

**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

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Il progetto IRSAP radiatore elettrico . . . . .	2
1.2	Problemi aperti . . . . .	3
<b>2</b>	<b>Definizione del problema</b>	<b>5</b>
2.1	Obblighi normativi . . . . .	5
2.2	Conseguenze economiche . . . . .	6
2.3	Prevenzione di danni fisici . . . . .	6
2.4	Prevenzione dei danni software . . . . .	7
2.5	Prassi attuale di test . . . . .	11
2.5.1	Test durante lo sviluppo . . . . .	11
2.5.2	Test di accettazione interna . . . . .	11
2.5.3	Test di accettazione del produttore . . . . .	13
2.6	Problemi dell'approccio attuale . . . . .	13
2.7	Requisiti software per l'automazione . . . . .	14
<b>3</b>	<b>Approccio</b>	<b>17</b>
3.1	Caratteristiche hardware . . . . .	17
3.1.1	Scheda elettronica . . . . .	17
3.1.2	Modulo Telit . . . . .	19
3.2	Caratteristiche software . . . . .	22
3.2.1	Stati interni del dispositivo . . . . .	23
3.2.2	Termoregolazione . . . . .	24

3.2.3	Comunicazione cloud . . . . .	26
3.2.4	API di configurazione locale . . . . .	32
3.2.5	Interfaccia utente . . . . .	34
3.3	Interfacciamento con il sistema di test . . . . .	35
3.3.1	Interazione con il dispositivo fisico . . . . .	35
3.3.2	Interazione con la sola elettronica . . . . .	36
3.3.3	Esecuzione del firmware in un emulatore . . . . .	37
3.4	Configurazione selezionata per i test . . . . .	38
<b>4</b>	<b>Implementazione</b>	<b>39</b>
4.1	Interfacciamento hardware . . . . .	39
4.2	Motore di esecuzione test . . . . .	40
4.2.1	Interfacciamento con l'I/O . . . . .	42
4.2.2	Interfacciamento con il cloud . . . . .	42
4.2.3	Interfacciamento con il Wi-Fi . . . . .	44
4.3	Integrazione continua . . . . .	46
<b>5</b>	<b>Validazione sperimentale</b>	<b>49</b>
5.1	Scenari da testare . . . . .	49
5.1.1	Abbinamento del dispositivo . . . . .	49
5.1.2	Downgrade locale . . . . .	51
5.1.3	Aggiornamento OTA cloud . . . . .	52
5.1.4	Ripristino di fabbrica . . . . .	54
5.1.5	Ripristino di fabbrica disconnesso . . . . .	55
5.1.6	Termoregolazione base . . . . .	57
5.1.7	Modalità stand-by . . . . .	58
5.1.8	Funzionamento disconnesso . . . . .	60
5.2	Prestazioni . . . . .	62
<b>6</b>	<b>Conclusioni</b>	<b>65</b>
6.1	Futuri radiatori elettrici . . . . .	65
6.2	Test di dispositivi RF . . . . .	67
6.3	Test di un termostato . . . . .	68

<b>A Implementazione casi di test</b>	<b>71</b>
A.1 Test pairing . . . . .	71
A.2 Test downgrade . . . . .	73
A.3 Test OTA . . . . .	74
A.4 Test ripristino di fabbrica . . . . .	76
A.5 Test ripristino di fabbrica disconnesso . . . . .	77
A.6 Test termoregolazione . . . . .	79
A.7 Test standby . . . . .	81
A.8 Test funzionamento offline . . . . .	83



## Introduzione

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.

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 in continua espansione attualmente si compone di 17 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 e sono utilizzati attivamente dai clienti circa 3000 radiatori elettrici smart.

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 Problemi aperti

Una delle sfide da affrontare è quella del garantire la qualità del lavoro svolto. Questo è ancora più importante quando si parla di dispositivi di *elettronica di consumo*, categoria in cui rientrano i dispositivi IoT, in quanto l'utente finale (il Consumatore) tipicamente non ha alcuna conoscenza tecnica specifica.

Per questo motivo è fondamentale la prevenzione, che si ottiene tramite una scrupolosa validazione di tutte le componenti del prodotto, e fra

queste il software. Queste attività sono tuttavia molto dispendiose in termini di tempo, in quanto aumentando il numero e la complessità dei prodotti, e di pari passo il numero di consumatori finali, le risorse da dedicarvi sono sempre maggiori.

In questa tesi vedremo come è stato possibile realizzare un sistema completamente automatizzato per l'esecuzione dei *test di accettazione* del Firmware del dispositivo.

# Capitolo 2

## Definizione del problema

Nessun utente installerebbe in casa propria un dispositivo che non è in grado di svolgere la funzione per la quale è stato acquistato, o che necessita di continue attenzioni. A maggior ragione nessun venditore consiglierebbe ad un cliente un prodotto che funziona male, visto che poi è lui che ci mette la faccia e che deve fornire una garanzia al cliente riguardo al dispositivo che gli ha venduto.

A maggior ragione il malfunzionamento di un impianto di riscaldamento può essere particolarmente fastidioso in quanto questo ricopre un ruolo fondamentale per il benessere domestico.

### 2.1 Obblighi normativi

Trattando prodotti destinati alla grande distribuzione e quindi all'acquisto da parte di privati si applica il *codice del consumo*, che regolamenta i contratti di vendita tra aziende e privati senza la necessità di negoziazioni in quanto diritti e doveri delle parti vengono stabiliti direttamente dalla legge italiana ed europea.

Esso stabilisce che le conseguenze legali legate alla sicurezza dei prodotti introdotti sul mercato sono a carico del produttore, ovvero di colui che appone il proprio marchio sullo stesso. Invece la garanzia sui difetti di produzione, che in Italia ha durata minima di 2 anni, è a carico del venditore. Tuttavia

il venditore, tramite accordi specifici, a sua volta si rivale solitamente sul produttore.

In definitiva il principale responsabile della qualità e sicurezza dei prodotti è il produttore.

## 2.2 Conseguenze economiche

Per quanto concerne i danni accidentali, ad esempio un radiatore che perdendo acqua allaga la casa, il produttore può tutelarsi avvalendosi di un'assicurazione, che, una volta appurato che il produttore ha messo in atto tutte le misure preventive possibili, risarcisce il danno.

Invece per quel che riguarda i difetti di fabbrica, che includono difetti del software, questi sono totalmente a carico del produttore, che è tenuto a correggerli o dove non sia possibile farlo a sostituire i prodotti difettosi con una campagna di richiamo.

Dobbiamo tenere presente che il richiamo di un prodotto ha un costo molto elevato per il produttore, soprattutto nel nostro caso, in quanto trattando di prodotti di *design* molto spesso nella loro spedizione per effettuare il reso questi vengono danneggiati in maniera irreparabili e devono quindi essere necessariamente sostituiti con un nuovo esemplare.

Oltre ciò c'è il danno d'immagine, se l'utente è poco soddisfatto va a parlare male del prodotto, ed una recensione negativa è una macchia difficilissima da cancellare, una volta che il prodotto si è fatto la fama di essere poco affidabile non è semplice, se non è impossibile, rimediare.

È quindi importantissimo per il produttore mettere in atto delle procedure tali da ridurre il più possibile la probabilità che questi difetti emergano.

## 2.3 Prevenzione di danni fisici

Per quanto concerne il prevenire danni fisici esistono leggi italiane e direttive europee che si applicano a determinate categorie di prodotti di elettronica di consumo. Nel nostro caso si applica, ad esempio, la direttiva 2014/53/UE

RED<sup>1</sup>, che ha campo di applicazione sui dispositivi elettronici in grado di effettuare comunicazioni radio (il che include l'uso del Wi-Fi).

Lo scopo di leggi e direttive è fissare i principi generali che essenzialmente si riducono al dire che il prodotto deve essere progettato in una maniera tale che non possa, se usato nei modi e per gli scopi indicati dal produttore, arrecare danni a cose, persone, o animali domestici.

Per poter commercializzare un prodotto all'interno dell'Unione Europea è necessario a rilasciare una *dichiarazione di conformità* ed apporre sul prodotto il marchio **CE**. A differenza di quanto si crede comunemente, e di quanto avviene negli Stati Uniti, il marchio **CE** è un'autocertificazione del produttore, che dichiara assumendosene la responsabilità di aver realizzato il prodotto *a regola d'arte*.

Chiaramente il modo più semplice (ma non l'unico) che ha il produttore per dimostrare di aver seguito la *regola dell'arte* è il seguire durante la sua progettazione le *norme di prodotto* che si applicano, dimostrandolo facendo eseguire dei test di laboratorio ad un ente terzo certificatore.

Tuttavia non è sufficiente eseguire solamente dei test a campione: è anche necessario mettere in piedi una procedura per collaudare ogni esemplare che esce dalla fabbrica, proprio per evitare che vi siano difetti sistematici in un intero lotto di produzione. Questo si ottiene mediante procedure di *quality testing*.

Nel nostro caso questi test sono effettuati sia dal produttore della scheda elettronica, che collauda ogni scheda a livello elettrico e funzionale, sia da IRSAP stessa, che collauda ogni radiatore assemblato (elettronica più corpo scaldante) per verificare che funzioni correttamente e che non presenti pericoli (ad esempio dispersioni elettriche).

## 2.4 Prevenzione dei danni software

Purtroppo attualmente non esistono (ancora) obblighi normativi per i prodotti di *elettronica di consumo* che impongano la realizzazione del software

---

<sup>1</sup><https://eur-lex.europa.eu/legal-content/IT/TXT/PDF/?uri=CELEX:32014L0053&from=lv>

seguendo determinati standard. Essi esistono solo per settori specifici, come l'aereospaziale, il medicale, il militare, ecc.

