

Università degli Studi di Verona

DIPARTIMENTO DI SCIENZE E INGEGNERIA

Corso di Laurea Magistrale in Ingegneria e Scienze informatiche

TESI DI LAUREA MAGISTRALE

Automazione di test di accettazione per dispositivi IoT embedded integrati nel cloud

Candidato:

Alessandro Righi

Matricola VR432403

Relatore:

Mariano Ceccato

Indice

1	Introduzione	1
1.1	Il progetto IRSAP radiatore elettrico	2
1.2	Necessità di test di un dispositivo IoT	3
1.2.1	Prassi attuale testing	5
1.2.2	Test durante lo sviluppo	6
1.2.3	Test di accettazione interna	6
1.2.4	Test di accettazione del cliente finale	7
1.3	Problemi dell'approccio attuale	8
1.4	Requisiti software per l'automazione	9
2	Lavori correlati	11
2.1	Strumenti di unit test	11
2.2	Strumenti commerciali di test elettronici	12
2.3	Conclusioni	12
3	Approccio	13
3.1	Caratteristiche hardware	13
3.1.1	Scheda elettronica	13
3.1.2	Modulo Telit	15
3.2	Caratteristiche software	18
3.2.1	Stati interni del dispositivo	18
3.2.2	Termoregolazione	19
3.2.3	Comunicazione cloud	21

3.2.4	API di configurazione locale	26
3.2.5	HMI	27
3.3	Intefacciamento con il sistema di test	28
3.3.1	Interazione con il dispositivo fisico	29
3.3.2	Interazione con la sola elettronica	29
3.3.3	Esecuzione del firmware in un emulatore	30
4	Implementazione	31
4.1	Interfacciamento hardware	31
4.2	Motore di esecuzione test	32
4.2.1	Interfacciamento con l'I/O	34
4.2.2	Interfacciamento con il cloud	34
4.2.3	Interfacciamento con il Wi-Fi	36
4.3	Integrazione continua	38
5	Validazione sperimentale	41
5.1	scenari da testare	41
5.1.1	Abbinamento del dispositivo ad un impianto	41
5.1.2	Downgrade del firmware mediante API REST locale . .	45
5.1.3	Aggiornamento di un dispositivo mediante OTA MQTT	46
5.1.4	Ripristino di fabbrica di un dispositivo connesso . . .	48
5.1.5	Ripristino di fabbrica di un dispositivo disconnesso . .	50
5.1.6	Termoregolazione di base	53
5.1.7	Attivazione modalità stand-by	56
5.1.8	Funzionamento disconnesso	58
5.2	Tempi di esecuzione	61
6	Comclusioni	63
6.1	Futuri radiatori elettrici	63
6.2	Test di dispositivi RF	65
6.3	Test di un termostato	66
6.4	Ringraziamenti	67

Introduzione

Oggi giorno nelle nostre case ci sono sempre più prodotti connessi, dalle lavatrici, ai televisori, fino agli impianti domotici che consentono di controllare la nostra casa mediante un comando vocale anche quando ci si trova dall'altra parte del pianeta.

Questi dispositivi svolgono anche funzioni critiche per il nostro benessere domestico, quale ad esempio il controllo della temperatura ambientale, che è oggetto del mio lavoro in IOTINGA.

IOTINGA s.r.l. nasce con lo scopo di aiutare altre aziende nel realizzare e commercializzare dispositivi Internet of Things (IoT). IOTINGA si distingue dagli altri concorrenti per un'attenzione particolare alla componente software, in tutte le sue sfaccettature, dall'interazione fisica con le periferiche hardware, alla gestione del dato mediante un'infrastruttura realizzata con tecnologie cloud serverless, fino alla sua presentazione ai consumatori, mediante realizzazione di applicazioni Android/iOS.

La mia esperienza in IOTINGA inizia nel Febbraio 2020. In questi 3 anni ho avuto l'occasione di vedere crescere l'azienda, e fornire il mio contributo nello sviluppo del progetto “IRsap NOW”, che ho avuto modo di seguire in prima persona fin dalla sua fase embrionale.

IRsap NOW è l'ecosistema domotico che integra al proprio interno tutti i prodotti connessi di IRSAP s.p.a., una grande impresa rodigina leader nel settore del comfort termico. Storicamente produttrice di radiatori, inventrice del termoarredo, si distingue oggi per prodotti dal design altamente ricercato,

nonché dall'elevato contenuto tecnologico, quali impianti di Ventilazione Meccanica Controllata (VMC), Radiatori Elettrici (RE) connessi, e sistemi di gestione remota di impianti di riscaldamento.

1.1 Il progetto IRSAP radiatore elettrico

All'interno di questa piattaforma si innesta il prodotto in esame, ovvero la gamma di RE connessi IRSAP.

Questi dispositivi sono dei corpi scaldanti del tutto simili ai normali radiatori idraulici in cui il riscaldamento viene però fornito da una resistenza elettrica alimentata dalla rete.

Il catalogo si compone di decine di prodotti, uno su tutti il “Polygon” (Figura 1.1), vincitore del “CES Best of Innovation 2022” (Figura 1.2) nella categoria Home Appliances, nonché di altri prestigiosi premi a livello internazionale, quali “Red Dot Design”, “German Design”, “AIFA” grazie al suo design innovativo ed al suo contenuto tecnologico, a cui noi di IOTINGA abbiamo contribuito.



Figura 1.1: Polygon

“Polygon” è dotato internamente di elettronica in grado di connettersi mediante Wi-Fi direttamente al cloud “IRsap NOW”, ed integra oltre alla funzione scaldante anche un’illuminazione ambientale LED colorati per un’illuminazione ambientale.

Al momento della scrittura di questo documento (Febbraio 2023) e ad un anno dal lancio del prodotto sul mercato sono stati installati ed sono utilizzati attivamente dai clienti circa 3000 radiatori elettrici smart IRSAP.

Mi sono occupato in prima persona dello sviluppo del firmware del dispositivo nella sua interezza, mentre alcuni colleghi hanno seguito la parte di progettazione hardware (che comunque è stata affidata da un'azienda esterna) e di integrazione all'interno dell'ecosistema cloud e della app.



Figura 1.2: IRSAP ed IOTINGA al CES 2022

1.2 Necessità di test di un dispositivo IoT

La caratteristica fondamentale per questi prodotti è l'affidabilità. Infatti nessun utente installerebbe in casa propria un dispositivo che non è in grado di svolgere la funzione per la quale è stato acquistato.

A maggior ragione i danni derivanti dal malfunzionamento di un impianto di riscaldamento possono coinvolgere non solo cose ma anche estendersi a persone ed animali domestici.

È tassativo prestare attenzione alle problematiche che si possono verificare in utenza, le quali non sono solo un danno per il cliente stesso ma anche per l'azienda produttrice:

- la prima impressione sul cliente è quella che conta, se il cliente si ritrova un prodotto che funziona male o addira non svolge la funzione prevista ne parlerà male, anche mediante recensioni negative online, e creerà un danno d'immagine all'azienda difficilissimo da sanare
- quando il problema si verifica dal cliente è complicata la diagnostica. I clienti, e spesso anche gli installatori stessi, non hanno competenze tecniche o il tempo da dedicare nel supportare il produttore nella ricerca del problema
- se il problema non è risolvibile mediante un aggiornamento firmware è necessario effettuare un reso, che ha dei costi molto elevati per il produttore, in quanto durante il trasporto molto spesso il prodotto viene danneggiato e quindi deve essere rimpiazzato con un nuovo

È di fondamentale importanza assicurarsi di identificare il prima possibile quanti più problemi possibili prima che il prodotto arrivi nelle mani dell'utente finale.

In tutto questo il software ricopre un ruolo sempre più da protagonista, in quanto per garantire la connettività al cloud è necessario gestire una complessità elevata.

I prodotti della precedente generazione utilizzavano il software per una mera gestione delle periferiche fisiche del prodotto, senza la necessità di interfacciarsi con sistemi terzi. Al contrario i prodotti della attuale per svolgere tutte le funzioni di integrazione cloud in maniera sicura richiedono un livello aggiuntivo di astrazione, ovvero quello di un Real Time Operating System (RTOS).

Anche l'hardware stesso è più evoluto, infatti si passa dalle piattaforme ad 8 bit ai microcontrollori a 32, con funzionalità sempre più assimilabili a quelle di un sistema general purpose, quali ad esempio una gestione di programmazione concorrente, uno stack di rete TCP/IP, ed una gestione della memoria virtual con MMU.

Un'altra differenza rispetto al passato è la possibilità di aggiornare il software dopo che il prodotto lascia la fabbrica, mediante aggiornamenti di tipo Over The Air update (OTA). Questo si rende necessario non solo per la mera introduzione di nuove funzionalità in un prodotto esistente, ma anche per mantenere il software al passo con l'evoluzione degli altri sistemi a cui esso si collega, quale ad esempio modifiche nei protocolli di rete dettate dall'arrivo di nuovi standard.

Sebbene l'aggiornamento consenta di risolvere problemi dopo che il dispositivo ha lasciato la fabbrica esso comporta anche una criticità, in quanto vi è il rischio di introdurre altri problemi in prodotti che fino a quel momento non li avevano, andando a creare un disservizio.

Bisogna infine tener conto che un prodotto di questo tipo segue un ciclo di vita molto lungo rispetto ad esempio a PC o smartphone, che può tranquillamente superare i 10 anni dalla data di immissione nel mercato, e l'utente si aspetta che in tutti questi anni possa continuare ad utilizzarlo come il giorno in cui lo ha acquistato.

Conseguentemente diventa prioritario garantire i massimi livelli di qualità possibile sul software rilasciato. Questo non si limita alla semplice assenza di bug, in quanto questa è una garanzia che matematicamente è impossibile offrire, ma anche all'adottare delle procedure procedure tali che consentano, dal momento che un bug si presenta, di individuarlo e sistemarlo nel minor tempo possibile.

Tracciabilità e mantenibilità del codice sono quindi parole chiave, la prima garantisce che sia sempre possibile risalire all'esatta versione del codice per il quale viene segnalato un problema di modo da poterlo riprodurre, la seconda invece assicura che la soluzione al problema sia implementabile nel minor tempo possibile e senza il rischio di regressioni.

Infine l'altro punto cardine è l'eseguire test metodologici prima che il firmware venga rilasciato al pubblico, sia tramite aggiornamento OTA che tramite installazione in fabbrica su di un nuovo prodotto.

1.2.1 Prassi attuale testing

Allo stato attuale abbiamo 3 momenti di validazione di un rilascio:

1. test durante lo sviluppo
2. test di accettazione interna (effettuati da IOTINGA)
3. test di accettazione del cliente finale (effettuati da IRSAP)

1.2.2 Test durante lo sviluppo

Quando uno sviluppatore finisce di implementare una nuova funzionalità o risolve un bug prima di considerare l'attività conclusa ed integrare il proprio lavoro nel ramo di sviluppo principale ed effettua i propri test.

Questi si occupano sia di verificare che quanto è stato implementato è conforme alla specifica approvata dal cliente (nel caso di nuove funzionalità) oppure che il bug sia stato risolto, sia che non siano stati modificati il funzionamento del sistema nelle parti che sono state impattate dalla modifica.

Tali test sono a discrezione dello sviluppatore, che avendo modificato il codice sa quali comportamenti sono impattati dalla modifica che ha realizzato e quindi devono essere provati.

Questi test possono essere automatizzati facilmente mediante la creazione di unit-test, che possono essere usati per validare sia le parti nuove che assicurarsi che non vi siano regressioni che fanno smettere di funzionare quanto prima andava.

1.2.3 Test di accettazione interna

Questi test sono effettuati prima di ogni rilascio verso il cliente IRSAP.

Si occupano di validare che il software garantisca il funzionamento di una serie di casi d'uso definiti critici, senza i quali il prodotto non sarebbe utilizzabile. Solo se una versione del software passa tutti questi test può essere consegnata al cliente.

Essi si pongono dal punto di vista dell'utente finale, pertanto sono effettuati su un hardware completo, isolato però dal resto del sistema, ovvero dalla componente cloud e dall'applicazione mobile. Questo per evitare che ci sia il dubbio che il bug sia nel cloud o nella app anziché nel dispositivo stesso.

Attualmente vengono effettuati seguendo un documento contenente una serie di scenari da testare. Ognuno di questi è diviso in passi, e per ognuno dei quali viene indicato:

- azione da compiere: un'operazione da effettuare sul sistema mediante l'interazione fisica con il dispositivo (pressione di pulsanti) oppure mediante cloud (tramite un apposito strumento che consente di simulare i messaggi inviati dal cloud e dalla app)
- risultato atteso: postcondizioni da verificare dopo aver effettuato l'azione, espresso in linguaggio non formale, ad esempio “i LED sono rossi” oppure “entro 5 secondi viene inviato al cloud un messaggio”. Nel caso la postcondizione sia verificata è possibile procedere al passo successivo, altrimenti il test viene interrotto con esito negativo, e deve essere segnalato ad uno sviluppatore il problema.

