed platform developers
18:31

Capitolo 6

Considerazioni finali

Al termine di questa esperienza di laboratorio riteniamo di aver appreso i concetti fondamentali del corso e di aver anche approfondito alcuni aspetti implementativi.
Il gruppo è costituito da uno studente full-time e da uno studente-lavoratore con un background di conoscenze e competenze notevolmente diverso. Tuttavia, nello svolgimento dei laboratori, questo aspetto non è stato affatto limitante, rivelandosi invece come un'opportunità di arricchimento e crescita personale.
Abbiamo imparato a collaborare, anche a distanza viste le misure emergenziali in atto; abbiamo sviluppato soft skills importanti; ci siamo prefissati degli obiettivi e degli standard elevati, per la buona riuscita del progetto di piattaforma IoT.
Sicuramente nello sviluppo della piattaforma ci sono state delle criticità, sia HW, sia SW, ma con determinazione sono state ampiamente risolte, ottenendo un prodotto finale flessibile e stabile.
L'esperienza è stata sicuramente stimolante e interessante.