Il fatto che non esistano obblighi non significa che volontariamente un produttore non possa auto imporsi determinati standard internamente. Lo standard System Integrity Level (SIL) è il più riconosciuto in ambito industriale per la valutazione della sicurezza del software ed è contenuto nella norma IEC 61508.

Esso definisce 4 livelli, dal meno stringente (livello 1) al più restrittivo (4). Ogni livello presenta requisiti per quanto concerne la progettazione, lo sviluppo e la validazione del software realizzato, che sono riassunti in Tabella 2.1:

<b>metrica</b>	<b>SIL 4</b>	<b>SIL 3</b>	<b>SIL 2</b>	<b>SIL 1</b>
Definizione delle specifiche e dei requisiti di progetto	Formale (descrizione strutturata e dettagliata dei comportamenti in linguaggio matematico)	Semi-Formale (descrizione strutturata e dettagliata dei comportamenti in linguaggio natural)	Informale (descrizione in linguaggio naturale)	Informale (descrizione in linguaggio naturale)
Configuration-Management	Completa (automatica per sviluppo e produzione)	Completa (automatica per sviluppo e produzione)	Con l'ausilio di strumenti	Manuale
Prototipazione	Si	Si	Opzionale	Opzionale
Progettazione strutturata (uso di flowchart, schemi relazionali o altro)	Si	Si	Preferenziale	Opzionale
Verifica della progettazione	Si (da parte del team di progetto)	Si (da parte del team di progetto)	Si (da parte del team di progetto)	Si (esperti esterni)

Project-Management	Si	Si	Si	Preferenziale
Valutazione tecnica indipendente	Si	Preferenziale	Opzionale	Opzionale
Analisi della gestione dei dati (necessaria per GDPR)	Si	Si	Si	Si
Analisi Statistica (es. Unit Testing)	Si	Si	Opzionale	Opzionale
Analisi Dinamica (es. Test automatici)	Si	Si	Si	Si
Testing indipendente	Si (da org. esterna)	Si (da org. esterna)	Si (da Committente)	Opzionale
Monitoraggio indipendente (test periodici)	Si (da org. esterna)	Si (da org. esterna)	Si (da Committente)	Opzionale

Tabella 2.1: standard SIL

Nei progetti che sviluppiamo in IOTINGA utilizziamo solitamente il livello 2. Il limite principale dei livelli 3 e 4 è la definizione della specifica, che deve essere formale o semi-formale. I clienti solitamente ci forniscono le specifiche in linguaggio naturale, in quanto il fare diversamente sarebbe molto dispendioso in termini di tempo, e ciò poco si adatta con il modello di sviluppo *agile* che seguiamo.

Infatti i prodotti di elettronica di consumo devono rispondere velocemente alle esigenze del mercato, così da rimanere competitivi verso la concorrenza,

e quindi devono essere sviluppati velocemente e poter evolvere durante la loro vita con l'aggiunta di nuove funzionalità per rimanere appetibili.

I prodotti della precedente generazione utilizzavano il software per una sola gestione delle periferiche fisiche del prodotto, senza la necessità di interfacciarsi con sistemi terzi. Il software era quindi molto semplice, e una volta validato in laboratorio non c'erano grosse possibilità che smettesse di funzionare (salvo guasti hardware).

Nella nuova generazione di prodotti invece l'hardware è più evoluto, passiamo ai microcontrollore ad 8 bit quelli a 32-bit, con funzionalità sempre più assimilabili a quelle di un sistema general purpose, quali ad esempio una gestione di programmazione concorrente tramite un Real Time Operating System (RTOS), 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 l'introduzione di nuove funzionalità in un prodotto esistente, ma anche per mantenere il software al passo con l'evoluzione degli altri sistemi lato server a cui si collega, che nel tempo necessariamente evolvono a volte in maniera non retrocompatibili.

Bisogna infatti 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 continui a funzionare. Sebbene la legge obblighi solo a garantire il funzionamento soltanto per il periodo di durata della garanzia, è chiaro che se il prodotto dopo 2 anni smette di funzionare come in origine il cliente sarebbe particolarmente arrabbiato e probabilmente per i prossimi acquisti si rivolgerebbe altrove.

Non basta più quindi andare a validare il software una sola volta quando si fanno i test di laboratorio prima di immettere il prodotto sul mercato, ma la validazione deve essere continua, per tutta la vita del prodotto, per ogni singolo aggiornamento Over The Air update (OTA) che viene mandato ai dispositivi.

## 2.5 Prassi attuale di test

Posto che il rischio zero è irraggiungibile è necessità del produttore mettere in campo tutte le procedure possibili per ridurre il più possibile la probabilità che i problemi sopraccitati si verifichino.

Un ruolo fondamentale lo assumono quindi i test funzionali che si occupano di validare ogni rilascio firmware per il dispositivo. Questo è attualmente fatto in tre momenti:

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

### 2.5.1 Test durante lo sviluppo

Quando uno sviluppatore finisce di implementare una nuova funzionalità o risolve un bug prima di poter considerare l'attività conclusa ed integrare il proprio lavoro nel ramo di sviluppo principale è tenuto ad effettuare dei test.

Questi sono volti ad assicurare:

- che le funzioni che prima c'erano continuino a funzionare come prima (test di regressione)
- che le eventuali nuove funzionalità aggiunte siano conformi alla specifica del cliente
- che gli eventuali i bug risolti non siano più presenti

Questi test possono essere svolti sia in maniera automatizzata, ovvero tramite strumenti di *unit testing*, che è il modo preferibile di procedere, sia manualmente, installando il software su un dispositivo.

### 2.5.2 Test di accettazione interna

I test di accettazione interna si occupano di validare l'output finale del processo di sviluppo, il binario eseguibile, prima che arrivi nelle mani del cliente.

Solo se una versione del software passa tutti questi test può essere consegnata al cliente.

Questi test si pongono dal punto di vista dell'utilizzatore finale e non entrano nel merito di come il codice è stato strutturato internamente, che viene trattato come una scatola chiusa.

Attualmente vengono effettuati esclusivamente in maniera manuale, con il solo ausilio di un documento di definizione contenente una serie di casi da testare. Per ognuno di essi è indicata una serie di passaggi, ognuno riportante:

- 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.

Per ogni passo chi esegue il test deve effettuare l'azione e verificare che il risultato atteso sia conforme a quanto indicato. Se così non è bisogna indicare a margine del documento cosa è andato storto, cercando di raccogliere quante più informazioni possibili per aiutare lo sviluppatore a replicare il problema per poi risolverlo.

Una nota interessante è che questa procedura preferiamo farla eseguire a persone che non sono sviluppatori del progetto in questione. 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.

### 2.5.3 Test di accettazione del produttore

Sul produttore ricade la responsabilità (legale) del prodotto che viene immesso sul mercato con il proprio marchio 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.

## 2.6 Problemi dell'approccio attuale

Per i test durante lo sviluppo (vedi sottosezione 2.5.1) non ci sono particolari problemi, come detto esistono una moltitudine di strumenti con i quali il lavoro può essere automatizzato.

La fase di test di accettazione del produttore (in sottosezione 2.5.3) invece è piena responsabilità del produttore che decide come meglio crede come svolgerla.

La componente più critica ora è la fase dei test di accettazione interna (sottosezione 2.5.2). L'approccio attuale infatti presenta varie problematiche:

- la procedura è ripetitiva, e questo può facilmente indurre l'operatore a commettere errori
- l'esecuzione della procedura richiede molto tempo, che potrebbe essere dedicato ad altre mansioni
- i test sono scritti in linguaggio naturale, non formale, e questo lascia un grado di interpretazione all'operatore che svolge i test

- per quanto detto ai punti precedenti la procedura non può prendere in considerazione tutti i possibili scenari, altrimenti sarebbe troppo lunga da eseguire, pertanto solo un sottoinsieme di casi d'uso “critici” viene testata
- infine ancora per quanto sopra il numero di build effettivamente testate è un sottoinsieme di quelle prodotte, 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 arrivano nelle mani di sempre più consumatori il rischio per il produttore aumenta, e quindi viene richiesta sempre una maggiore qualità del software, che ci costringe ad effettuare sempre più test.

La cosa positiva è che vista la definizione pressoché schematica e meccanica degli scenari da validare si potrebbe pensare di scrivere un software in grado di eseguirlo in maniera del tutto automatizzata, che possa richiedere un intervento umano nullo.

## 2.7 Requisiti software per l'automazione

Vediamo quindi i requisiti che questo sistema deve avere:

- 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 dipanare 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

Osservato che sul mercato non esistono sistemi commerciali o open-source in grado di rispondere ai requisiti sopra elencati. L'obiettivo di questa tesi è implementare e analizzare sperimentalmente un sistema in grado di rispondere a questa esigenza.



# Capitolo 3

## Approccio

Prima di descrivere l'implementazione è necessario andare a sviscerare il dispositivo RE andando a vedere i dettagli più tecnici.

### 3.1 Caratteristiche hardware

Il corpo principale dei radiatori elettrici di IRSAP è del tutto identico al radiatore idraulico, ed è realizzato mediante tubi d'acciaio saldati in un'unico corpo all'interno del quale circola il liquido termovettore, che tipicamente è l'acqua riscaldata da una caldaia o da una pompa di calore.

Nel caso dell'elettrico viene inserita una resistenza elettrica, in gergo specifico *cartuccia*, all'interno del radiatore, che viene riempito con del glicole, che a differenza della normale acqua ha un punto di congelamento più basso ed un punto di ebollizione più alto. Vengono infine sigillati tutti gli ingressi/uscite che normalmente consentirebbero il collegamento ad un impianto idraulico con dei tappi.

Il cuore del sistema è poi la scheda elettronica di controllo che va ad alimentare la *cartuccia*, e viene installata nella parte bassa del radiatore.

#### 3.1.1 Scheda elettronica

Attualmente la gamma di radiatori IRSAP elettrificati smart include 17 modelli distinti, ognuno dei quali disponibile in decine di colorazioni, ed ogni

anno ne vengono aggiunti di nuovi.