Preferibilmetne questa procedura viene fatta eseguire da chi non ha preso parte allo sviluppo del sistema stesso. Questo per evitare che chi esegue la procedura, conoscendo le logiche interne del software, possa essere portato a saltare o ignorare determinati passaggi in quanto “ovvi”, mentre chi non conosce il sistema è più propenso a seguire i passaggi alla lettera e segnalare ogni singolo comportamento discordante con quanto atteso.

1.2.4 Test di accettazione del cliente finale

Sul cliente finale ricade la responsabilità (anche a livello legale) del prodotto che viene immesso sul mercato con il proprio nome sopra, e questo include anche il software. Questo comporta il fatto che a sua volta deve svolgere dei test per assicurarsi che il software sia conforme a quanto atteso, e nel caso segnalare i problemi riscontrati in maniera tale che vengano corretti.

Questi test sono volti a testare tutte le funzionalità del prodotto in tutte le loro possibili configurazioni, anche mediante l'ausilio di strumentazione altamente specializzata quali camere climatiche per valutarne l'efficacia di termoregolazione.

Nel caso questi test abbiano successo l'artefatto testato passa da stato di candidato al rilascio a produzione, e viene quindi installato in fabbrica su tutti

i nuovi radiatori prodotti, nonché viene lanciato un aggiornamento OTA su tutti i dispositivi già installati presso i clienti finali.

1.3 Problemi dell'approccio attuale

L'approccio attuale, basato sul documento di test con fasi da seguire, presenta alcune problematiche:

- la procedura è ripetitiva, e questo può facilmente introdurre l'operatore che esegue i test a commettere errori, con dolo o meno
- l'esecuzione dei test porta via tempo, che altrimenti potrebbe essere dedicato a svolgere altre mansioni
- i test sono scritti in linguaggio naturale, non formale, e questo lascia spazio a libera interpretazione dalla persona che esegue i test
- il fatto che sia un costo eseguire la procedura fa sì che questa sia limitata nel numero di scenari che sono effettivamente testati
- per quanto detto al punto precedente il numero di build effettivamente testate è un sottoinsieme di quelle che vengono effettivamente effettuate, il che significa che i rilasci verso il cliente avvengono con meno frequenza, e si tende ad accorpare più funzionalità in maniera tale da effettuare un unico ciclo di test.

A questo si aggiunge il fatto che man mano che i prodotti vengono venduti il cliente ci chiede sempre più test ad ogni release, in quanto un potenziale bug va ad impattare un numero sempre maggiore di utenti (e quindi il danno potenziale per il produttore è sempre più grande).

Viene quindi da chiedersi se, vista la ripetitività e schematicità di queste operazioni, sia possibile farle eseguire ad un computer, senza perdite di generalità rispetto all'esecuzione manuale.

1.4 Requisiti software per l'automazione

Abbiamo quindi stabilito i seguenti requisiti per un sistema di automatizzazione dei test di accettazione:

- il sistema deve testare esattamente il binario che poi viene rilasciato agli utenti finali. Questo perché ogni alterazione, come potrebbe essere una ricompilazione del codice, può andare ad aggiungere errori e quindi invalidare i test effettuati
- l'ambiente di esecuzione (runtime environment) sul quale il test viene eseguito deve essere quanto più vicino possibile all'ambiente reale. Idealmente il test viene effettuato sullo stesso hardware, di modo da divanare ogni possibile dubbio che il firmware una volta installato sull'hardware abbia comportamenti differenti
- l'ambiente deve poter eseguire il firmware in differenti configurazioni di hardware che esso supporta (attualmente supporta due tipologie di hardware, e ne verranno aggiunte altrettante due a breve)
- il sistema deve avere abbastanza flessibilità per poter validare almeno le funzionalità che oggi sono teste manualmente
- deve essere sufficientemente semplice andare ad aggiungere nuovi scenari e configurazioni da provare al sistema
- deve essere possibile integrare lo strumento all'interno del flusso di CI attualmente in uso in azienda, di modo che ogni release del firmware prodotta venga testata senza necessità di un intervento manuale

Capitolo 2

Lavori correlati

Vediamo ora cosa è disponibile in commercio per fare quanto descritto al capitolo precedente.

2.1 Strumenti di unit test

Una prima soluzione è quella di utilizzare gli strumenti di unit-test anche per effettuare i test di accettazione. È possibile infatti andare ad effettuare il mock di tutte le interfacce utilizzate dall'SDK al fine di poter eseguire singoli test-case all'interno del binario.

Questa soluzione presenta il vantaggio di poter testare in maniera veloce ed agevole ogni nuova release del firmware, ma al contempo presenta alcuni svantaggi che la rendono inadatta allo scopo di approvare un nuovo firmware per la produzione.

Il problema principale è che viene testato il codice facendolo girare in un ambiente simulato, che per quanto simile all'ambiente reale non sarà mai completamente uguale. Soprattutto quando ci si interfaccia con periferiche complesse, come il Wi-Fi, è fondamentale che l'ambiente sul quale vengono eseguiti i test sia quanto più uguale possibile al runtime effettivo.

In questo caso la soluzione migliore è effettuare i test facendo eseguire il firmware direttamente all'hardware sul quale dovrà girare.

2.2 Strumenti commerciali di test elettronici

In commercio esistono vari strumenti che consentono di testare un dispositivo elettronico andando a comandare i vari input/output con i quali si interfaccia verso il mondo esterno.

Uno di questi software è TestStand della National Instrument¹. Questo software è infatti pensato per l'interfacciamento con apparecchiature di misura prodotte dalla medesima azienda, quali ad esempio oscilloscopi, multimetri da banco, generatori di frequenza, ecc. e consente di automatizzare le operazioni di test e misura.

Come questo esistono altri software, principalmente proprietari e sviluppati internamente dalle singole aziende elettroniche, che svolgono la medesima funzione. Questi software tipicamente si interfacciano con la scheda hardware attraverso una fixture, composta da un letto di aghi conduttori che vanno a fare contatto con dei test-point sulla scheda da testare.

Questi sistemi possono benissimo essere utilizzati anche per testare il firmware. Tuttavia non è il loro scopo primario, essendo pensati per identificare in primo luogo difetti nella produzione fisica delle schede. In particolare non offrono routine per testare componenti complesse come Wi-Fi o comunicazione con il cloud, ma si limitano a fare verifiche fra gli input e gli output.

Inoltre l'hardware di questi dispositivi è molto costoso, essendo pensato per effettuare misurazioni precise a livello fisico, cosa che è fondamentale per identificare difetti sull'hardware, ma lo è meno per testare la release del firmware (dove si assume che l'hardware sul quale viene lanciato il test sia correttamente funzionante).

2.3 Conclusioni

In generale abbiamo visto che il software che rispondeva ai nostri requisiti non esisteva già sul mercato, sia come software commerciale che come software open-source.

¹<https://www.ni.com/it-it/shop/software/products/teststand.html>

Approccio

Prima di descrivere l'implementazione è necessario fare un passo indietro ed andare ad analizzare con più attenzione il dispositivo RE preso in esame.

3.1 Caratteristiche hardware

Il corpo principale dei radiatori elettrici di IRSAP è del tutto identico al radiatore idraulico, con la differenza che il calore non viene fornito dall'acqua che circola nell'impianto ma da una resistenza elettrica, installata all'interno del radiatore. Il tutto viene riempito con del glicole, per garantire una protezione antigelo, e viene sigillato con dei tappi al posto dei rubinetti di attacco per la mandata/ritorno dell'acqua calda.

3.1.1 Scheda elettronica

Attualmente la gamma di radiatori elettrici IRSAP elettrificati smart include 17 prodotti distinti, ma ogni anno ne vengono aggiunti di nuovi. Per tutti questi progetti è quindi fondamentale, per questioni di ottimizzazione dei costi, avere un'elettronica quanto più possibile simile.

Attualmente vi sono due tipologie di scheda logica che viene installata all'interno dei prodotti:

- *luxury*: è la versione più basilare, che viene utilizzata sulla linea di radiatori da bagno (chiamati anche “scaldasalviette”). Essendo installata

in ambiente particolarmente umido e con il rischio di schizzi d'acqua deve sopportare un grado IP (misura standard del grado di protezione per componenti elettrici) superiore all'altro modello

- *design*: è la versione riservata ai prodotti design. È la più completa, che offre anche modularità essendo che tutte le periferiche (pulsantiera, sensore di temperatura, sensore VOC, LED) sono collegati alla stessa con dei cavi, il che consente di inglobare l'elettronica all'interno della carcassa in maniera da renderla invisibile, cosa molto importante per non rovinare il design del prodotto. Offre inoltre funzionalità aggiuntive, quali un sensore di qualità dell'aria (in grado di rilevare i valori di VOC e CO₂) e la possibilità di controllare una striscia LED Red Green Blue White (RGBW), utilizzata su alcuni modelli per un'illuminazione ambientale.

Ogni scheda elettronica, sia essa di tipo *luxury* che *design*, include le seguenti periferiche hardware:

- un modulo Wi-Fi Telit GS2200M
- circuito di alimentazione a tensione di rete (230V AC)
- un circuito composto da un relè combinato ad un triac dedicato al controllo della cartuccia, ossia l'elemento riscaldante inserito all'interno del corpo del radiatore
- una sonda di temperatura Negative Temperature Coefficient (NTC) usata per misurare la temperatura ambiente
- una pulsantiera dotata di due pulsanti capacitivi e di LED RGB di illuminazione come feedback verso l'utente
- un buzzer utilizzato per un feedback uditorio della pressione dei pulsanti
- un circuito di lettura per il segnale “Fil Pilote”, uno standard francese per il controllo dei RE mediante una centralina di controllo centrale

Oltre a quanto descritto precedentemente la versione *design* include inoltre:

- un sensore di qualità dell'aria in grado di misurare i livelli VOC e CO₂
- la circuiteria di controllo per una striscia a LED RGBW di illuminazione ambientale ed il relativo alimentatore da 230V AC a 24V DC
- la circuiteria di controllo per una seconda sonda di temperatura in grado di misurare la temperatura del corpo riscaldante, in maniera tale da migliorare l'accuratezza degli algoritmi di termoregolazione

Entrambe le versioni di scheda elettronica eseguono la stessa versione del firmware, che ha una fase iniziale in cui identifica la tipologia di scheda e le funzionalità opzionali abilitate leggendole da un file di configurazione caricato nel dispositivo in fase di collaudo, e di conseguenza configura le periferiche del microcontrollore.

Tuttavia per essere esaustivi è necessario svolgere i test di accettazione su entrambe le versioni di scheda elettronica, in quanto possono esservi comportamenti differenti.

3.1.2 Modulo Telit

Come cuore del sistema abbiamo il GS2200M. Questo microcontrollore inizialmente protetto da Gainspan, successivamente acquisita da Telit, presenta le seguenti caratteristiche hardware:

- processore dual core ARM Cortex M3 fino a 120Mhz, di cui un core dedicato alla gestione del Wi-Fi ed uno all'esecuzione dell'applicativo
- 1Mb di RAM in totale, di cui all'incirca 500kB utilizzabile dall'applicazione, il resto dedicata all'uso della parte Wi-Fi
- 4Mb di memoria flash interna, in parte dedicata al codice del firmware ed in parte come filesystem interno in cui memorizzare i dati dell'applicazione
- interfaccia Wi-Fi b/g/n a 2.4Ghz in grado di operare sia in modalità Station (STA) sia che Access Point (AP) a cui connettersi direttamente

- 19 input/output digitali
- 3 output Pulse Width Modulation (PWM)
- 2 ingressi analogici mediante Analog to Digital Converter (ADC), uno a 10 ed uno a 12 bit
- un interfaccia I2c hardware
- un interfaccia Serial Peripheral Interface (SPI) hardware
- due interfacce seriali Universal Asynchronous Receive and Transmit (UART)
- modulo Real Time Clock (RTC) interno

Il microcontrollore è dotato di due core ARM, di cui uno è deputato alla gestione della parte Wi-Fi ed esegue un firmware scritto dal produttore stesso. I due microcontrollori comunicano attraverso una memoria condivisa (dual-port) da 64Mb. Il core principale si occupa invece di gestire la parte applicativa.

La toolchain fornita dal produttore per lo sviluppo del firmware in grado di girare sul core applicativo si basa sull'ambiente di sviluppo proprietario IAR.