Per tutti questi dispositivi è quindi fondamentale, per ottimizzare i costi, avere un'elettronica unica. Attualmente vi è soltanto un'unica tipologia di scheda, che viene prodotta in due modelli leggermente diversi dal punto di vista fisico:

- *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
- *design* (Figura 3.1): 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 inglobarla all'interno della carcassa del RE in maniera da renderla invisibile, 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 CO<sub>2</sub>) e la possibilità di controllare una striscia LED Red Green Blue White (RGBW), utilizzata su alcuni modelli per un'illuminazione ambientale.

Il modello *luxury* è dal punto di vista delle caratteristiche hardware un sottoinsieme di quanto previsto dal modello superiore (*design*). Entrambe montano:

- un modulo Wi-Fi Telit GS2200M
- un circuito di alimentazione a tensione di rete (230V AC)
- un circuito composto da un relè combinato ad un triac dedicato al controllo di potenza, ossia l'accensione e spegnimento della resistenza scaldante
- 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* (il cavo nero collegato in basso a destra nella scheda in Figura 3.1)

La versione *design* include inoltre:

- un sensore di qualità dell'aria in grado di misurare i livelli VOC e CO<sub>2</sub>
- 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 NTC secondaria in grado di misurare la temperatura del corpo riscaldante, in maniera tale da migliorare l'accuratezza degli algoritmi di termoregolazione (solo per alcuni modelli)

Vista la similarità entrambe le versioni di scheda elettronica eseguono la stessa versione del firmware, che all'avvio identifica la tipologia di scheda tramite un file di configurazione caricato in fase di produzione ed abilita/disabilita di conseguenza le periferiche opzionali.

### 3.1.2 Modulo Telit

Il fulcro del sistema è il modulo Wi-Fi GS2200M. Questo microcontrollore, inizialmente prodotto da Gainspan e 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

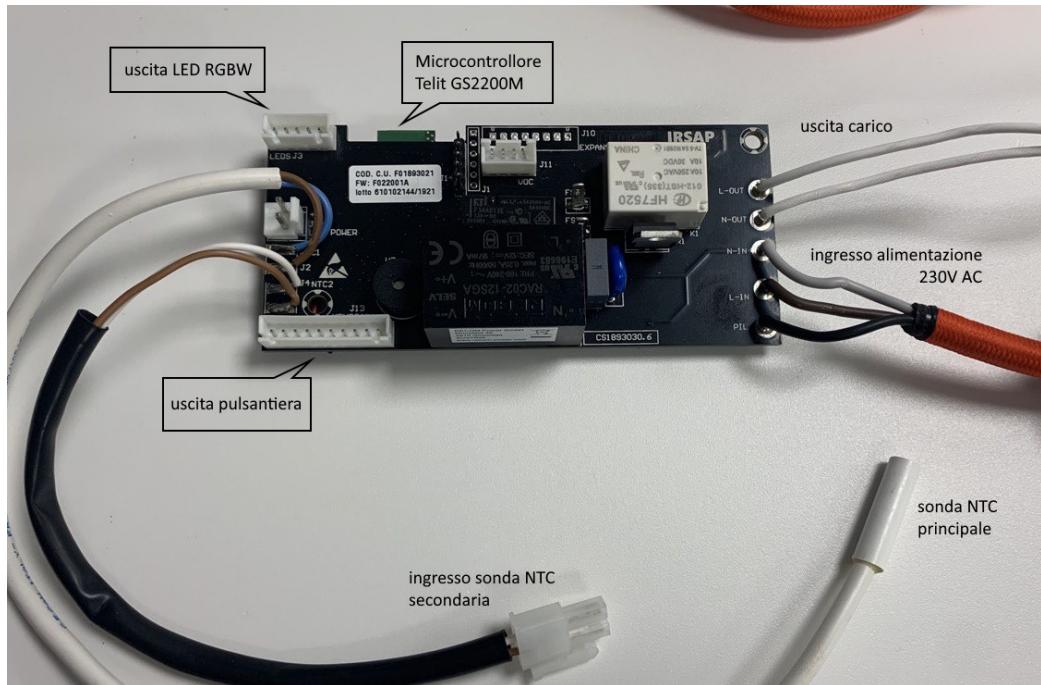


Figura 3.1: scheda elettronica modello *design*

- 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 ingressi/uscite digitali General Purpose Input Output (GPIO)
- 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 UART

- modulo Real Time Clock (RTC) interno

Il microcontrollore è dotato due due core ARM, di cui uno è deputato alla gestione della parte Wi-Fi ed esegue un firmware scritto dal produttore stesso, l'altro invece è sotto il controllo dello sviluppatore.

La toolchain fornita dal produttore si basa sull'ambiente di sviluppo proprietario IAR. Insieme a questa Telit fornisce un Software Development Kit (SDK) che offre:

- un Real Time Operating System (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 a 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 Simple Network Time Protocol (SNTP)
- aggiornamento OTA con doppia partizione
- hardware abstraction layer (HAL) per la gestione di tutti i General Purpose Input Output (GPIO), lettura dei due Analog to Digital Converter (ADC), del modulo PWM, gestione del RTC, del bus i2c ed SPI

I moduli Wi-Fi di questo tipo generalmente 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 UART e comunica con i comandi AT (standard per la gestione dei modem)
- *hostless*: il modulo Wi-Fi esegue direttamente l'applicativo senza l'aiuto di un microcontrollore principale. Tutte le funzionalità del dispositivo sono implementate sul modulo Telit

L'approccio più seguito è il primo approccio, tuttavia noi usiamo il secondo. Questo offre svariate semplificazioni: non dobbiamo gestire la distribuzione e l'aggiornamento OTA di due firmware ed abbiamo più controllo sulla parte di Wi-Fi.

Un secondo microcontrollore comporta poi problemi di approvvigionamento, soprattutto di questi tempi in cui il mercato dei componenti elettronici è problematico e importante ridurre al minimo il numero di componenti critici utilizzati, quelli di difficile rimpiazzo, usati in un progetto.

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

## 3.2 Caratteristiche software

Il firmware sviluppato è suddiviso in moduli (ognuno dei quali gira in un *thread* dedicato):

- *core*: implementa la macchina a stati che gestisce gli stati principali del dispositivo. È il primo a partire e pertanto gestisce tutta la fase di inizializzazione del sistema, riceve e gestisce gli eventi dagli altri moduli mediante una coda di messaggi.

- *termostato*: si occupa di tutto quel che concerne la regolazione della temperatura ambiente secondo le modalità di funzionamento selezionata. Oltre a ciò effettua le letture delle sonde NTC e VOC, e nei modelli che ne sono dotati gestisce anche i LED RGBW,
- *AWS*: si occupa di mantenere attiva la connessione MQTT verso IoT Core, mantenendo sincronizzato lo stato interno del dispositivo con il server “IRsap NOW” attraverso il protocollo *stateless* che vedremo in seguito
- *HMI*, si occupa dell’interazione fisica con l’utente mediante l’input di due pulsanti + e -, e l’output dei LED di stato e del buzzer. In base agli input dell’utente invia i relativi messaggi agli altri moduli
- *HTTP*, si occupa di fornire mediante webserver HTTP un’API REST con la quale è possibile interagire direttamente con il dispositivo. È utilizzata principalmente durante per la fase di abbinamento del dispositivo con la app, ma ha anche funzioni di diagnostica.
- *Network Connection Manager (NCM)*, si occupa di gestire la connessione di rete Wi-Fi.

### 3.2.1 Stati interni del dispositivo

Il dispositivo ad alto livello (*core*) può trovarsi in 3 stati:

- *non abbinato*: in attesa di un primo abbinamento da app. Ogni funzione del dispositivo è disabilitata 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, è la modalità di funzionamento usuale

stato	modo Wi-Fi	moduli attivi
<i>non abbinato</i>	AP	core, NCM, HMI, HTTP
<i>disconnesso</i>	STA	core, NCM, HMI, termostato
<i>connesso</i>	STA	core, NCM, HMI, termostato, AWS

Tabella 3.1: moduli attivati negli stati del dispositivo

In Figura 3.2 si può vedere come è possibile muoversi fra i vari stati del dispositivo:

- tramite la procedura di *abbinamento* il dispositivo passa dallo stato *non abbinato* allo stato *connesso*. Questa procedura è quella che fa l'utente tramite app quando installa per la prima volta il dispositivo a casa propria, ed ogni qualvolta il dispositivo viene reinizializzato alle impostazioni di fabbrica
- con la procedura di *ripristino di fabbrica* è possibile portare un dispositivo alle impostazioni iniziali e quindi allo stato *non abbinato*. La procedura è utile in caso di problemi di connettività sia quando si vuole abbinare il radiatore ad un altro impianto
- ogni qualvolta che la connessione Wi-Fi è assente il dispositivo passa in stato di funzionamento *disconnesso*
- quando invece la connessione torna disponibile il dispositivo passa allo stato *connesso*

Ad ogni stato del dispositivo corrisponde l'attivazione/disattivazione di uno o più moduli come in Tabella 3.1

### 3.2.2 Termoregolazione

Il modulo di termoregolazione si occupa di regolare la temperatura ambiente cercando di mantenerla uguale a quanto desiderato dal consumatore (set-point). Questo avviene comandando l'accensione della resistenza in modo on/off. Il feedback sulla temperatura ambiente è ottenuto grazie alla sonda NTC principale.

Il dispositivo ha diversi modi di funzionamento, in ordine di priorità:

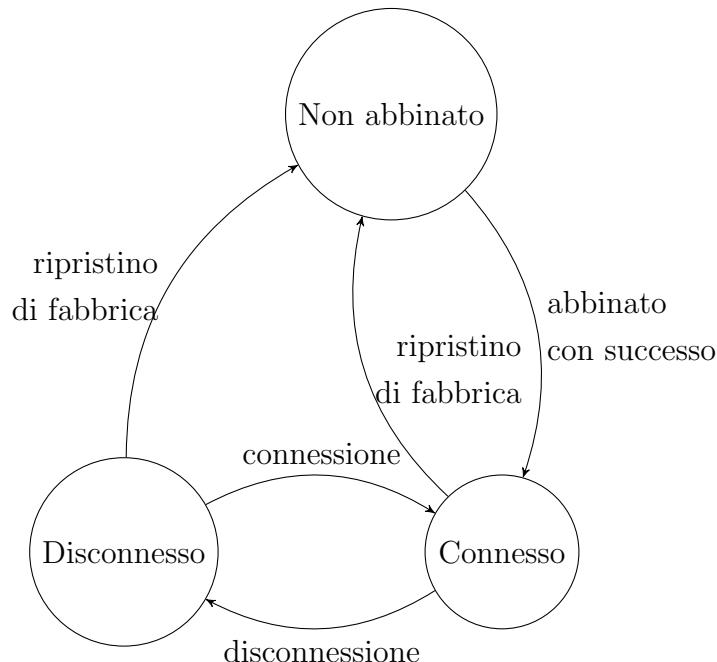


Figura 3.2: Stati del radiatore

- *standby*: dispositivo completamente spento, sia per quanto riguarda il riscaldamento che per l'illuminazione LED
- *Fil Pilote*: il dispositivo è controllato da un segnale esterno in ingresso sul cavo Fil Pilote
- *antigelo*: il dispositivo mantiene una temperatura di sicurezza (imposta dall'utente) per prevenire danni alla casa ed al dispositivo causati da una temperatura ambiente troppo bassa
- *vacanza*: funziona all'interno di un intervallo impostato fra due timestamp di inizio e di fine come in modalità *antigelo*
- *away*: imposta un set-point ridotto (ECO) in quanto l'utente non è in casa
- *manuale temporaneo*: segue un set-point manuale impostato dall'utente fino al raggiungere un timestamp di fine

- *programmato*: segue una programmazione settimanale che consente per ogni giorno della settimana di creare fino ad 8 fasce orarie
- *manuale*: segue il set-point utente che è fisso e non varia mai configurato dall'utente, quindi torna a funzionare nella modalità precedente

Tutti i modi sopraelencati sono attivabili tramite app. Invece tramite pulsantiera del dispositivo è possibile muoversi nelle modalità *manuale a tempo* oppure *manuale* secondo lo schema in Figura 3.3:

- se ci si trovava in stato *programmato* alla prima modifica manuale il dispositivo passa in modo *manuale a tempo*
- se ci si trovava in stato *manuale a tempo* invece si passa in stato *manuale* indefinito
- in ogni caso quando il dispositivo è in stato *manuale* vi resta fino a che non viene comandato il passaggio in un altro stato tramite la app

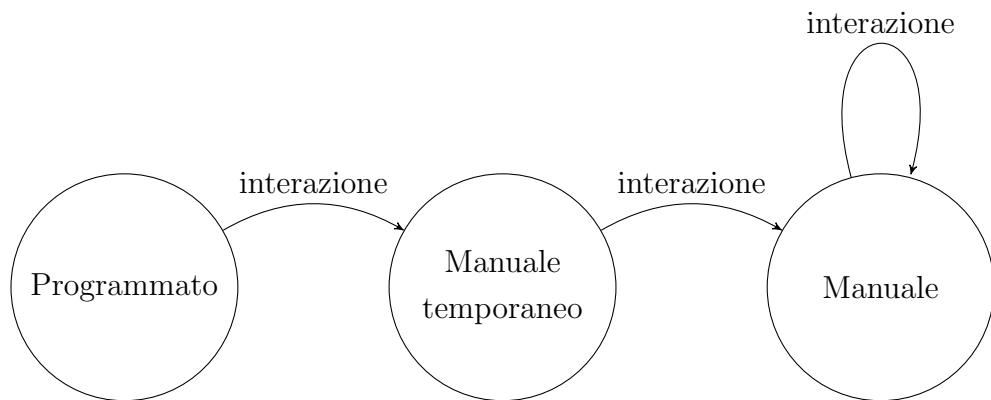


Figura 3.3: cambiamento di modi tramite tastiera

### 3.2.3 Comunicazione cloud

Il dispositivo come anticipato si collega al cloud “IRSAP NOW”. La componente server è realizzata su piattaforma Amazon Web Services (AWS) attraverso l’uso di tecnologie serverless, che consentono al sistema di poter scalare in base al carico effettivo dinamicamente.

Il protocollo di comunicazione usato dal dispositivo è MQTT, ed il broker MQTT al quale viene collegato è IoT Core.

### Autenticazione

La connessione è protetta mediante TLS 1.2 ed autenticata tramite l'utilizzo di un certificato TLS client.

Questo certificato è univoco per ogni singolo dispositivo ed è salvato all'interno del suo filesystem durante la produzione dell'elettronica. Il certificato non è emesso direttamente da AWS ma è generato attraverso una Certification Authority (CA) intermedia del produttore delle elettroniche come in Figura 3.4. Questo consente di poter generare dei certificati “offline” durante le fasi di produzione, in cui non è sempre garantita una connessione ad internet, sia consentirebbe di generare il certificato direttamente sul dispositivo andando a far sì che la chiave *privata* non sia mai nota al produttore.



Figura 3.4: Catena TLS

La registrazione presso IoT Core del dispositivo avviene quando questo si collega per la prima volta, che nel nostro caso corrisponde alla prima accensione della scheda elettronica durante il collaudo presso il produttore elettronico. A questo punto tramite delle regole viene automaticamente collegata una o più *policy* al dispositivo, che gli danno i permessi per effettuare le operazioni fondamentali necessarie per la comunicazione su cloud:

- *connect*: autorizza il client a connettersi ad IoT Core utilizzando il *clientId* corrispondente al proprio indirizzo MAC
- *publish*: consente al client precedentemente autenticato di pubblicare i messaggi sui topic ad esso riservati, ovvero i topic MQTT con prefisso `re/things/{MAC}`
- *subscribe*: consente al dispositivo di sottoscriversi per gli aggiornamenti ai topic a lui dedicati, ovvero i topic MQTT con prefisso `re/things/{MAC}`

### Protocollo stateless di comunicazione

Per quanto concerne il protocollo di comunicazione abbiamo valutato inizialmente l'uso del protocollo Device Shadowing<sup>1</sup> nativamente supportato da AWS IoT Core.

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

- il pacchetto viene codificato in JSON, la cui gestione richiede memoria e tempo CPU non trascurabile per un dispositivo embedded
- IoT Core ha 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 le possibilità di futura espansione futura
- il protocollo di comunicazione trasferisce dei delta, che sebbene riducono la dimensione di un pacchetto di dati rendono più complessa la gestione sia lato dispositivo sia lato server

Per questo abbiamo scelto di definire un nostro protocollo binario *stateless*, tramite il quale andiamo a trasferire stati completi del dispositivo.

Abbiamo deciso di mantenere per convenzione i concetti di alto livello del protocollo AWS Device Shadowing, in particolare la nomenclatura *shadow* per indicare uno stato *completo* del dispositivo, che si divide in:

- *state desired*: lo stato voluto del dispositivo. Include tutte quelle variabili di stato che definiscono una configurazione del dispositivo, quali i parametri per la termoregolazione.
- *state reported*: lo stato rilevato dal dispositivo. Include, oltre a tutte le variabili di stato dello *state desired* anche tutte le variabili contenenti dati statistici e di monitoraggio, quali temperatura ambiente, valori VOC e CO<sub>2</sub>, qualità della connessione Wi-Fi, allarmi, ecc.

---

<sup>1</sup>[https://docs.aws.amazon.com/it\\_it/iot/latest/developerguide/iot-device-shadows.html](https://docs.aws.amazon.com/it_it/iot/latest/developerguide/iot-device-shadows.html)

Attualmente sono previste dal protocollo 4 tipi di richieste, ognuna delle quali consente di eseguire un’azione verso il dispositivo o verso il cloud (come visibile in Figura 3.5):

- *get*: il dispositivo richiede al cloud l’ultimo stato *desired*
- *delete*: il dispositivo richiede al cloud di eliminare lo stato corrente
- *reported-update*: il dispositivo invia al cloud lo stato *reported* corrente
- *desired-update*: il cloud notifica al dispositivo che vi è un nuovo stato *desired*

Il tipo di messaggio dipende dal topic MQTT sul quale i pacchetti sono pubblicati. Ad un messaggio pubblicato dal dispositivo verso il cloud segue una risposta sullo stesso topic MQTT con aggiunto un suffisso che indica l’esito dell’operazione:

- */accepted*: la richiesta ha avuto successo
- */rejected*: la richiesta ha generato un errore, che è indicato nel payload della risposta

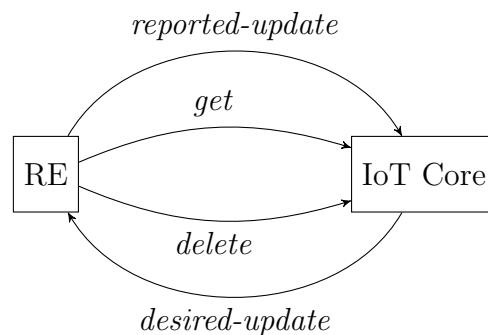


Figura 3.5: Schema di comunicazione dispositivo/cloud

### Formato messaggi cloud

Ogni messaggio su cloud include un header sempre uguale, che include i campi in Tabella 3.2.

chiave	tipo	descrizione
timestamp	u32	timestamp di generazione del messaggio
clientToken	u32	identificativo della richiesta aggiunto dal richiedente. Nelle risposte viene riportato questo valore senza variarlo, in maniera tale che il richiedente possa capire a quale richiesta una risposta di riferisce
version	u32	versione dello stato. Viene incrementata ad ogni modifica dello stato <i>desired</i> da parte della app
length	u16	lunghezza totale del messaggio in byte (header incluso)
type	u8	tipologia di messaggio inviata

Tabella 3.2: header del pacchetto

Per i messaggi che prevedono un body sono aggiunti i parametri in Tabella 3.3 (alcuni di questi parametri sono presenti solo nello stato *reported*, altri sia nel *desired* che nel *reported*).