Viene fornito un Software Development Kit (SDK) che offre:

- un RTOS basato su ThreadX
- uno stack di rete TCP/IP basato su NetX
- implementazione software Wi-Fi sia in modalità Station (STA) che Access Point (AP), con relativi protocolli WEP, WPA/WPA2 sia personal che enterprise
- un filesystem (simile al FAT32) per la gestione della memoria flash integrata
- implementazione Transport Layer Security (TLS)
- implementazione client e server del protocollo HTTP ed HTTPS

- implementazione client del protocollo SNTP
- aggiornamento OTA con doppia partizione
- hardware abstraction layer (HAL) per la gestione di tutti i General Purpose Input Output (GPIO), lettura dei due ADC, del modulo PWM, gestione del RTC, del bus i2c ed SPI

I moduli Wi-Fi di questo tipo possono essere utilizzati secondo due modalità:

- *hosted*: il modulo Wi-Fi viene interfacciato ad un microcontrollore principale che implementa le funzioni del dispositivo. Quest'ultimo si interfaccia con il Telit mediante interfaccia seriale e comunica con i comandi AT (standard per la gestione dei modem)
- *hostless*: il modulo Wi-Fi esegue direttamente l'applicativo senza che vi sia un microcontrollore principale. Tutte le funzionalità del dispositivo sono impelementate sul modulo Telit

Tipicamente viene seguito il primo approccio, tuttavia noi abbiamo optato per il secondo. Questo comporta svariate semplificazioni, quali il non dover gestire la distribuzione e l'aggiornamento di due firmware, e l'avere quindi un prodotto più solido.

Un secondo microcontrollore comporta inoltre problemi di approvvigionamento, soprattutto di questi ultimi periodi in cui il mercato dei componenti elettronici è impazzito, con articoli che un tempo costavano pochi dollari arrivati a costare svariate decine.

La scelta di seguire l'approccio *hostless* non è stata senza difficoltà, dovute principalmente alla scarsa documentazione fornita dal produttore Telit. Infatti il produttore ha deciso di seguire un approccio completamente closed-source, dove tutta la documentazione ed il codice di esempio non è pubblico ma dato solo sotto NDA.

Questo comporta che l'unico modo per risolvere i problemi sia quello di rivolgersi al supporto ufficiale, non trovando nulla riguardo questa particolare piattaforma hardware con una semplice ricerca su Google. Questo ha indubbiamente reso molto più difficoltoso lo sviluppo.

3.2 Caratteristiche software

Il firmware è suddiviso in moduli (task):

- *core*, si occupa della gestione della macchina a stati principale del dispositivo e del coordinamento di tutti gli altri task. Inizializza tutte le periferiche hardware e la rete, oltre che gestire gli eventi notevoli che arrivano dagli altri task
- *termostato*, task che si occupa di tutto quel che è necessario per regolare la temperatura ambiente secondo le modalità di funzionamento del dispositivo, della lettura delle sonde ambiente/VOC, nonché della gestione dell'illuminazione LED nel caso sia presente
- *MQTT*, si occupa di mantenere la connessione MQTT verso MQTT IoT Core, sincronizzando lo stato interno del dispositivo con il server “IRSAP NOW” mediante protocollo MQTT
- *hmi*, si occupa di rispondere agli input dell'utente sulla pulsantiera e di darne relativo feedback mediante l'uso dei LED RGB e del buzzer
- *HTTP*, si occupa di fornire mediante webserver HTTP un'API REST con la quale è possibile interagire direttamente con il dispositivo. È utilizzata per la fase di provisioning.
- *NCM*, si occupa di gestire la connessione di rete Wi-Fi. Questo task è fornito dal produttore.

3.2.1 Stati interni del dispositivo

Il dispositivo ad alto livello può trovarsi in 3 stati distinti:

- *non abbinato*: in attesa di un primo abbinamento da app. Ogni funzione del dispositivo è esclusa finché l'utente non lo collega mediante l'applicazione
- *disconnesso*: il dispositivo è stato in passato abbinato ma al momento non è connesso al cloud, perché ad esempio la connessione Wi-Fi non è momentaneamente disponibile

- *connesso*: il dispositivo è connesso e sincronizzato con il cloud

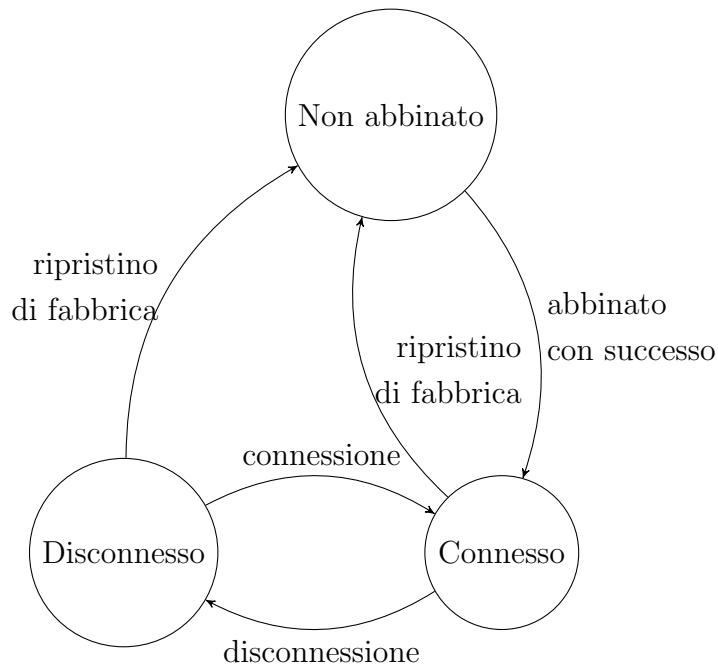


Figura 3.1: Stati del radiatore

Ad ogni stato del dispositivo corrisponde l'attivazione/disattivazione di uno o più componenti software del dispositivo:

stato	modo Wi-Fi	moduli attivi
<i>non abbinato</i>	AP	HMI, HTTP
<i>disconnesso</i>	client	HMI, termostato
<i>connesso</i>	client	HMI, termostato, MQTT

3.2.2 Termoregolazione

La componente di termoregolazione si occupa di regolare la temperatura ambiente portandola il più vicino possibile a quanto desiderato dall'utente (set-point) mediante il controllo dell'accensione (on/off) dell'elemento riscaldante. Il feedback sulla temperatura ambiente è ottenuto mediante la sonda di temperatura NTC.

Il dispositivo ha diversi modi di funzionamento:

- standby: dispositivo completamente spento, sia per quanto riguarda il riscaldamento che per l'illuminazione LED
- antigelo: il dispositivo mantiene una temperatura di sicurezza (imposta dall'utente) per prevenire danni dati da una temperatura ambiente troppo bassa (ad es. congelamento delle tubature)
- vacanza: all'interno di un intervallo temporale impostato dall'utente funziona in modalità antigelo
- away: imposta un set-point ridotto (ECO) in quanto l'utente non è in casa
- programmato: segue una programmazione settimanale che consente per ogni giorno della settimana di creare fino ad 8 fasce orarie
- manuale temporaneo: segue un set-point manuale per un determinato tempo
- manuale: segue il set-point utente che è fisso e non varia mai configurato dall'utente, quindi torna a funzionare nella modalità precedente
- fil pilote: il dispositivo è controllato (ove disponibile) da un segnale esterno in ingresso sul cavo fil pilote

È possibile mediante interfaccia utente muoversi fra le varie modalità come dettagliato in Figura 3.2.

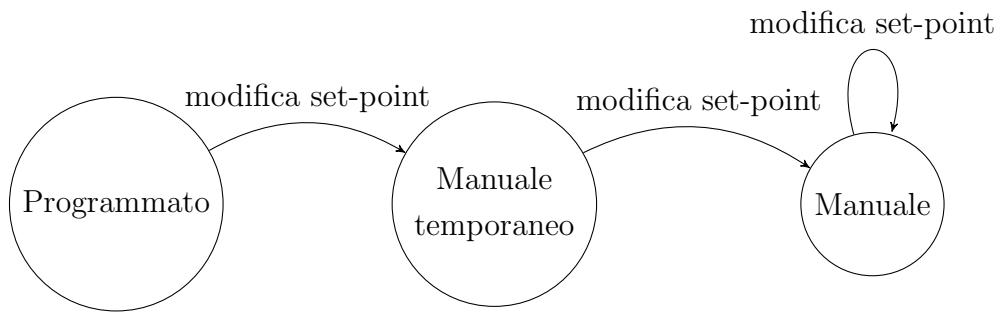


Figura 3.2: Modi del radiatore

3.2.3 Comunicazione cloud

La componente cloud è implementata su MQTT. La connessione avviene grazie al protocollo MQTT usando il servizio broker di MQTT, IoT Core.

Autenticazione

La connessione è cifrata mediante TLS 1.2 ed autenticata mediante certificato del client.

Tale certificato è generato per ogni singolo dispositivo durante le fasi di produzione, e viene firmato da una CA intermedia del produttore hardware, come nel seguente schema:

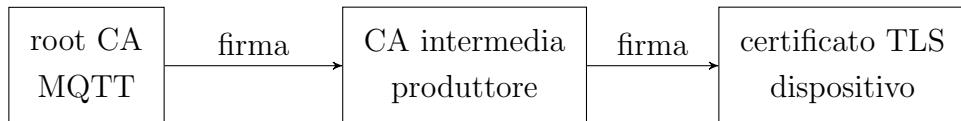


Figura 3.3: Catena TLS

La CA authority intermedia è generata e firmata dalla CA generale di MQTT IoT Core. In questo modo è possibile generare in fase di programmazione offline un certificato per ogni dispositivo che deve essere prodotto. Alla prima connessione del dispositivo al cloud questo automaticamente si registrerà all'interno di IoT Core senza ulteriori necessità di interventi manuali.

IoT Core permette di associare ad ogni certificato una policy, che può andare a garantire al client dei permessi per quanto riguarda MQTT:

- *connect*: autorizza il client a connettersi ad IoT Core utilizzando un particolare *clientId*
- *publish*: consente al client di pubblicare su determinati topic MQTT
- *subscribe*: consente al dispositivo di effettuare una subscription su determinati topic MQTT al fine di ricevere aggiornamenti in tempo reale dal cloud

La policy predefinita restringe i permessi ai soli topic MQTT dedicati per il client con il determinato certificato che si connette al cloud. Per fare questo all'interno del topic MQTT viene inserito il seriale dello specifico dispositivo, che è lo stesso contenuto all'interno del certificato.

Protocollo stateless di comunicazione

Per quanto concerne il protocollo di comunicazione in origine abbiamo valutato inizialmente l'uso del protocollo Device Shadowing¹ supportato nativamente da MQTT IoT Core.

Tuttavia, tale protocollo aveva delle limitazioni che non ne consentivano l'utilizzo nella nostra applicazione, in particolare:

- il pacchetto viene codificato in JSON, il che presenta un overhead di memoria e CPU notevole per il microcontrollore scelto. Inoltre la codifica JSON può introdurre dei bug di encoding
- vi è un hard-limit di 8Kb di dimensione massima di un documento di stato (shadow). Questo, seppur poteva essere sufficiente nelle prime versioni del prodotto, andava a limitare possibilità di espansione futura del prodotto
- il protocollo di comunicazione trasferisce dei delta, che sebbene riducono la dimensione di un pacchetto di dati rendono più complessa la sincronizzazione degli stati del sistema

Per tutte queste ragioni abbiamo scelto di adottare un protocollo binario proprietario, tramite il quale andiamo a trasferire stati completi del dispositivo.

Abbiamo deciso di mantenere comunque i concetti di alto livello dati dal protocollo MQTT Device Shadowing, in particolare la nomenclatura *shadow* per indicare uno stato del dispositivo, che è suddiviso in:

- *state desired* come lo stato in cui si vuole portare il dispositivo, ovvero le impostazioni che l'utente può modificare agendo dalla app, quali ad

¹https://docs.aws.amazon.com/it_it/iot/latest/developerguide/iot-device-shadows.html

esempio la modalità di funzionamento, la programmazione oraria, la configurazione dei LED RGB, etc.

- *state reported*: lo stato attuale del dispositivo. È un superset dello stato desired, in quanto oltre a tutti i campi di quest'ultimo include anche tutti quei valori in sola lettura (ovvero che solo il dispositivo può modificare), ovvero i parametri statistici e di monitoraggio quali la temperatura ambiente, il livello di qualità dell'aria (VOC), gli allarmi del dispositivo, la qualità della connessione Wi-Fi, etc.