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
totConsumption	u32	consumo cumulato in tutta la vita del radiatore
totConsumptionValue	u32	valore all'ultimo snapshot di consumo
totConsumptionTime	u32	consumo all'ultimo snapshot
modelName	char[32]	nome modello del radiatore

Tabella 3.3: stato del dispositivo

### 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

endpoint	metodo	descrizione
/irsap/state	GET	ottiene informazioni sul 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

Tabella 3.4: API locale

quale collegarsi prima che esso possa iniziare a funzionare tramite la app “IRSAP NOW”.

Questo avviene tramite Wi-Fi in quanto il modulo GS2200M non dispone di interfaccia BLE. Il cellulare effettua una connessione ad una rete Wi-Fi il cui SSID inizia con il prefisso di IRSAP (IRSAP\_). Dopo di che può chiamare le API REST implementate dal dispositivo in Tabella 3.4.

Il flusso di abbinamento funziona pertanto nel seguente modo:

- il cellulare si collega all’AP messo a disposizione dal RE
- si chiama prima l’API /irsap/state per identificare la tipologia di dispositivo
- se dovesse essercene l’esigenza la app invia al dispositivo un aggiornamento OTA locale, quindi attende che il dispositivo si aggiorni e si riconnette, ricominciando questa procedura da capo.
- nel caso si sia identificato un dispositivo RE (la stessa API sarà in futuro usata da più dispositivi della famiglia “IRSAP NOW” come gli impianti VMC) chiama l’API /irsap/wifi/scan per effettuare una scansione delle reti<sup>2</sup>

---

<sup>2</sup>questo si rende necessario poiché per questioni di privacy i cellulari con sistema operativo iOS non hanno accesso all’elenco delle reti visibili dal dispositivo

- dopo aver chiesto all’utente di scegliere una delle reti restituite dalla scansione e di averne chiesto la passphrase, qualora fosse richiesta, invia una richiesta `/irsap/provision` al RE per completare l’abbinamento
- a questo punto il RE si riavvia e prova a connettersi alla rete, mentre la app resta in attesa di ricevere tramite cloud una prima comunicazione dal RE

### 3.2.5 Interfaccia utente

La Human Machine Interface (HMI) del dispositivo permette di interagire con l’utilizzatore mediante:

- due pulsanti capacitivi, + e -
- dei led LED RGB di illuminazione dei pulsanti
- un buzzer in grado di dare un feedback acustico ad ogni tocco della pulsantiera

Attraverso i due pulsanti è possibile comandare diverse operazioni:

- pressione breve +: incrementa il set-point corrente (se RE in modo *manuale* o *manuale a tempo*)
- pressione breve -: decrementa set-point corrente (se RE in modo *manuale* o *manuale a tempo*)
- pressione per un tempo compreso fra 3 e 5 secondi del tasto -: attiva modalità *antigelo*
- pressione per più di 5 secondi del tasto -: attivazione modalità *stand-by*
- pressione prolungata dei tasti + e - per più di 5 secondi: avvia la procedura di ripristino impostazioni di fabbrica (*factory reset*)

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

### 3.3 Interfacciamento con il sistema di test

Dalle sezioni precedenti possiamo riassumere che ad alto livello il RE interagisce con l’ambiente esterno attraverso 3 interfacce:

1. cloud: invio e ricezione di stati completi (*shadow*) tramite MQTT ad AWS IoT Core
2. API locale: metodi messi a disposizione mediante il server HTTP REST del RE
3. fenomeni fisici: calore emesso, pulsanti che vengono premuti, luce e suoni emessi, temperatura che cambia, ecc

Riguardo ai primi due punti si possono usare strumenti “convenzionali” per automatizzare programmaticamente le interazioni. Esistono diversi strumenti per farlo che sono comunemente utilizzati. La complessità quindi sta nelle interazioni fisiche con il dispositivo. Possiamo pensare di affrontare il problema in 3 modi:

- interazione hardware con il prodotto finito
- interazione hardware con la scheda elettronica
- esecuzione del firmware in un emulatore

#### 3.3.1 Interazione con il dispositivo fisico

Possiamo immaginare di prendere un radiatore, sistemarlo in un ambiente attrezzato per modificarne le condizioni (come una camera climatica a temperatura controllabile) e dotato di sensori per misurare il comportamento del dispositivo. A questo punto possiamo collegare degli attuatori per eseguire automaticamente le operazioni che l’utilizzatore può fare sul RE.

Questo ci garantisce di testare uno scenario del tutto identico, o comunque quasi identico, a quello che si trova nelle case dei consumatori finali. Tuttavia l’approccio presenta una serie di problematiche che lo rendono impraticabile:

- è necessaria strumentazione molto specifica e costosa (le camere climatiche), che sebbene a disposizione di IRSAP sono occupate per altri scopi. Inoltre le misurazioni che si possono effettuare hanno i limiti dati dall'osservabilità dei fenomeni fisici, che comporta un inevitabile errore di misura che limita la precisione del test
- i test dovrebbero seguire le tempistiche dettate dai fenomeni fisici che si vanno a stimolare, come il riscaldamento di un ambiente. Questo rende il test molto lungo in termini temporali, rendendo necessari giorni per poter validare una versione firmware. Questo limita il numero di versioni che si possono realisticamente testare e rallenta l'intero flusso di rilasci.

### 3.3.2 Interazione con la sola elettronica

Viene da pensare che si possa quindi ridurre all'osso il RE andando ad eliminare tutte quelle componenti che hanno un impatto marginale o nullo sul software, che è l'oggetto dei nostri test.

Ci rendiamo conto che non è necessario prendere in esame l'intero radiatore, che è un corpo meccanico, ma si può concentrarsi sulla sola scheda elettronica, andando a simulare gli ingressi/uscite digitali/analogici con la quale interagisce con l'esterno.

Notato questo, possiamo anche fare un passo ulteriore, e rimuovere completamente anche l'elettronica, andando a testare solo il microcontrollore, che è l'unica componente del sistema sulla quale vi è un software in esecuzione. Questo semplifica ulteriormente le cose, dato che le schede elettroniche sono alimentate con tensione di rete (230V AC nominale) che renderebbe quindi necessario isolare le schede sia per questioni di sicurezza che per evitare danni alle apparecchiature collegate.

L'eliminare il fattore scheda elettronica non causa alcun problema, essendo che l'hardware viene già collaudato, come detto all'inizio del capitolo 2, durante la produzione in fabbrica. Gli scopi infatti di questa fase di test è la validazione di un software assumendo che l'ambiente di esecuzione funzioni come da specifica hardware.

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) ed analogici 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

Vediamo infine un’ultima possibilità, quella di eseguire il firmware non sull’hardware reale ma su un sistema emulato.

Questo presenta alcune problematiche nel nostro caso. In primis è difficile realizzarlo sulla piattaforma hardware Telit, essendo proprietaria è difficile ottenere tutte le specifiche di funzionamento per andare ad emularne in maniera quanto più simile possibile il funzionamento. Su altri microcontrollori (ad esempio il popolare Espressif ESP-32) è possibile con il software di emulazione QEMU andare a simulare il microcontrollore in maniera molto fedele all’originale.

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 dispendioso) avrebbe come effetto che non si sta effettivamente testando il binario che poi andrà in produzione, che era uno dei requisiti che avevamo stabilito all’inizio.

In secondo luogo l’emulazione comunque non è del tutto adatta a test di accettazione, in quanto non va veramente a validare l’interazione del software con l’hardware reale, e soprattutto quando si parla di interfacce radio come il Wi-Fi il comportamento potrebbe essere particolarmente diverso. A titolo di esempio infatti uno dei bug principali che abbiamo avuto sul RE era proprio l’incompatibilità con alcuni tipi di Access Point Wi-Fi, una condizione che sarebbe stata impossibile da diagnosticare senza far girare il software su hardware fisico.

### 3.4 Configurazione selezionata per i test

Concludiamo quindi stabilendo che il modo migliore di procedere è realizzare un dispositivo hardware in grado di interfacciarsi a livello elettrico con un kit di sviluppo (*devkit*) del modulo Telit GS2200M. Tramite questo hardware sarà possibile simulare tramite software le interazioni con il dispositivo RE mediante un apposito software di test.

# Implementazione

## 4.1 Interfacciamento hardware

Come visto alla fine del capitolo precedente ho scelto di interfacciarmi direttamente con il *devkit* del Telit.

Il kit di sviluppo mette a disposizione tutti i pin connessi al microcontrollore su comodi pin header. Integra inoltre un convertitore Universal Asynchronous Receive and Transmit (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 senza necessità di ulteriori hardware.

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 generare output analogici con sufficiente precisione per i nostri scopi.

Il raspberry dispone inoltre di un'interfaccia Wi-Fi a 2.4Ghz che consente di agire sia come AP per consentire al RE di connettersi, che come STA consentendo di connettersi all'AP del RE.