Il protocollo è quindi stateless, e consente di effettuare 4 messaggi, più relative risposte, (come visibile in Figura 3.4):

- *get*: disponibile fra dispositivo e cloud, consente la richiesta dello stato *desired* corrente
- *reported-update*: disponibile fra dispositivo e cloud, trasmette lo stato completo del dispositivo al server
- *delete*: eliminazione dello stato corrente presente su cloud
- *desired-update*: unico messaggio inviato dal cloud al dispositivo, trasmette lo stato completo *desired*

Il tipo di messaggio dipende dal topic MQTT sul quale i pacchetti sono pubblicati. Ad un messaggio pubblicato dal dispositivo verso il server il server risponde in base allo stato della richiesta sullo stesso topic MQTT con aggiunto un suffisso:

- */accepted*: la richiesta è stata accettata dal server
- */rejected*: la richiesta è stata respinta dal server in quanto è avvenuto un errore

Il client può identificare a quale richiesta fa riferimento ad una risposta attraverso un token (*clientToken*) che il client setta su ogni richiesta inviata e che il server aggiunge ad ogni risposta che invia al client.

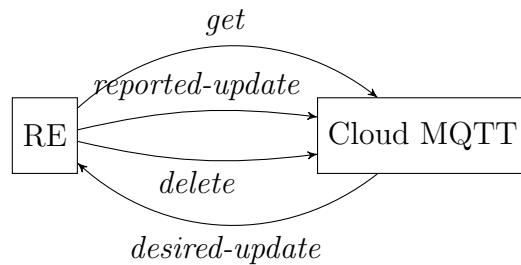


Figura 3.4: Schema di comunicazione dispositivo/cloud

Formato messaggi cloud

Ogni messaggio su cloud include un header fisso, che comprende i seguenti campi

chiave	tipo	descrizione
timestamp	u32	timestamp di invio del messaggio
clientToken	u32	un ID della richiesta che il client aggiunge ad ogni messaggio per poter correlare le risposte alle richieste effettuate
version	u32	versione del messaggio
length	u16	lunghezza totale del messaggio (header incluso)
type	u8	tipo di messaggio, identifica il payload presente

A seguire nel messaggio è presente il payload:

chiave	tipo	descrizione
connected	u8	1 se il dispositivo è online, altrimenti 0
firmwareVersion	u8[2]	versione firmware (major, minor)
hardwareVersion	u8	versione hardware
macAddress	u8[6]	MAC address (seriale del dispositivo)

systemStatus	u8	stato del Sistema (bitmask)
filPiloteStatus	u8	stato ingress Fil Pilote
alarm	u8	allarmi (bitmask)
heatingStatus	u8	stato riscaldamento (bitmask)
rssi	u8	Potenza Wi-Fi
noise	u8	Rumore Wi-Fi
currentSetPoint	i16	set point corrente (in decimi di grado)
currentSetPointEnd	u32	scadenza set-point corrente
vocValue	u16	valore VOC (qualità dell'aria)
co2Value	u16	valore CO2 (qualità dell'aria)
temperature	i16	temperatura ambiente (in decimi di grado)
setPointOff	u16	set point di OFF (espresso in decimi di grado)
setPointEco	u8	set point di ECO (espresso in decimi di grado - 10C)
manualSetPoint	u8	set point manual (espresso in decimi di grado -10C)
temporaryManualSetPoint	u8	set point manual temporaneo (in decimi di grado)
temporaryManualEnd	u32	fine manuale temporaneo (timestamp)
hysteresis	u8	isteresi (espressa in decimi di grado)
temperatureSensorOffset	i8	correzione sensore di temperatura (espresso in decimi di grado)
loadTemperature	i16	temperatura carico (in decimi di grado)
loadOnSeconds	u32	cumulativo in secondi di accensione del carico
holidayStart	u32	timestamp inizio vacanza
holidayEnd	u32	timestamp fine vacanza
metricInterval	u8	intervallo invio metriche periodiche (minuti)
systemId	u8[16]	ID impianto (UUID)
timezone	i16	offset timezone rispetto ad orario UTC
systemConfiguration	u8	configurazione di sistema (bitmask)
ipAddress	u8[4]	indirizzo IP Wi-Fi

openWindowOffTime	u8	tempo di spegnimento in caso rilevamento finestra aperta
ledManualSetPoint	u32	colore LED ambiente (HSV)
schedule	u8[196]	programmazione riscaldamento
ledSchedule	u8[196]	programmazione LED (stesso formato di prima)
ledEnabled	u8	LED ambiente on/off
ledMode	u8	modalità led ambiente (manuale/programmato)
ledColor	u32[10]	preset colori LED ambiente per uso in fasce orarie
temporaryManLedSP	u32	set point LED manuale a tempo
temporaryManLedSPEnd	u32	fine manuale a tempo LED
estimatedTemperature	i16	temperatura ambiente (scritta da cloud per uso con altri sensori)
externalTemperature	i16	umidità esterna (scritta da cloud)
estimatedHumidity	u8	umidità ambiente (scritta da cloud)
externalHumidity	u8	umidità esterna (scritta da cloud)
currentLedSetPoint	u32	set point LED corrente
currentLedSetPointEnd	u32	fine fascia oraria LED corrente
cartridgePowerWatts	u16	potenza della cartuccia
cumuConsumption	u32	consumo cumulato in tutta la vita del radiatore
cumuConsumptionValue	u32	valore all'ultimo snapshot di consumo
cumuConsumptionTime	u32	consumo all'ultimo snapshot
modelName	char[32]	nome modello del radiatore

3.2.4 API di configurazione locale

Quando il dispositivo viene acceso per la prima volta è necessario fornirgli la configurazione della rete Wi-Fi e l'identificativo (UUID) dell'impianto al

quale collegarsi prima che esso possa iniziare a funzionare.

Per fare ciò il dispositivo mette a disposizione un'API REST attraverso la quale la app “IRsap NOW” comunica attraverso una connessione Wi-Fi diretta (in questa fase il dispositivo imposta la propria interfaccia Wi-Fi in modo AP).

Mette a disposizione le seguenti API REST:

endpoint	metodo	descrizione
/irsap/state	GET	ottiene lo stato corrente del dispositivo
/irsap/wifi/scan	GET	ottiene l’elenco di reti Wi-Fi visibili dal dispositivo
/irsap/provision	POST	invia al dispositivo la configurazione
/irsap/test	POST	attiva la modalità collaudo del dispositivo
/gainspan/system/fwup	POST	invia un aggiornamento firmware al dispositivo

3.2.5 HMI

L’interfaccia utente del dispositivo si compone di due pulsanti, + e -, i LED RGB di illuminazione della pulsantiera ed il buzzer.

Tramite l’HMI è possibile effettuare le seguenti operazioni:

- pressione breve +: incrementa il set-point corrente
- pressione breve -: decrementa set-point corrente
- pressione per più di 3 secondi (ma meno di 5) del tasto -: attiva modalità *antigel*
- pressione per più di 5 secondi del tasto -: attivazione modalità *stand-by*
- pressione prolungata dei tasti + e - per più di 5 secondi: ripristino impostazioni di fabbrica

I LED invece sono utilizzati per segnalare la temperatura impostata dall'utente (in base al set-point passano da un colore più freddo ad uno più caldo), sia ad indicare condizioni particolari quali radiatore non abbinato (LED rossi), connessione in corso (viola lampeggiante), modo *stand-by* (viola fisso) o *antigelo* (bianco).

3.3 Intefacciamento con il sistema di test

Ora che abbiamo visto a grande linee come è strutturato il dispositivo sia dal punto di vista hardware che software, possiamo affermare che il sofware presente al suo interno può interagire con il mondo esterno essenzialmente in tre modi:

1. comunicazione cloud: invio e ricezione di stati complessi (*shadow*) mediante il collegamento MQTT al cloud MQTT
2. comunicazione locale: invio e ricezione di comandi mediante l'API REST locale
3. fenomeni fisici: calore emesso, pulsanti che vengono premuti, luce e suoni emessi, temperatura ambiente che varia

Riguardo ai primi due punti è facile pensare ad un sistema per automatizzare e programmare le interazioni, passando da interfacce digitali. Il problema è quindi l'interazione fisica con il dispositivo, che passa per fenomeni analogici.

Possiamo pensare a tre modi:

- interazione con il dispositivo fisico
- interazione con la scheda elettronica
- esecuzione del firmware in un emulatore

3.3.1 Interazione con il dispositivo fisico

Una prima possibilità che si pensa potrebbe essere quella di piazzare il radiatore in una camera climatica e mediante sensori ed attuatori interagire sul dispositivo stesso come farebbe un essere umano.

Questo garantisce di testare uno scenario del tutto identico a quel a cui sarebbe difronte l'utente che si installa il termosifone in casa, tuttavia presenta alcune problematiche che rendono questa strada non percorribile.

Prima di tutto necessita di strumentazione molto specifica e costosa (le camere climatiche), che sebbene a disposizione di IRSAP sono occupate per altri scopi, come lo sviluppo e la validazione degli algoritmi di termoregolazione.

In secondo luogo i test dovrebbero necessariamente seguire le tempistiche dettate dai processi fisici che vengono goveernati, come il riscaldamento di una stanza. Questo rende il test molto lungo in termini temporali, anche svariate ore per eseguire ogni singolo passaggio, considerando che l'ambiente andrebbe ripristinato a pari condizioni iniziali prima di poter eseguire uno nuovo.

3.3.2 Interazione con la sola elettronica

Viene da pensare che si possa quindi eliminare il resto del sistema (il radiatore stesso) e concentrarsi quindi sul test della sola scheda elettronica, andando a simulare gli ingressi e le uscite con cui la scheda comunica con le periferiche a cui è solitamente collegata. Questa è una possibilità che viene effettivamente utilizzata in fase di collaudo dell'elettronica, quando è necessario assicurarsi ad esempio che non vi siano stati problemi di produzione quali saldature fredde.

Tuttavia presenta una problematica la scheda del radiatore è alimentata con la tensione di rete a 230V, il che rende necessario dover isolare la scheda rispetto al dispositivo con cui la si interfaccia per ragioni sia di sicurezza elettrica sia di possibili danni che possono essere arrecati al dispositivo che lavora a bassa tensione.

In effetti possiamo pensare che non ci serva interfacciare la scheda elettronica vera e propria del radiatore: dopotutto a noi interessa testare la componente software, che gira sul microcontrollore presente sulla scheda,

assumendo che l'elettronica è funzionante come da specifica (in quanto già collaudata in fase di produzione).

Possiamo quindi isolare solo la parte che ci interessa, ovvero il chip Telit, utilizzando un kit di sviluppo fornito direttamente dal produttore. Questo permette di connettere la nostra interfaccia di test con tutti gli input/output digitali (GPIO) della scheda, così che potremo andare a simulare tutte le periferiche hardware più agevolmente via software.

Questa è la soluzione che alla fine ho scelto.

3.3.3 Esecuzione del firmware in un emulatore

Infine un'ultima possibilità è quella di eseguire il firmware non sull'hardware reale ma su un sistema emulato.

Questo presenta alcune problematiche, in particolare essendo la piattaforma Telit proprietaria è difficile carpirne tutte le specifiche di funzionamento per andare ad emularne in maniera quanto più simile possibile il funzionamento.

Si potrebbe quindi decidere di compilare un binario x86, andando a sostituire tutte le funzioni utilizzate del framework Telit con stub implementati ex-novo. Questo sebbene potrebbe funzionare (anche se il suo sviluppo sarebbe molto lungo e dispensioso) avrebbe come effetto che non si sta effettivamente testando il binario che poi andrà in produzione, compresa l'integrazione con l'SDK e le caratteristiche fisiche dell'hardware.

Ho quindi deciso di escludere questa strada, che invece è percorribile su altre piattaforme hardware quali l'ESP-32, in quanto utilizza un SDK completamente open-source, e mette già a disposizione la possibilità di emulare un hardware con il software *qemu*.

Implementazione

In questo capitolo vediamo come il sistema di test descritto sopra è stato implementato.

4.1 Interfacciamento hardware

Come già detto in precedenza ho scelto di interfacciarmi direttamente con un kit di sviluppo del microcontrollore scelto, come si può vedere in Figura 4.1.

Il kit di sviluppo mette a disposizione tutti i pin connessi al microcontrollore su comodi pin header. Integra inoltre un convertitore UART (seriale) - USB per permettere di programmarlo e per interagire con la eventuale console di debug (che il radiatore mette a disposizione).

Tutti gli I/O lavorano nel dominio dei 3.3v, quindi ho scelto di utilizzare un Raspberry Pi come interfaccia hardware, connettendo direttamente i suoi GPIO con quelli del kit di sviluppo.

Questo ha un'unica limitazione, che è quella che il Raspberry Pi non mette a disposizione uscite analogiche (DAC), che sarebbero utili per simulare la lettura del sensore di temperatura. Tuttavia con un uscita PWM ed un opportuno filtro analogico passivo è possibile comunque simularne il valore, fosse necessario. Nel mio caso ho scelto di non realizzare questo circuito, connettendo una resistenza fissa al sensore di temperatura del radiatore, in quanto non rientra negli scopi di questi test la lettura di valori differenti del

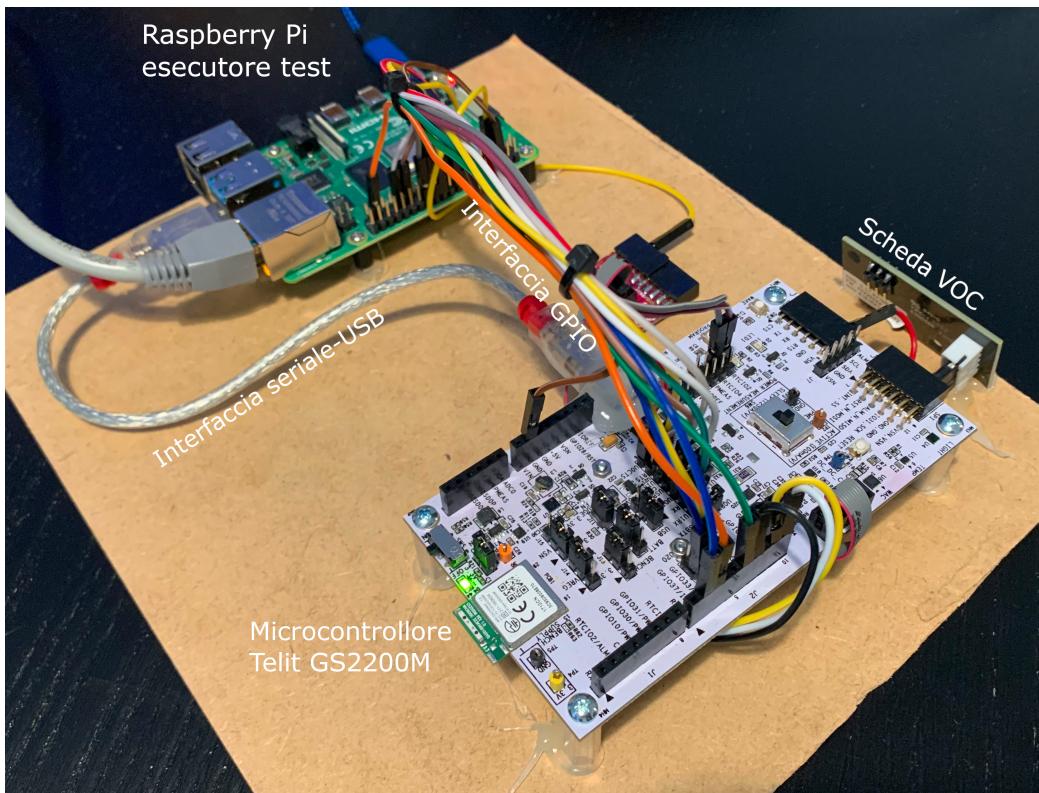


Figura 4.1: Hardware di test realizzato

sensore di temperatura (che dipende da caratteristiche fisiche del sensore più che dal software).

Il raspberry dispone inoltre di un'interfaccia Wi-Fi a 2.4Ghz che consente quindi di avere tutto quanto server per il testing del dispositivo in un'unico comodo dispositivo.

4.2 Motore di esecuzione test

Per la componente software ho deciso di utilizzare il linguaggio di programmazione Python. Questa scelta è motivata sia dalla versatilità che offre nell'interazione anche di basso livello con funzionalità del sistema operativo e periferiche, sia dal fatto che è il linguaggio che solitamente viene utilizzato in azienda per la scrittura di tools e script.

Il codice è realizzato come una libreria python, `fw_test`, sottoforma di un pacchetto installabile con `pip`. In questa maniera l’interfacciamento verso l’hardware e la fixture di test viene separata dal codice effettivo dei test, che può potenzialmente risidere anche su un repository separato.

La libreria mette a disposizione un oggetto principale `Context` che, una volta instanziato con un suo file di configurazione, mette a disposizione una serie di interfacce che consentono l’interazione con i vari moduli visti pocanzi:

- `io`: è responsabile a gestire tutte le periferiche di I/O verso la scheda, ossia GPIO e UART di debug. Mette a disposizione dei metodi di alto livello per interagire con la scheda e verificare lo stato del dispositivo, quali ad esempio `get_status_led_color()` che ottiene il colore del LED di stato, oppure `press_plus()` che invia la pressione del stato +.
- `cloud`: è il modulo responsabile all’interazione con il cloud. Si occupa di mantenere attiva la connessione MQTT con il broker MQTT IoT Core, gestire la codifica/decodifica di messaggi da oggetti a binario e vice versa, di interfacciarsi con la gestione dei Job IoT Core per la distribuzione di aggiornamenti OTA.
- `wifi`: è il modulo deputato al controllo dell’interfaccia Wi-Fi della raspberry. Consente di cambiare la modalità fra AP e STA, supportando l’uso di configurazioni differenti di access point.
- `api`: implementa l’API REST di comunicazione locale con il dispositivo, che come detto è usata dalla app in fase di abbinamento del dispositivo

Come motore di esecuzione dei test ho deciso di utilizzare `pytest`, che è lo standard de factor in ambiente Python. Definendo un file `conftest.py` sono andato a meglio adattare il sistema di test per integrarlo con la libreria sviluppata sopra. In particolare il motore di test:

- si occupa di instanziare la libreria `fw_test` prima dell’avvio dei test, ed a terminarla una volta terminati tutti i test

- prima di ogni test, si preoccupa di portare il sistema in uno stato normale, ovvero il dispositivo deve avere in esecuzione la versione firmware sotto test e deve essere resettato
- raccoglie i log dai vari moduli, inclusa la libreria `fw_test` che usa il modulo logging standard di Python. In questo modo quando un test fallisce si può risalire alle operazioni che ha fatto il Raspberry per causare il fallimento, in modo da poter più agevolmente riprodurre il problema

4.2.1 Interfacciamento con l'I/O

Per l'interfacciamento con l'I/O ho deciso di utilizzare la libreria di riferimento per comunicare con i GPIO del Raspberry Pi.

In una prima fase vengono inizializzati tutti i pin in base alla loro funzione, input o output, e viene aperta l'interfaccia seriale UART con la quale si può comunicare con il kit di sviluppo.

I messaggi di debug del radiatore vengono quindi catturati e convogliati sul logger di sistema, di modo che poi siano raccolti dal sistema CI (continuous integration) e raccolti per scopi di debugging di eventuali test falliti.

Vengono inoltre offerti dei metodi di alto livello per interagire con le varie periferiche del dispositivo, ad esempio ottenere il colore dei led, premere un pulsante sulla pulsantiera, riavviare il dispositivo, etc. Inoltre sono creati dei wrapper per i comandi seriali più comuni da mandare al radiatore.

4.2.2 Interfacciamento con il cloud

Come visto in precedenza il dispositivo si collega al cloud MQTT IoT Core mediante autenticazione con certificato client. Essendo il meccanismo di autenticazione con il server oggetto stesso del test, ed essendo che il broker MQTT IoT Core non è installabile su un normale computer essendo una componente proprietaria, è impossibile testare il dispositivo facendolo collegare ad un MQTT broker differente (come può essere mosquitto).

Inoltre IoT Core mette a disposizione alcune funzionalità che non sono disponibili in altri broker. In particolare la funzionalità dei job, ossia dei

processi batch che è possibile schedulare e fare eseguire ai dispositivi, che consentono di distribuire l'esecuzione di operazioni. Noi utilizziamo principalmente i job per la distribuzione degli aggiornamenti OTA, ma anche per altre operazioni di servizio (come l'aggiornamento di alcuni dati statistici).

Per questo motivo la cosa più semplice è utilizzare IoT Core stesso, ma isolare il collegamento fra il broker ed il resto del sistema per il dispositivo sotto test. Il che si traduce nel fatto che le regole che azionerebbero il resto del sistema ad un evento di pubblicazione sui topic MQTT usati dal cloud sono disabilitate per il seriale del dispositivo in test.

Il raspberry di test si interfaccia quindi con IoT Core come client MQTT che può andare a pubblicare/sottoscriversi ai topic MQTT su cui il dispositivo comunica, come vediamo in Figura 4.2. Dato che il broker non ha un ruolo attivo nel sistema (si occupa solo di inoltrare i messaggi ma non applica alcuna logica di protocollo) la sua presenza all'interno dello scenario di test non è problematica, in quanto viene trattato come un qualsiasi dispositivo di infrastruttura (come ad esempio può essere un router su internet).

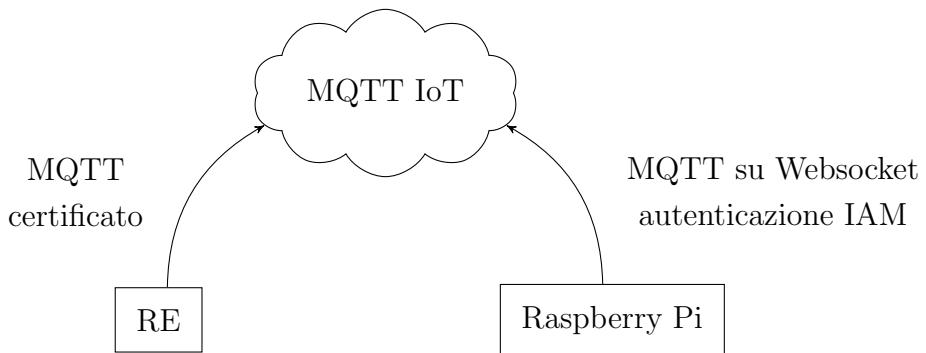


Figura 4.2: interfacciamento fra raspberry e cloud

A livello implementativo uso per collegare il software ad IoT Core la libreria ufficiale di AWS, *awsiot sdk*. Questa libreria, a differenza dei normali client MQTT, consente infatti di autenticarsi grazie a credenziali IAM, non richiedendo quindi la generazione di certificati per il tool di test.

Il modulo cloud del sistema di test, oltre a gestire a basso livello la connessione al server MQTT, aggiunge un layer di astrazione che automatizza la

conversione fra binario e oggetti Python, e vice versa, dei messaggi secondo il protocollo custom implementato.

È fornito un metodo `publish(msg)` per inviare un messaggio, che viene codificato ed inviato sul topic MQTT opportuno (scelto in base al tipo di messaggio secondo la specifica del protocollo). Ad ogni messaggio ricevuto sui topic MQTT a cui il dispositivo fa la subscription viene decodificato e salvato in una coda. Viene fornito un metodo `receive(timeout)` per ricevere un messaggio, ossia scodare l'ultimo messaggio dalla coda, o attendere fino ad un dato timeout che arrivi un messaggio dal dispositivo.

Per quanto converne la gestione degli MQTT IoT Core Jobs, che come detto prima vengono utilizzati per distribuire sui dispositivi operazioni batch da eseguire, viene utilizzata la libreria ufficiale Python di MQTT `boto3`. Vengono fornite 3 interfacce:

- `job_create(document)`: che crea un job con il documento JSON specificato
- `job_state(job_id)` che consente di verificare lo stato di un job con determinato id
- `job_delete(job_id)` che consente di cancellare un job avviato (tipicamente questo viene effettuato alla fine del test per cancellare eventuali job pendenti che erano stati creati, evitando che vengano eseguiti alla prossima iterazione dei test).

4.2.3 Interfacciamento con il Wi-Fi

Il Wi-Fi del Raspberry deve asservire a due compiti:

- collegarsi all'access point del radiatore per poter utilizzare l'API REST locale di configurazione
- consentire al radiatore di collegarsi al cloud mediante Wi-Fi client

Pertanto, vengono supportate due modalità di funzionamento, che non possono essere attive entrambe:

- STA, in questo caso il Raspberry si collega all’interfaccia access-point del RE
- AP, in questo caso il Raspberry agisce come AP software a cui il dispositivo RE si può connettere per raggiungere il cloud.

Il vincolo che non possano essere attive entrambe è una semplificazione che viene ma che non riduce generalità, in quanto il radiatore in un momento può trovarsi a sua volta o in modalità *ap* o in modalità *client*. Il Raspberry dovrà quindi in maniera speculare andare ad attivare l’altra modalità. Faccendo fede sul fatto che il Raspberry è più veloce a cambiare modalità del radiatore (cosa vera in quanto il dispositivo per poter cambiare modalità deve essere riavviato) la limitazione non toglie possibilità di test.

Per la modalità client è sufficiente configurare l’interfaccia di rete per connettersi al radiatore. La rete del radiatore ha un SSID ben definito, che viene caricato dal file di configurazione del programma, e non ha alcuna autenticazione. Per questioni di robustezza viene anche impostato sul client un indirizzo IPv4 statico, in modo da non dover eseguire un client DHCP.

A questo punto è possibile fare richieste locali al webserver del radiatore mediante un qualsiasi client HTTP. In questo caso ho scelto di utilizzare la popolare libreria *requests*.