In Figura 4.1 si può vedere l'hardware che ho costruito. In alto si può vedere il Raspberry Pi su cui gira l'esecutore di test ed in basso il *devkit* del

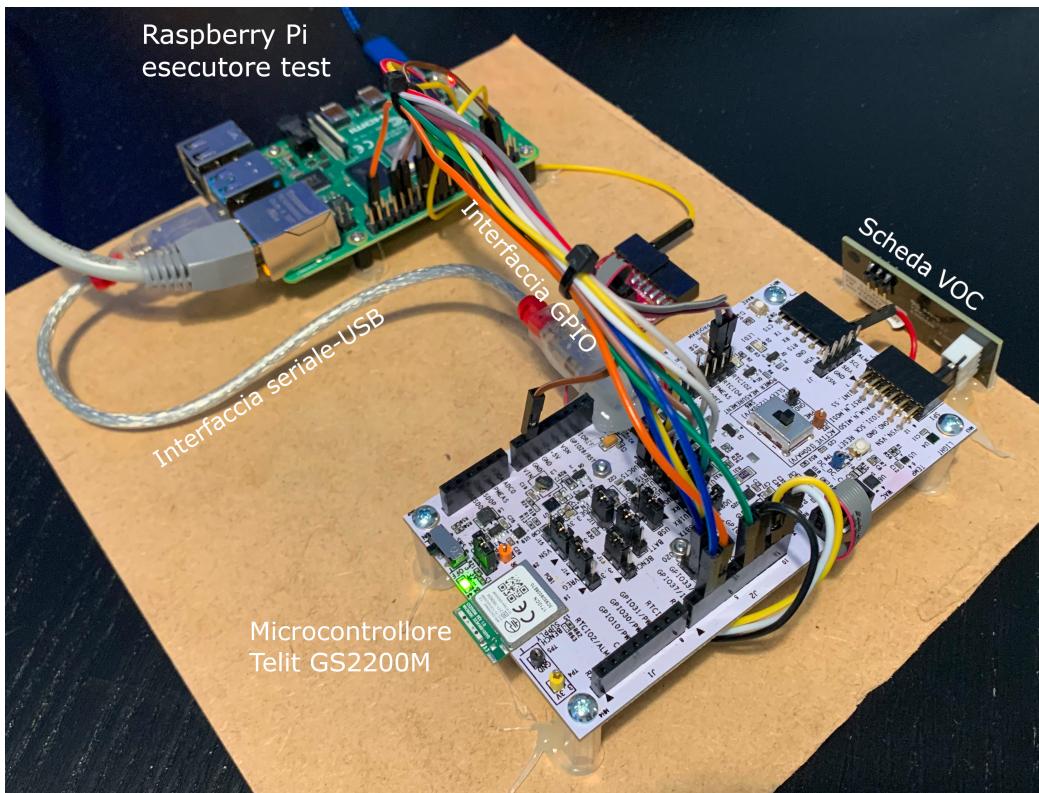


Figura 4.1: hardware di test realizzato

GS2200M. I due componenti sono collegati come si vede da un fascio di cavi che interfaccia i GPIO del Raspberry con quelli dell'RE, ed il cavo USB che porta la seriale di debug e programmazione.

## 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`, sotto forma 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 risiedere 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 oggetti che consentono l'interazione con i vari moduli visti al capitolo precedente:

- `io`: è il 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`, lo standard in ambiente Python per la realizzazione di *unit test*. Definendo un file `conftest.py` sono andato ad integrare la libreria di interazione con l'hardware citata 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 `RPi.GPIO`.

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 siano raccolti dal sistema Continuous Integration (CI) per scopi di debug dei 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 AWS 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 broker MQTT MQTT differente (come può essere Mosquitto MQTT).

Inoltre IoT Core mette a disposizione alcune funzionalità difficilmente replicabili con altri broker MQTT. In particolare la funzionalità dei Job IoT Core, ossia dei processi batch che è possibile programmare e fare eseguire ai dispositivi. Essi consentono di distribuire l'esecuzione di operazioni batch. 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 MQTT ed il resto del sistema. Il che si ottiene disabilitando le regole che invocherebbero le funzioni AWS *lambda* per il dispositivo con seriale sotto esame.

Il raspberry di test si interfaccia quindi con il server 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, ma è anzi fondamentale per assicurare una buona riuscita del test, dato che lo scopo è appunto anche il valutare l'integrazione con la componente server di basso livello.

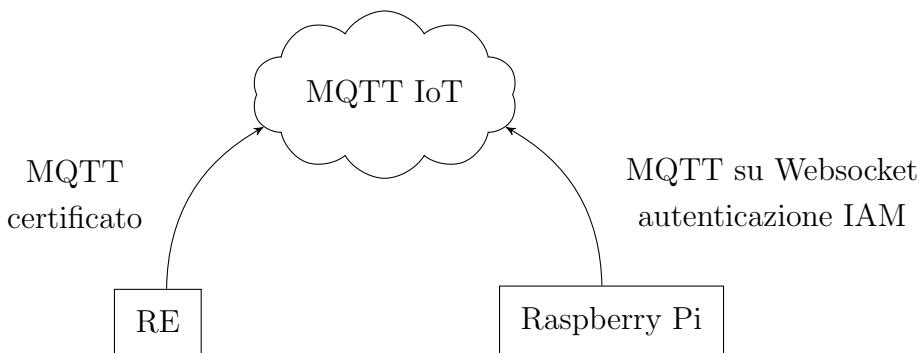


Figura 4.2: interfacciamento fra raspberry e cloud

A livello implementativo utilizzo per collegare il *test runner* ad IoT Core la libreria ufficiale di AWS, *awsiot sdk*. Questa libreria, a differenza dei normali client MQTT, consente di autenticarsi grazie a credenziali IAM, ovvero API key, non richiedendo quindi la generazione di certificati TLS ad-hoc per il Raspberry di test.

Il modulo cloud del sistema di test, oltre a gestire a basso livello la connessione broker 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 opportunamente ed inviato sul topic MQTT scelto in base al tipo di messaggio passato come parametro. Viceversa ogni messaggio ricevuto sui topic MQTT a cui il dispositivo fa la sottoscrizione viene decodificato e salvato in una coda locale. Viene fornito un metodo `receive(timeout)` per scodare l'ultimo messaggio dalla coda, attendendo fino ad un timeout opzionalmente specificato.

Per quanto concerne la gestione dei Job IoT Core viene utilizzata la libreria ufficiale Python di AWS *boto3*. Fornisco quindi 3 metodi:

- `job_create(document)`: crea un job con il documento JSON specificato
- `job_state(job_id)`: ottiene lo stato di un job con determinato id
- `job_delete(job_id)`: elimina un job precedentemente creato

#### 4.2.3 Interfacciamento con il Wi-Fi

Il Wi-Fi del Raspberry deve poter funzionare in due modalità:

- Station (STA): si usa per collegarsi all'access point del RE in modo da utilizzare l'API REST locale di configurazione
- Access Point (AP): si usa per consentire al RE di collegarsi al cloud mediante

Il vincolo che non possano essere attive entrambe è una semplificazione, in quanto di per se si potrebbero creare due interfacce di rete virtuali in modo da avere contemporaneamente sia AP che STA. Tuttavia anche il radiatore può trovarsi in una sola di queste due modalità contemporaneamente, per cui fintantoché il Raspberry può cambiare da un modo di funzionamento all'altro più velocemente di quanto lo può fare il RE non ci sono perdite di funzionalità a seguire questo approccio.

Per la modalità client è sufficiente configurare l'interfaccia di rete per connettersi al radiatore. La rete del radiatore ha un SSID che viene definito

nel file di configurazione che è caricato al momento in cui viene istanziata la libreria.

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<sup>1</sup>)

Per creare l'access point ho quindi scelto di utilizzare il software *hostapd*. L'interfaccia di test si occupa di generare un file di configurazione per tale servizio, e di lanciare il programma con tale configurazione. I log di questo processo vengono catturati ed inoltrati al logger di sistema, così che possano essere raccolti per un eventuale debugging di test falliti.

Riguardo il server DHCP ho deciso di utilizzare *dnsmasq*, in grado di fornire sia DHCP che DNS al dispositivo. Anche in questo caso viene avviato come processo figlio ed i log vengono catturati in maniera analoga a quanto avviene per *hostapd*.

Per garantire il collegamento fra l'interfaccia Wi-Fi e la rete utilizzo delle regole di *nftables* tramite quali definisco un NAT andando ad abilitare l'opzione *masquerade* per i pacchetti in uscita. Una volta abilitato l'IPv4 forwarding da *sysctl* del kernel Linux ogni client che si collega all'AP del Raspberry è in grado di accedere alla rete.

In questa maniera sarà possibile in futuro andare a testare configurazioni di rete differenti ed andare ad indurre malfunzionamenti nella rete, quali ad

---

<sup>1</sup>lo standard definito per i prodotti venduti all'interno dell'Unione Europea

esempio perdite di pacchetti o eccessive latenze, e valutare se il dispositivo si comporta come da specifica.

### 4.3 Integrazione continua

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

Per la gestione del controllo versione del codice sorgente utilizziamo Git sul servizio *GitHub*. Come politica interna di *branching model* abbiamo scelto di usare il *trunk based development*<sup>2</sup>.

Questo prevede che vi sia sempre un unico *branch*, 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 l'uso di strumenti automatici di integrazione continua, che continuino a compilare e testare ogni nuovo commit sul *master*, in maniera da accorgersi il prima possibile di eventuali bug. Questo assicura che ci 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 *Proxmox* locale 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).

---

<sup>2</sup><https://trunkbaseddevelopment.com/>

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. Quando lo sviluppatore esegue un push sul branch *master* *Github Actions* avvia il processo di CI. Per prima cosa viene avviato il job di *compilazione* che, nel caso completi con successo, va a produrre un nuovo binario con un numero di versione *vX.Y.Z*, che è automaticamente calcolato andando ad incrementare la *patch version* rispetto all'ultimo *tag* presente nel repository.

Nel caso questo abbia successo viene quindi attivata la procedura di test automatizzato, andando a lanciare un processo sul worker che viene eseguito sul Raspberry Pi di test. A questo punto, nel caso il test sia completato con successo, GitHub Actions considera l'intero *workflow* come completato con successo e quindi informa mediante mail lo sviluppatore che il processo di CI ha avuto successo.