Ben più complesso invece è l’esposizione di un access point software da parte del Raspberry. Per questo caso ho voluto tenere il requisito di supportare differenti possibilità di configurazione, in maniera tale da poter variare i seguenti parametri:

- SSID
- tipo di sicurezza della rete (nessuna, WEP, WPA, WPA2)
- passphrase della rete
- canale Wi-Fi utilizzato (scelto nell’insieme di canali ammessi dallo standard ETSI)

Per creare l’access point ho quindi scelto di utilizzare il software *hostapd*. L’interfaccia di test si occupa quindi di generare un file di configurazione per

tale servizio, quindi di lanciarlo come processo figlio. I log di questo processo vengono inoltre catturati ed inoltrati al logger, di modo che possano essere anch'essi raccolti dall'esecutore di test per un eventuale debugging di test falliti.

Manca ora l'implementazione del server DHCP: per questo ho deciso di utilizzare *dnsmasq*, in grado di fornire sia un server DHCP che DNS al dispositivo. Anche in questo caso viene avviato come processo figlio ed i log vengono catturati in maniera del tutto analoga a quanto avviene per *hostapd*.

Infine manca il collegamento fra l'interfaccia Wi-Fi e la rete: con delle sepolte regole di *iptables* è possibile creare un NAT andando ad abilitare l'opzione *masquerade* per i pacchetti in uscita, quindi è sufficiente abilitare l'IPv4 forwarding dalle impostazioni del kernel Linux per consentire ad ogni client che si connette al dispositivo di avere accesso internet.

In questa maniera sarà possibile in futuro andare a testare non solo configurazioni di rete più disparate, ma anche indurre malfunzionamenti nella rete, quali ad esempio perdite di pacchetti o eccessive latenze, e valutare se il dispositivo si comporta come da specifica.

Queste sono le casistiche più difficili da testare ed avere la possibilità di automatizzarle può voler dire accorgersi subito di problemi che altrimenti sarebbero difficilissimi se non impossibili da individuare in utenza.

4.3 Integrazione continua

L'ultimo passo è quindi quello di integrare i test automatizzati nel flusso attuale di *continuous integration* (CI) aziendale, in maniera tale da non richiedere un intervento manuale ogni volta che viene prodotta una nuova release.

Per la gestione del versionamento del codice sorgente utilizziamo il sistema di controllo versionamento Git sul servizio *GitHub*. Come politica interna di *branching model* abbiamo scelto di usare il *trunk based development*, che sta sempre prendendo più piede all'interno delle aziende.

Questo prevede che vi sia sempre un unico ramo, nel nostro caso il *master*, sul quale tutte le modifiche vengono integrate, idealmente il più spesso

possibile, in maniera tale da evitare che i vari rami di sviluppo divergano al punto da rendere difficile, se non impossibile, integrarli.

Fondamentale per questa filosofia di sviluppo è però l'uso di strumenti di *continuous integration*, che continuino a compilare e testare ogni nuovo commit sul *master*, in maniera tale da accorgersi il prima possibile di eventuali bug introdotti. Questo assicura che vi sia sempre una versione compilata e funzionante potenzialmente rilasciabile agli utenti.

Utilizzando *Github* viene naturale l'integrazione con il servizio di CI *Github actions*, che permette l'esecuzione di flussi di lavoro (*workflow*), a sua volta suddivisi in *job*, su delle macchine worker, che possono essere sia gestite da *Github* stesso, sia essere installate su dei server gestiti da se.

Nel nostro caso utilizziamo un server di build con al interno container e macchine virtuali sia Linux che Windows, oltre che ad un paio di server macOS per la compilazione di applicazioni iOS.

Oltre a questo utilizziamo un database *CouchDB* per archiviare lo stato di tutti gli artefatti prodotti dai processi di CI e tracciare i loro rilasci nei vari ambienti di esecuzione che noi abbiamo (produzione, staging, test).

Per questo progetto ho creato un unico *workflow* che viene invocato quando viene eseguito un push sul branch *master* del repository contenente il firmware del RE, che contiene a sua volta all'interno due *job*:

- *compilazione*: il primo step è eseguito su una macchina virtuale Windows ed è quello che si occupa di compilare una nuova versione del firmware, e di archiviarla nel bucket contenente tutti gli artefatti prodotti dal processo CI
- *test*: esegue quanto descritto fin ora attraverso un runner installato direttamente sulla raspberry di test

Vediamo in maniera schematica questa integrazione in sezione 4.3.

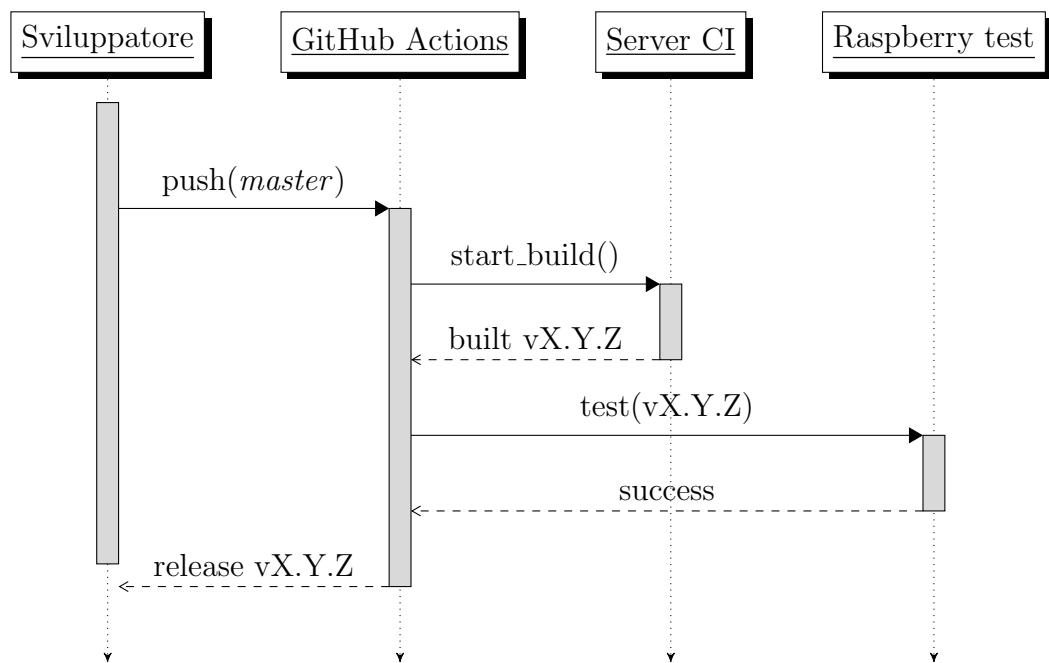


Figura 4.3: infrastruttura CI

Capitolo 5

Validazione sperimentale

5.1 scenari da testare

Per iniziare ho deciso di andare a scrivere un test per ognuno degli scenari attualmente testati manualmente nei test di accettazione di IOTINGA.

Questi corrispondono ai casi d'uso considerati critici in quanto un malfunzionamento di essi pregiudica ogni tipo di funzionalità del dispositivo o non ne permette l'aggiornamento a successive versioni.

5.1.1 Abbinamento del dispositivo ad un impianto

Viene testata la capacità di abbinare mediante app il dispositivo ad un impianto. Il dispositivo deve correttamente abbinarsi all'impianto ed inviare al cloud la versione del firmware corretta.

Caso di test

azione	risultato atteso
prendere un dispositivo in stato <i>non abbinato</i>	
connettersi alla rete Wi-Fi del RE in test	la connessione ha successo. È possibile pingare il radiatore all'indirizzo 192.168.240.1
effettuare una richiesta HTTP GET all'API <i>/irsap/status</i>	viene restituito un JSON contenente alcune informazioni sul dispositivo. Verificare che la versione del firmware indicata sia quella in test
effettuare una richiesta HTTP GET all'API <i>/irsap/wifi/scan</i>	viene restituito un JSON contenente l'elenco delle reti Wi-Fi viste dal radiatore
inviare una richiesta HTTP POST all'API <i>/irsap/provision</i>	entro 5 secondi il dispositivo si riavvia. I led quindi lampeggiano di viola fino che non viene stabilita una connessione correttamente alla rete, a quel punto si spengono e si vede arrivare su cloud un messaggio di <i>update</i> .

Implementazione

```
from time import time, sleep
from logging import getLogger

import uuid

from fw_test.context import Context
from fw_test.wifi import ApConfiguration, WifiSecurityType
from fw_test.cloud import Message, Action, Response, PacketType
from fw_test.firmware import FirmwareVersion
```

```
LOGGER = getLogger(__name__)

TEST_SSID = "TEST-NETWORK"
TEST_PASSPHRASE = "test-network-passphrase"

def test_pairing(ctx: Context):
    # avvio la connessione del Raspberry all'AP
    ctx.wifi.client_connect()

    sleep(1)

    # chiedo lo stato al dispositivo
    response = ctx.api.status()

    # mi assicuro che la versione firmware del dispositivo sia quella da testare
    assert FirmwareVersion.from_str(response["system"]["fwVer"]) == ctx.firmware.ver

    # effettuo la scansione Wi-Fi
    response = ctx.api.wifi_scan()

    # mi assicuro che la scan restituisca almeno una rete
    assert len(response) > 0

    # invio la richiesta provision
    ap_config = ApConfiguration(
        ssid=TEST_SSID,
        passphrase=TEST_PASSPHRASE,
        security_type=WifiSecurityType.WPA2,
        channel=6,
    )
    env_id = uuid.uuid4()
    response = ctx.api.provision(ap_config, env_id)
```

```

assert response["status"] == "success"

# communto la raspberry in AP mode
ctx.wifi.start_ap(ap_config)

# attendo il primo messaggio su cloud
msg = ctx.cloud.receive(timeout=60)
assert msg.action == Action.GET

# invio una GET rejected al dispositivo device
ctx.cloud.publish(Message(
    action=Action.GET,
    response=Response.REJECTED,
    state={
        "clientToken": msg.state["clientToken"],
        "timestamp": int(time()),
        "requestId": 0,
        "type": PacketType.HEADER,
    }
))
# mi aspetto ora un reported update
msg = ctx.cloud.receive(timeout=5)
assert msg.method == Action.REPORTED_UPDATE

# la versione iniziale deve essere zero
assert msg.state["version"] == 0

# il system id deve essere uguale all'env id
assert msg.state["systemId"] == env_id.bytes()

# la versione firmware deve corrispondere a quella in test
assert FirmwareVersion.from_bytes(msg.state["firmwareVersion"]) == ctx.fi

```

5.1.2 Downgrade del firmware mediante API REST locale

Il dispositivo deve essere sempre aggiornabile mediante API REST locale quando questo è in stato *non abbinato*. Garantisce che in caso di problemi la app possa sempre installare sul dispositivo un firmware sicuramente funzionante, nonché il firmware possa essere aggiornato all'ultima versione fabbrica durante la procedura di collaudo.

Caso di test

azione	risultato atteso
prendere un dispositivo appena inizializzato alle impostazioni di fabbrica	il dispositivo si trova in stato <i>non abbinato</i>
inviare al dispositivo una versione del firmware precedente a quella installata	il dispositivo entro 30 secondi si riavvia e presenta nel suo stato la versione firmware inviatagli

Implementazione

```
from time import sleep

from fw_test.context import Context
from fw_test.firmware import FirmwareVersion


def test_downgrade(ctx: Context):
    # connette il Raspberry all'AP del radiatore elettrico
    ctx.wifi.client_connect()

    # invio un aggiornamento firmware locale
    response = ctx.api.firmware_update(ctx.config.prev_firmware_path)
    assert response.status_code == 200

    # attendo che il dispositivo si riavvii
```

```

sleep(2)

# mi ricollego al radiatore
ctx.wifi.client_connect()

response = ctx.wifi.status()

# controllo che la versione firmware sia quella inviata
assert FirmwareVersion.from_str(["system"]["fwVer"]) == ctx.config.prev_f

```

5.1.3 Aggiornamento di un dispositivo mediante OTA MQTT

Un dispositivo abbinato ad una app e connesso ad internet deve poter ricevere gli aggiornamenti OTA mediante Job IoT Core.

Caso di test

azione	risultato atteso
portare un dispositivo in stato <i>connesso</i>	i LED della pulsantiera sono spenti
inviare mediante Job IoT Core la versione firmware precedente	il job passa in stato <i>in corso</i> sulla console di MQTT. Entro un minuto il dispositivo si riavvia con il nuovo firmware ed il job passa in stato <i>successo</i> . Su cloud è inviato un messaggio riportante la versione del firmware inviatagli.