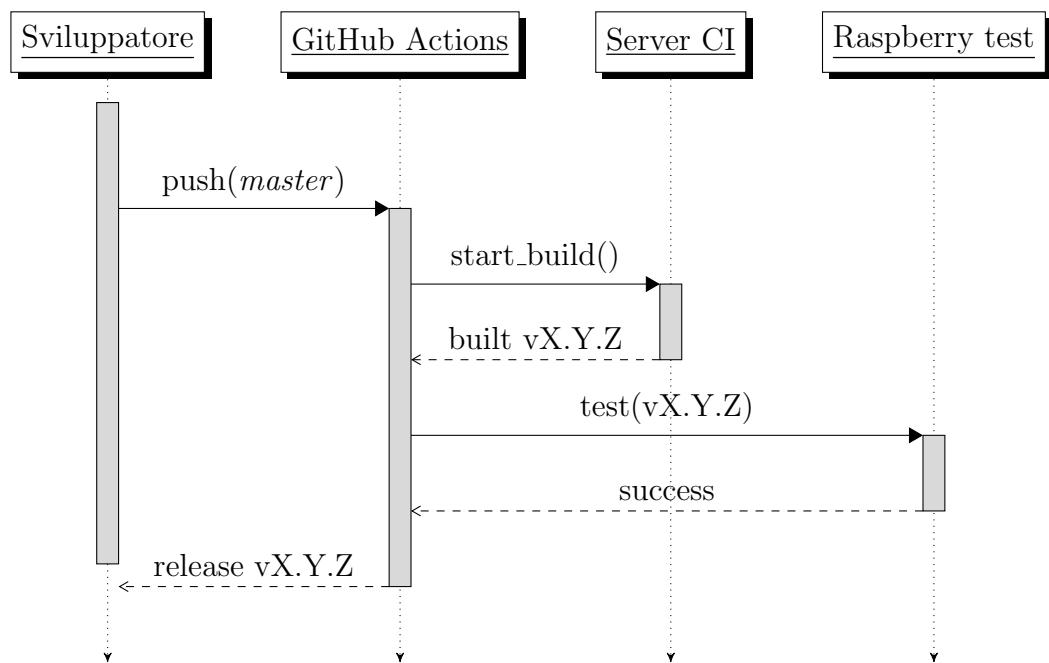


Figura 4.3: infrastruttura CI

# 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. Questo è solo un inizio, in quanto gli scenari attualmente validati sono pochi proprio perché altrimenti il tempo necessario per eseguirli diventerebbe eccessivo, vincolo che viene completamente rimosso eseguendoli in automatico.

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

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

#### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.1.

<b>azione</b>	<b>risultato atteso</b>
prendere un dispositivo in stato <i>non abbinato</i>	
connettersi alla rete Wi-Fi del RE in test	la connessione ha successo. È possibile raggiungere il radiatore all'indirizzo locale 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> .

Tabella 5.1: test abbinamento del dispositivo

1. connettiti all'AP del RE
2. chiama l'API di stato del RE
  - **verifica:** la versione del firmware indicata è compatibile con il firmware che si sta testando
3. chiama l'API per la scansione delle reti Wi-Fi del RE
  - **verifica:** la scansione delle reti non restituisce un risultato vuoto
4. chiama l'API per l'abbinamento del RE
  - **verifica:** la risposta presenta uno stato di successo
5. commuta la modalità Wi-Fi del Raspberry in AP
6. attendi una messaggio su cloud dal RE
  - **verifica:** la richiesta presenta un'azione di tipo *GET*
7. rispondi alla richiesta *GET* con uno stato di *rejected*, dato che il dispositivo era precedentemente resettato e non vi erano dati in cloud
8. attenti un messaggio su cloud dal RE
  - **verifica:** il metodo indicato nella richiesta è *REPORTED\_UPDATE*
  - **verifica:** la versione indicata nella richiesta è 0
  - **verifica:** l'*envId* indicato nella richiesta corrisponde con quello indicato nella richiesta pairing inviata al punto 4.

### 5.1.2 Downgrade 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.

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

Tabella 5.2: test di downgrade locale del dispositivo

### Caso di test

#### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.2.

1. connettiti all'AP del RE
2. chiama l'API per l'aggiornamento locale del firmware
  - **verifica:** l'API risponde con stato HTTP 200
3. attendi che il RE si riavvii
4. riconnettiti all'AP del RE
5. chiama l'API di stato del RE
  - **verifica:** la versione del firmware restituita dall'API è identica a quella inviata al RE

#### 5.1.3 Aggiornamento OTA cloud

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

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.

Tabella 5.3: test di aggiornamento OTA

## Caso di test

### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.3.

1. connettiti all'AP del RE
2. chiama l'API per l'abbinamento del RE
  - **verifica:** la risposta presenta uno stato di successo
3. commuta la modalità Wi-Fi del Raspberry in AP
4. attendi un messaggio da parte del RE
  - **verifica:** il messaggio indica l'azione *GET*
5. rispondi alla richiesta *GET accepted* con stato *MANUALE*
6. attendi un secondo
  - **verifica:** il LED di stato del RE è spento
7. invia un aggiornamento OTA al RE mediante il cloud con versione precedente da quella attualmente in esecuzione

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>

Tabella 5.4: test ripristino di fabbrica

8. attenti 60 secondi per consentire al RE di completare l'aggiornamento

- **verifica:** lo stato del Job IoT Core è *SUCCESSO*

9. attendi un messaggio su cloud

- **verifica:** il messaggio ricevuto è di tipo *REPORTED\_UPDATE*
- **verifica:** la versione firmware indicata nella risposta è identica alla versione del firmware inviato al punto punto 7

### 5.1.4 Ripristino di fabbrica

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

#### Caso di test

#### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.4.

1. connettiti all'AP del RE

2. chiama l'API per l'abbinamento del RE
  - **verifica:** la risposta presenta uno stato di successo
3. commuta la modalità Wi-Fi del Raspberry in AP
4. attendi un messaggio da parte del RE
  - **verifica:** il messaggio indica l'azione *GET*
5. rispondi alla richiesta *GET accepted* con stato *MANUALE*
6. attendi un secondo
  - **verifica:** il LED di stato del RE è spento
7. esegui la procedura *reset di fabbrica*
8. attendi un messaggio su cloud
  - **verifica:** il metodo indicato è *DELETE*
9. attendi un secondo affinché il RE si possa riavviare
  - **verifica:** i LED di stato del RE sono di colore rosso

### 5.1.5 Ripristino di fabbrica disconnesso

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

#### Caso di test

#### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.5.

1. connettiti all'AP del RE

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

Tabella 5.5: test di ripristino di fabbrica con dispositivo disconnesso

2. chiama l'API per l'abbinamento del RE
  - **verifica:** la risposta presenta uno stato di successo
3. commuta la modalità Wi-Fi del Raspberry in AP
4. attendi un messaggio da parte del RE
  - **verifica:** il messaggio indica l'azione *GET*
5. rispondi alla richiesta *GET accepted* con stato *MANUALE*
6. attendi un secondo
  - **verifica:** il LED di stato del RE è spento
7. esegui la procedura *reset di fabbrica*
8. attendi un secondo affinché il RE si possa riavviare
  - **verifica:** i LED di stato del RE sono di colore rosso

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>

Tabella 5.6: test di termoregolazione base

### 5.1.6 Termoregolazione base

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

#### Caso di test

#### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.6.

1. connettiti all'AP del RE
2. chiama l'API per l'abbinamento del RE
  - **verifica:** la risposta presenta uno stato di successo
3. commuta la modalità Wi-Fi del Raspberry in AP
4. attendi un messaggio da parte del RE
  - **verifica:** il messaggio indica l'azione *GET*

5. rispondi alla richiesta *GET accepted* con stato *MANUALE*
6. attendi 1 secondo
  - **verifica:** il LED di stato del RE è spento
  - **verifica:** il carico del *re* è spento
7. premi il pulsante + per 20 volte
8. attendi un secondo
  - **verifica:** il carico del radiatore (resistenza elettrica) è acceso
9. attendi un messaggio su cloud
  - **verifica:** il messaggio è di tipo *REPORTED\_UPDATE*
  - **verifica:** il campo *heatingStatus* riporta lo stato di chiamata
10. premi il pulsante - per 20 volte
11. attendi un secondo
  - **verifica:** il carico del radiatore (resistenza elettrica) è spento
  - **verifica:** il messaggio è di tipo *REPORTED\_UPDATE*
  - **verifica:** il campo *heatingStatus* riporta uno stato di non chiamata

### 5.1.7 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).

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

Tabella 5.7: test modalità standb-by

### Caso di test

#### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.7.

1. connettiti all'AP del RE
2. chiama l'API per l'abbinamento del RE
  - **verifica:** la risposta presenta uno stato di successo
3. commuta la modalità Wi-Fi del Raspberry in AP
4. attendi un messaggio da parte del RE
  - **verifica:** il messaggio indica l'azione *GET*
5. rispondi alla richiesta *GET accepted* con stato *MANUALE*
6. attendi un secondo
  - **verifica:** il LED di stato del RE è spento

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

Tabella 5.8: test funzionamento disconnesso

- **verifica:** il carico del RE è attivo

7. tieni premuto per 6 secondi il pulsante -

- il colore dei led di stato è magenta
- il carico del RE è spento

8. attendi un secondo

9. tieni premuto per 6 secondi il pulsante -

- il colore dei led di stato è diverso da magenta
- il carico del RE è acceso

### 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

#### Implementazione

Vediamo l'implementazione del test realizzata ponendoci dal punto di vista del sistema di test. È possibile vedere il codice Python in sezione A.8.