Implementazione

```

from time import sleep
import uuid

from fw_test.context import Context

```

```
from fw_test.io import LedColor, Action
from fw_test.cloud import JobState
from fw_test.wifi import ApConfiguration, WifiSecurityType
from fw_test.firmware import FirmwareVersion

TEST_SSID = "TEST-NETWORK"
TEST_PASSPHRASE = "test-network-passphrase"

def test_ota(ctx: Context):
    # avvio la connessione del Raspberry all'AP
    ctx.wifi.client_connect()

    sleep(1)

    # invio la richiesta provision
    ap_config = ApConfiguration(
        ssid=TEST_SSID,
        passphrase=TEST_PASSPHRASE,
        security_type=WifiSecurityType.WPA2,
        channel=6,
    )
    env_id = uuid.uuid4()
    response = ctx.api.provision(ap_config, env_id)
    assert response["status"] == "success"

    # communto la raspberry in AP mode
    ctx.wifi.start_ap(ap_config)

    # attendo che il pairing abbia successo
    sleep(10)

    assert ctx.io.status_led_color == LedColor.OFF

    job_id = ctx.cloud.send_ota(ctx.config.prev_firmware_path)
```

```

sleep(60)

# il job ha avuto successo
assert ctx.cloud.job_state(job_id) == JobState.SUCCEEDED

# su cloud arriva la nuova versione
# mi aspetto ora un reported update
msg = ctx.cloud.receive(timeout=5)
assert msg.method == Action.REPORTED_UPDATE

# la versione firmware deve corrispondere a quella in test
assert FirmwareVersion.from_bytes(msg.state["firmwareVersion"]) == ctx.co

```

5.1.4 Ripristino di fabbrica di un dispositivo connesso

Deve essere sempre garantito che un dispositivo possa essere riportato alle impostazioni di fabbrica.

Caso di test

azione	risultato atteso
portare un dispositivo in stato <i>connesso</i>	i LED della pulsantiera sono spenti
tenere premuti i tasti + e - per 5 secondi	nel mentre si premono i tasti la pulsantiera lampeggia di giallo ed emette beep. Infine i LED rimangono accesi colore giallo fisso e la frequenza dei beep aumenta
rilasciare i pulsanti quindi premere il tasto +	entro 5 secondi il colore della tastiera diventa rosso. Su cloud arriva un messaggio <i>DELETE</i>

Implementazione

```
from time import sleep

import uuid

from fw_test.context import Context
from fw_test.io import LedColor
from fw_test.cloud import Action
from fw_test.wifi import ApConfiguration, WifiSecurityType

TEST_SSID = "TEST-NETWORK"
TEST_PASSPHRASE = "test-network-passphrase"

def test_factory_reset(ctx: Context):

    # avvio la connessione del Raspberry all'AP
    ctx.wifi.client_connect()

    sleep(1)

    # invio la richiesta provision
    ap_config = ApConfiguration(
        ssid=TEST_SSID,
        passphrase=TEST_PASSPHRASE,
        security_type=WifiSecurityType.WPA2,
        channel=6,
    )
    env_id = uuid.uuid4()
    response = ctx.api.provision(ap_config, env_id)
    assert response["status"] == "success"
```

```

# communto la raspberry in AP mode
ctx.wifi.start_ap(ap_config)

# attendo che il pairing abbia successo
sleep(10)

assert ctx.io.status_led_color == LedColor.OFF

# ora posso fare l'hard reset
ctx.io.hard_reset()

# attendo che arrivi su cluod una DELETE
msg = ctx.cloud.receive(timeout=5)
assert msg.method == Action.DELETE

sleep(1)

# il LED indica dispositivo resettato
assert ctx.io.status_led_color() == LedColor.RED

```

5.1.5 Ripristino di fabbrica di un dispositivo disconnesso

Deve essere possibile resettare un dispositivo anche quando questo non è connesso alla rete, in quanto l'utente potrebbe volerlo riabbinare da app in quanto ha sostituito o cambiato la configurazione del proprio AP Wi-Fi.

Caso di test

azione	risultato atteso
portare un dispositivo in stato <i>connesso</i>	i LED della pulsantiera sono spenti
spegnere l'AP al quale il dispositivo è connesso	il dispositivo continua a funzionare in modalità <i>disconnesso</i>
tenere premuti i tasti + e - per 5 secondi	nel mentre si premono i tasti la pulsantiera lampeggia di giallo ed emette beep. Infine i LED rimangono accesi colore giallo fisso e la frequenza dei beep aumenta
rilasciare i pulsanti quindi premere il tasto +	entro 5 secondi il colore della tastiera diventa rosso

Implementazione

```
from time import sleep

import uuid

from fw_test.context import Context
from fw_test.io import LedColor
from fw_test.wifi import ApConfiguration, WifiSecurityType


TEST_SSID = "TEST-NETWORK"
TEST_PASSPHRASE = "test-network-passphrase"

def test_factory_reset(ctx: Context):

    # avvio la connessione del Raspberry all'AP
    ctx.wifi.client_connect()
```

```
sleep(1)

# invio la richiesta provision
ap_config = ApConfiguration(
    ssid=TEST_SSID,
    passphrase=TEST_PASSPHRASE,
    security_type=WifiSecurityType.WPA2,
    channel=6,
)
env_id = uuid.uuid4()
response = ctx.api.provision(ap_config, env_id)
assert response["status"] == "success"

# communto la raspberry in AP mode
ctx.wifi.start_ap(ap_config)

# attendo che il pairing abbia successo
sleep(10)

assert ctx.io.status_led_color == LedColor.OFF

# ora stoppo l'access-point
ctx.wifi.stop_ap()

# attendo 5 secondi
sleep(5)

# ora posso fare l'hard reset
ctx.io.hard_reset()

sleep(1)

# il LED indica dispositivo resettato
assert ctx.io.status_led_color() == LedColor.RED
```

5.1.6 Termoregolazione di base

Questo test verifica che il radiatore sia in grado di riscaldare l'ambiente seguendo un set-point manuale.

Caso di test

azione	risultato atteso
abbinare un radiatore e portarlo in modo funzionamento <i>manuale</i> con set-point inferiore alla temperatura ambiente	il radiatore non è in riscaldamento
incrementare premendo il tasto + ripetutamente il set-point manuale fino che i LED della pulsantiera diventano rossi	il radiatore inizia a scaldare. Su cloud viene inviato un messaggio indicante lo stato <i>in chiamata</i>
decrementare premendo il tasto - ripetutamente il set-point manuale fino che i LED della pulsantiera non diventano blu	il radiatore smette di scaldare. Su cloud arriva uno stato con il bit <i>in chiamata</i> settato a <i>false</i>

Implementazione

```
from time import sleep, time
import uuid

from fw_test.context import Context
from fw_test.io import LedColor
from fw_test.cloud import Message, Response, Action
from fw_test.wifi import ApConfiguration, WifiSecurityType

from .state import STATE_MANUAL

TEST_SSID = "TEST-NETWORK"
```

```
TEST_PASSPHRASE = "test-network-passphrase"

def test_thermoregulation(ctx: Context):
    # avvio la connessione del Raspberry all'AP
    ctx.wifi.client_connect()

    sleep(1)

    # invio la richiesta provision
    ap_config = ApConfiguration(
        ssid=TEST_SSID,
        passphrase=TEST_PASSPHRASE,
        security_type=WifiSecurityType.WPA2,
        channel=6,
    )
    env_id = uuid.uuid4()
    response = ctx.api.provision(ap_config, env_id)
    assert response["status"] == "success"

    # communto la raspberry in AP mode
    ctx.wifi.start_ap(ap_config)

    # attendo che il pairing abbia successo
    sleep(10)

    assert ctx.io.status_led_color == LedColor.OFF

    ctx.cloud.publish(Message(
        action=Action.GET,
        state={
            **STATE_MANUAL,
            "clientToken": msg.state["clientToken"],
            "timestamp": int(time()),
        }
    ))
```

```
        "systemId": env_id.bytes(),
        "requestId": 1,
    }
))

sleep(2)
msg = ctx.cloud.receive(timeout=5)
assert msg.method == Action.REPORTED_UPDATE

for _ in range(20):
    ctx.io.press_plus()
    sleep(0.5)

sleep(1)

# il radiatore deve scalare ora
assert ctx.io.is_load_active()

msg = ctx.cloud.receive(timeout=5)
assert msg.method == Action.REPORTED_UPDATE
assert msg.status["heatingStatus"] != 0

for _ in range(20):
    ctx.io.press_minus()
    sleep(0.5)

sleep(1)

# il radiatore non deve più scaldare
assert not ctx.io.is_load_active()

msg = ctx.cloud.receive(timeout=5)
assert msg.method == Action.REPORTED_UPDATE
assert msg.status["heatingStatus"] == 0
```

5.1.7 Attivazione modalità stand-by

Scopo di questo test è la verifica che la funzionalità di stand-by funzioni correttamente. È molto importante in quanto una normativa europea impone che ogni prodotto abbia una modalità stand-by imponendo anche un limite massimo di consumo in questa modalità di 1W (e più di recente 0.5W).

Caso di test

azione	risultato atteso
prendere un radiatore dotato di LED RGBW <i>connesso</i> ed in modo <i>manuale</i> che sta chiamando e con i LED accesi	
tenere premuto per almeno 5 secondi il tasto -	il radiatore smette di riscaldare ed i LED RGBW si spengono
toccare la pulsantiera del radiatore	i LED della pulsantiera si illuminano di viola
tenere premuto per almeno 5 secondi il tasto -	il radiatore riprende a riscaldare ed i LED RGBW si riaccendono con lo stesso colore che avevano in precedenza

Implementazione

```
from time import sleep, time
import uuid

from fw_test.context import Context
from fw_test.io import LedColor
from fw_test.cloud import Message, Response, Action
from fw_test.wifi import ApConfiguration, WifiSecurityType

from .state import STATE_MANUAL
```

```
def test_standby(ctx: Context):
    # avvio la connessione del Raspberry all'AP
    ctx.wifi.client_connect()

    sleep(1)

    # invio la richiesta provision
    ap_config = ApConfiguration(
        ssid=TEST_SSID,
        passphrase=TEST_PASSPHRASE,
        security_type=WifiSecurityType.WPA2,
        channel=6,
    )
    env_id = uuid.uuid4()
    response = ctx.api.provision(ap_config, env_id)
    assert response["status"] == "success"

    # communto la raspberry in AP mode
    ctx.wifi.start_ap(ap_config)

    # attendo che il pairing abbia successo
    sleep(10)

    assert ctx.io.status_led_color == LedColor.OFF

    ctx.cloud.publish(Message(
        action=Action.GET,
        state={
            **STATE_MANUAL,
            "clientToken": 0,
            "timestamp": int(time()),
            "systemId": env_id.bytes(),
        }
    ))
```

```
        "requestId": 1,  
        "manualSetPoint": 200,  
    }  
)  
  
sleep(1)  
assert ctx.io.load_active()  
  
ctx.io.press_minus(6)  
  
assert ctx.io.status_led_color() == LedColor.MAGENTA  
assert not ctx.io.load_active()  
  
sleep(1)  
  
ctx.io.press_minus(6)  
  
assert ctx.io.status_led_color() != LedColor.MAGENTA  
assert ctx.io.load_active()
```

5.1.8 Funzionamento disconnesso

Lo scopo di questo test è verificare che in assenza di connessione ad internet il dispositivo continui a funzionare per quanto previsto dalla modalità *degradata*.

Caso di test

azione	risultato atteso
prendere un dispositivo abbinato ad un impianto in modo <i>programmato</i> spento	
spegnere l'access point Wi-Fi configurato nel dispositivo	
accendere il dispositivo	i LED della pulsantiera entro 30 secondi si illuminano di giallo
riconnettere l'access point	entro 30 secondi i LED del dispositivo si spengono. Su cloud arriva una richiesta GET dal dispositivo

Implementazione

```
from time import sleep

import uuid

from fw_test.context import Context
from fw_test.io import LedColor
from fw_test.cloud import Action
from fw_test.wifi import ApConfiguration, WifiSecurityType

TEST_SSID = "TEST-NETWORK"
TEST_PASSPHRASE = "test-network-passphrase"

def offline_working(ctx: Context):
    # avvio la connessione del Raspberry all'AP
    ctx.wifi.client_connect()

    sleep(1)
```

```
# invio la richiesta provision
ap_config = ApConfiguration(
    ssid=TEST_SSID,
    passphrase=TEST_PASSPHRASE,
    security_type=WifiSecurityType.WPA2,
    channel=6,
)
env_id = uuid.uuid4()
response = ctx.api.provision(ap_config, env_id)
assert response["status"] == "success"

# communto la raspberry in AP mode
ctx.wifi.start_ap(ap_config)

# attendo che il pairing abbia successo
sleep(10)

assert ctx.io.status_led_color == LedColor.OFF

# ora stoppo l'access-point
ctx.wifi.stop_ap()

# riavvio il dispositivo
ctx.io.reset()

sleep(30)

# il LED indica dispositivo resettato
assert ctx.io.status_led_color() == LedColor.YELLOW

ctx.wifi.start_ap(ap_config)

sleep(5)
```

```
msg = ctx.cloud.receive(timeout=5)
assert msg.method == Action.REPORTED_UPDATE
```

5.2 Tempi di esecuzione

Vediamo ora la differenza in termini di tempo fra l'esecuzione del test in maniera automatica e l'esecuzione manuale:

test	manuale	automatico
Abbinamento del dispositivo ad un impianto (5.1.1)	??min	??min
Downgrade del firmware mediante API REST locale (5.1.2)	??min	??min
Aggiornamento di un dispositivo mediante OTA MQTT (5.1.3)	??min	??min
Ripristino di fabbrica di un dispositivo connesso (5.1.4)	??min	??min
Ripristino di fabbrica di un dispositivo disconnesso (5.1.5)	??min	??min
Termoregolazione di base (5.1.6)	??min	??min
Attivazione modalità stand-by (5.1.7)	??min	??min
Funzionamento disconnesso (5.1.8)	??min	??min

Conclusioni

Abbiamo visto come l’automatizzazione dei test di accettazione può essere implementata ed utilizzata anche in un progetto embedded andando ad evitare danni (e quindi costi) derivanti dal rilascio di software non adeguatamente o erroneamente testato.

Visti i risultati ottenuti è possibile pensare di espandere il sistema sviluppato anche ad altri progetti sia di IRSAP sia che IOTINGA ha per altri clienti.

Di seguito vedremo alcuni esempi di progetti esistenti e futuri dove questa metodologia potrà essere applicata, e le relative sfide per implementarla.

6.1 Futuri radiatori elettrici

Attualmente è in progettazione una nuova elettronica per il RE, che mira a produrre un’unica piattaforma modulare per tutti i dispositivi radiatori elettrici (e non solo) smart prodotti da IRSAP.

Questa elettronica è composta da tre schede distinte, ognuna delle quali assolve ad una funzione:

- scheda logica: contiene il microcontrollore ESP-32 e tutte le funzioni del dispositivo
- scheda di alimentazione: si occupa di fornire alimentazione al resto dell’elettronica da una fonte appropriata (230V AC)

- scheda di potenza: si occupa di controllare il corpo riscaldante

Di queste tre (eventualmente le ultime due potranno essere combinate in una unica scheda) la prima è identica per tutti i prodotti, le altre invece cambiano da un prodotto all'altro, anche per adattarlo alle esigenze di mercati differenti (come ad esempio il mercato americano dove la tensione di rete è 115V).

Per la scheda logica abbiamo deciso di scegliere un modulo Wi-Fi Espressif ESP-32, che non solo è più economico rispetto al Telit ma ha delle caratteristiche hardware superiori, ed integra oltre al Wi-Fi anche un interfaccia Bluetooth Low Energy (BLE).

Questo rende anche possibile adottare l'elettronica smart anche su prodotti di fascia entry-level, che attualmente montano un controllo totalmente analogico se non addirittura meccanico, che tuttavia nei prossimi anni diventerà non più a norma (in quanto per garantire il risparmio energetico si chiede che sia implementata una gestione a fasce orarie).

Dato il basso costo di alcuni di questi dispositivi non tutti di fabbrica usciranno connessi al cloud “IRsap NOW”, ma avranno una modalità di funzionamento solo locale tramite BLE. Questo perché l'avere svariate migliaia di dispositivi (si pensa di arrivare a produrne circa 60.000 all'anno) connessi al cloud comporterebbe dei costi insostenibili. Inoltre per molti clienti, soprattutto dei modelli più di fascia bassa, l'uso della app è solo un impedimento e preferiscono avere una gestione manuale del dispositivo.

Se il cliente lo deciderà potrà acquistare la possibilità di connetterlo alla piattaforma NOW e renderlo quindi gestibile non solo in locale ma anche da remoto, oltre che ottenere tutti i benefici e le funzioni che “IRsap NOW” può offrire, come la raccolta di dati statistici sul funzionamento dell'impianto.

In questi dispositivi è previsto infine il supporto al protocollo *Matter*, il nuovo standard aperto per l'interoperabilità fra dispositivi di marche diverse.

Quanto detto sopra consegue che il numero di test che dovranno essere effettuati ad ogni nuovo rilascio aumenta di parecchio: diventa ancora più vantaggioso andare ad utilizzare le procedure di testing automatizzato visto fin ora.

6.2 Test di dispositivi RF

Sempre il cliente IRSAP dispone di un altro prodotto nell'ecosistema "IRSAP NOW", un sistema di controllo per impianti idraulici composto da teste termostatiche, termostati ambiente ed un modulo di interfacciamento verso la caldaia. Tutti questi dispositivi (Figura 6.1) comunicano con una unità centrale di controllo (CU), che è il cuore del sistema, tramite un protocollo Radio Frequency (RF) a 868MHz.



Figura 6.1: Sistema IRSAP NOW per impianti idraulici

Una delle difficoltà attuali è l'impossibilità di andare a testare in maniera isolata i dispositivi RF e l'unità centrale.

Ad esempio per validare una nuova versione firmware della CU è necessario avere a disposizione tanti dispositivi quanti ne supporta al massimo il sistema, ovvero 48, un numero che rende molto complessa anche a livello logistico l'operazione (si pensi ad es. solo al fatto che questi dispositivi funzionano a batterie che dovrebbero essere continuamente mantenute cariche).

Si potrebbe quindi, mediante un’interfaccia radio opportunamente tarata sulle frequenze usate dal sistema, andare a simulare i dispositivi lato software, consentendo con un’unica interfaccia radio di simulare tutti i 48 dispositivi senza fisicamente averli in ufficio.

Oltre a ciò in questo progetto si può porsi anche ad un livello più basso: l’unità centrale contiene due microcontrollori (e di conseguenza due firmware) che comunicano fra di loro mediante un protocollo seriale (su interfaccia UART). Uno integra il ricetrasmettitore a radiofrequenza ed è deputato a gestire la comunicazione con i dispositivi finali, l’altro invece implementa tutte le logiche di funzionamento incluso l’interfacciamento con il cloud, che avviene tramite porta ethernet.

Si potrebbe quindi andare a testare i due firmware in maniera isolata l’uno dall’altro, andando a simulare sempre tramite software la controparte mancante. Questo evita di dover testare tutto il sistema, in particolare la parte RF, quando uno solo dei due software viene aggiornato. Infatti la parte radio viene aggiornata decisamente più di rado rispetto alla gestione delle logiche di funzionamento, ma è la più dispendiosa in termini di tempo da testare (anche perché gestendo a livello fisico la comunicazione RF, in linea teorica, ogni volta andrebbero rieseguiti i test di laboratorio per assicurarsi che i parametri radio restino nei limiti imposti dalla normativa vigente).

6.3 Test di un termostato

In IOTINGA infine abbiamo altri progetti firmware più complessi su cui questa metodologia di test potrebbe essere applicata. Uno su tutti il progetto YAT (Yet Another Thermostat), un cronotermostato Wi-Fi in grado di comunicare con la caldaia su bus OpenTherm, ma che in futuro verrà prodotto anche in versione Modbus.

La peculiarità di questo progetto è il fatto che è prevista l’interazione fra più termostati mediante BLE in modalità Long Range. In questo modo si può gestire un impianto multizona, in cui vi è un termostato “master” (alimentato tramite tensione di rete) che mantiene la connessione verso il cloud e la comunicazione con gli altri termostati “slave”, che possono essere dei

dispositivi a batteria in quanto non necessitano di una costante connessione con la rete.

Altra caratteristica è un’interfaccia utente (HMI) locale completa, che utilizza un display a matrice in bianco e nero (o in modelli futuri a colori TFT) e dei pulsanti capacitivi per navigare nei vari menù. Anche qui la sfida è riuscire a testare anche l’interfaccia utente in maniera più o meno automatizzata. Si potrebbe ad esempio escludere completamente il display e verificare tramite interfacciamento SPI che il microcontrollore arrivi a scrivere i dati correttamente nella memoria video dello schermo.

Come negli scenari visti fin ora anche qui abbiamo una possibilità di gestire configurazioni differenti, unita al fatto che questo dispositivo verrà venduto anche in modalità “white label”, ossia ogni produttore potrà scegliere di andare a personalizzare alcuni aspetti e quindi avere un firmware differente dagli altri. Conseguì quindi che ad una modifica potrebbe corrispondere un numero elevato di binari differenti da testare, uno per cliente, e ancora la necessità di effettuare dei test automatizzati per garantire la qualità di quanto finisce in mano ai clienti.

6.4 Ringraziamenti

Ringrazio IOTINGA s.r.l, ed in particolare Matteo e Simone, per questi 3 anni di lavoro assieme, dove ho potuto sperimentare e migliorare la mia conoscenza su un numero di tecnologie ed ambiti che è impossibile elencare in un paragrafo.

Ringrazio IRSAP s.p.a., ed in particolare Marco, Leandro, Daniele, Mario, e tutto il team di “IRsap NOW” con cui ho potuto lavorare assieme in questi ultimi anni per realizzare quanto si è visto solo in piccola parte in questo documento.

Glossario

“IRsap NOW” Piattaforma di termoregolazione cloud dell’azienda IRSAP s.p.a. Consente di gestire attraverso un’unica app iOS/Android tutti i dispositivi di termoregolazione connessi del catalogo IRSAP, quali radiatori elettrici, impianti idraulici e sistemi VMC. 1, 2, 18, 27, 64, 65, 67

broker MQTT Componente server del protocollo MQTT che si occupa di instradare i messaggi pubblicati dai client ad essi connessi ad altri client a seconda dei topic MQTT ai quali si sono sottoscritti. 69

cloud Insieme di tecnologie che permettono di archiviare ed elaborare i dati in rete, anziché su un dispositivo locale. 1, 70

firmware La componente software che viene eseguita su un dispositivo embedded. 3–5, 9, 11, 15–18, 27, 28, 30, 34, 39, 41, 42, 45, 46, 66, 67

Git Sistema di versionamento del codice sorgente gestito distribuito. Creato inizialmente da Linus Torvalds per l’uso in Linux, è ad oggi il sistema più utilizzato ed evoluto sul mercato. 38

i2c Protocollo di comunicazione seriale su due segnali elettrici (SDA ed SCL) per la comunicazione fra chip embedded, quali microcontrollori e sensori. 16

IoT Core Servizio serverless di broker MQTT gestito da Amazon Web Services (AWS). 70

Job IoT Core Sistema di distribuzione di processi batch da eseguire sui dispositivi collegati ad IoT Core. Ogni processo è identificato da un documento JSON, a libera scelta di chi lo crea, che il dispositivo interpreta per ottenere l'operazione da effettuare. Il dispositivo aggiorna lo stato di esecuzione del processo batch in AWS IoT Core, ed AWS IoT Core permette di settare delle logiche di distribuzione graduali del processo (ad esempio annulla se il job fallisce in più del 5% dei dispositivi, ecc). . 33, 46

MQTT Protocollo di scambio di messaggi di tipo publish/subscribe pensato per l'utilizzo su dispositivi con scarse risorse hardware, quali ad esempio i dispositivi IoT. 18, 21–24, 28, 33–36, 46, 70

serverless Modello di sviluppo cloud che consente di creare ed eseguire applicazioni in rete senza doversi preoccupare della gestione dei server. Tipicamente il sistema scala in maniera automatica, andando a creare/eliminare risorse in base al carico del sistema. 1, 69

topic MQTT Stringa testuale che identifica un canale di comunicazione sul quale un client MQTT può inviare o ricevere messaggi. 21–23, 35, 36

Acronimi

ADC Analog to Digital Converter. 16, 17

AP Access Point. 15, 16, 27, 33, 37, 50, 51

AWS Amazon Web Services. 69

BLE Bluetooth Low Energy. 64, 66

GPIO General Purpose Input Output. 17, 30, 31, 33, 34

HMI Human Machine Interface. 27, 67

IoT Internet of Things. 1, 70

LED Light Emitting Diode. 2, 7, 14, 15, 18, 20, 23, 26–28, 33, 46, 48, 51, 53, 56, 59

NTC Negative Temperature Coefficient. 14

OTA Over The Air update. 5, 8, 17, 33, 35, 46

PWM Pulse Width Modulation. 16

RE Radiatore Elettrico. 2, 13, 14, 24, 35, 37, 39, 42, 63

RF Radio Frequency. 65, 66

RGBW Red Green Blue White. 14, 15

RTC Real Time Clock. 16

RTOS Real Time Operating System. 4, 16

SDK Software Development Kit. 11, 16, 30

SPI Serial Peripherial Interface. 16, 67

STA Station. 15, 16, 33, 37

TLS Transport Layer Security. 16, 21

UART Universal Asynchronous Receive and Transmit. 16, 31, 33, 34, 66

USB Universal Serial Bus. 31

VMC Ventilazione Meccanica Controllata. 2, 69