1. connettiti all'AP del RE
2. chiama l'API per l'abbinamento del RE
  - **verifica:** la risposta presenta uno stato di successo
3. commuta la modalità Wi-Fi del Raspberry in AP
4. attendi un messaggio da parte del RE
  - **verifica:** il messaggio indica l'azione *GET*
5. rispondi alla richiesta *GET accepted* con stato *MANUALE*
6. attendi un secondo
  - **verifica:** il LED di stato del RE è spento
7. disabilita l'AP del raspberry
8. riavvia il RE
9. attendi 30 secondi
  - **verifica:** il LED di stato del RE è giallo
10. attiva l'AP del RE
11. attendi un messaggio su cloud
  - **verifica:** il messaggio presenta un'azione *GET*

## 5.2 Prestazioni

Analizziamo le prestazioni di questo approccio. Per farlo possiamo confrontare il tempo di esecuzione dei test effettuati dallo strumento automatico con i tempi di esecuzione ottenuti con l'approccio manuale.

Per misurare i tempi di esecuzione manuale ho cronometrato l'esecuzione dei test, considerando che per ogni test l'esecutore deve:

- predisporre il dispositivo alle condizioni iniziali prima del test
- leggere la descrizione di ogni passo da eseguire
- eseguire il passaggio indicato nella tabella
- verificare attentamente che l'esito effettivo sia identico a quello atteso
- aggiornare il documento con l'esito del test di successo/fallimento. In caso di fallimento dettagliare l'esito che realmente il test ha avuto.

Vi è anche una fase iniziale di setup, dove l'operatore deve predisporre tutto il materiale necessario, prendere il dispositivo, avviare sul PC i tool necessari (client HTTP REST, client MQTT), aprire il documento di test.

Per la misurazione dei tempi automatizzata invece ho lanciato prima i test singolarmente, dopo di che ho lanciato la test suite completa, facendola eseguire in automatico su tutti i test. Il tempo di setup indicato per il test automatizzato è il tempo necessario ad inizializzare il programma di test.

In Tabella 5.9 i risultati dei testi effettuati.

Come si può vedere per ogni test il tempo di esecuzione manuale è circa 4 volte superiore a quello di esecuzione dato dal tool automatico. Questo è comprensibile dal fatto che il tool riduce al massimo i tempi morti, andando ad eseguire i controlli in maniera automatizzata.

Vi è da considerare anche che la misurazione dei tempi manuali è stata effettuata tenendo conto di quanto ci impiega una persona che abitualmente lavora sul progetto ad effettuarli. Infatti prima di diventare produttivi è

---

<sup>1</sup>Tempo di esecuzione lanciando assieme tutti i test, potrebbe non corrispondere esattamente alla somma dei tempi di esecuzione dei singoli test

Test	Automatico	Manuale
<b>Tempo di setup</b>	0m 5s	circa 10 minuti
Abbinamento del dispositivo (5.1.1)	0m 18s	3m 56s
Downgrade locale (5.1.2)	0m 28s	3m 47s
Aggiornamento OTA cloud (5.1.3)	0m 48s	5m 12s
Ripristino di fabbrica (5.1.4)	0m 27s	3m 34s
Ripristino di fabbrica disconnesso (5.1.5)	0m 32s	3m 50s
Termoregolazione base (5.1.6)	0m 41s	3m 58s
Modalità stand-by (5.1.7)	0m 40s	4m 9s
Funzionamento disconnesso (5.1.8)	1m 20s	5m 30s
<b>Totale</b>	4m 52s <sup>1</sup>	33m 56s

Tabella 5.9: confronto fra tempi di esecuzione

necessario conoscere il contesto e gli strumenti, e questo richiede un tempo di circa un'ora.

Ma il risultato più importante non è solo il tempo, ma la precisione dei risultati ottenuti. I test automatici sono riproducibili, e si è sicuri che ogni volta vengono eseguiti nelle medesime condizioni. Al contrario il test manuale è meno affidabile e potrebbe non andare ad identificare tutti i problemi possibili, perché l'operatore dimentica un controllo.

Infine durante la validazione della procedura di test ho riscontrato un bug all'interno del firmware. Infatti ho provato a lanciarla con un firmware che non è ancora stato rilasciato. La procedura mi ha permesso di individuare un problema nella gestione dei LED di stato che non si comportavano come da specifica, rimanendo accesi in certe casistiche anche quando in realtà dovevano essere spenti.



# Conclusioni

Abbiamo visto come l’automatizzazione dei test di accettazione può essere implementata ed utilizzata anche in un progetto embedded andando ad evitare danni (che sono sia economici sia d’immagine) 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 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 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 la Connection Unit (CU), l'unità centrale di controllo (gateway) che coordina fra di loro tutti i dispositivi radio, tramite un protocollo RF su banda 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 Radio Frequenza (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 Yet Another Thermostat (YAT), 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 multizone, in cui vi è uno YAT *master* (alimentato dalla rete) che mantiene la connessione verso il cloud e la comunicazione con gli altri YAT *slave*, che possono essere alimentati a batteria in quanto non necessitano di una costante connessione con la rete.

Altra caratteristica dello YAT 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. Consegue 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.



# Appendice A

## Implementazione casi di test

### A.1 Test pairing

```
1 from time import time, sleep
2 from logging import getLogger
3
4 import uuid
5
6 from fw_test.context import Context
7 from fw_test.wifi import ApConfiguration, WifiSecurityType
8 from fw_test.cloud import Message, Action, Response, PacketType
9 from fw_test.firmware import FirmwareVersion
10
11 LOGGER = getLogger(__name__)
12
13 TEST_SSID = "TEST-NETWORK"
14 TEST_PASSPHRASE = "test-network-passphrase"
15
16
17 def test_pairing(ctx: Context):
18     # avvio la connessione del Raspberry all'AP
19     ctx.wifi.client_connect()
20
21     sleep(1)
22
23     # chiedo lo stato al dispositivo
```

```

24     response = ctx.api.status()
25
26     # mi assicuro che la versione firmware del dispositivo sia quella da testare
27     assert FirmwareVersion.from_str(response["system"]["fwVer"]) == ctx.firmware.version
28
29     # effettuo la scansione Wi-Fi
30     response = ctx.api.wifi_scan()
31
32     # mi assicuro che la scan restituiscia almeno una rete
33     assert len(response) > 0
34
35     # invio la richiesta provision
36     ap_config = ApConfiguration(
37         ssid=TEST_SSID,
38         passphrase=TEST_PASSPHRASE,
39         security_type=WifiSecurityType.WPA2,
40         channel=6,
41     )
42     env_id = uuid.uuid4()
43     response = ctx.api.provision(ap_config, env_id)
44     assert response["status"] == "success"
45
46     # communto la raspberry in AP mode
47     ctx.wifi.start_ap(ap_config)
48
49     # attendo il primo messaggio su cloud
50     msg = ctx.cloud.receive(timeout=60)
51     assert msg.action == Action.GET
52
53     # invio una GET rejected al dispositivo device
54     ctx.cloud.publish(Message(
55         action=Action.GET,
56         response=Response.REJECTED,
57         state={
58             "clientToken": msg.state["clientToken"],
59             "timestamp": int(time()),
60             "version": 0,
61             "type": PacketType.HEADER,
62         }
63     ))
64

```

```

65     # mi aspetto ora un reported update
66     msg = ctx.cloud.receive(timeout=5)
67     assert msg.action == Action.REPORTED_UPDATE
68
69     # la versione iniziale deve essere zero
70     assert msg.state["version"] == 0
71
72     # il system id deve essere uguale all'env id
73     assert msg.state["envId"] == env_id.bytes
74
75     # la versione firmware deve corrispondere a quella in test
76     assert msg.state["firmwareVersion"][0] == ctx.firmware.version.major
77     assert msg.state["firmwareVersion"][1] == ctx.firmware.version.minor

```

---

## A.2 Test downgrade

```

1  from time import sleep
2
3  from fw_test.context import Context
4  from fw_test.firmware import FirmwareVersion
5
6
7  def test_downgrade(ctx: Context):
8      # connette il Raspberry all'AP del radiatore elettrico
9      ctx.wifi.client_connect()
10
11     sleep(2)
12
13     # invio un aggiornamento firmware locale
14     response = ctx.api.firmware_update(ctx.prev_firmware)
15     assert response.status_code == 200
16
17     # attendo che il dispositivo si riavvii
18     sleep(2)
19
20     # mi ricollego al radiatore
21     ctx.wifi.client_connect()
22

```

```

23     sleep(2)

24

25     status = ctx.api.status()

26

27     # controllo che la versione firmware sia quella inviata
28     assert FirmwareVersion.from_str(status['system']['fwVer']) == ctx.prev_firmware.version

```

---

### A.3 Test OTA

```

1  from time import sleep, time
2  import uuid
3
4  from fw_test.context import Context
5  from fw_test.io import LedColor
6  from fw_test.cloud import JobState, Action, Message, Response
7  from fw_test.firmware import FirmwareVersion
8
9  from .utils import TEST_AP_CONFIG, STATE_MANUAL
10
11
12 def test_ota(ctx: Context):
13     # avvio la connessione del Raspberry all'AP
14     ctx.wifi.client_connect()
15
16     sleep(1)
17
18     # invio la richiesta provision
19
20     env_id = uuid.uuid4()
21     response = ctx.api.provision(TEST_AP_CONFIG, env_id)
22     assert response["status"] == "success"
23
24     # communto la raspberry in AP mode
25     ctx.wifi.start_ap(TEST_AP_CONFIG)
26
27     # attendo che il pairing abbia successo
28     msg = ctx.cloud.receive(timeout=30)
29     assert msg.action == Action.GET

```

```

30
31     ctx.cloud.publish(Message(
32         action=Action.GET,
33         response=Response.ACCEPTED,
34         state={
35             **STATE_MANUAL,
36             "clientToken": msg.state["clientToken"],
37             "timestamp": int(time()),
38             "envId": env_id.bytes,
39         }
40     ))
41     sleep(1)
42
43     assert ctx.io.status_led_color() == LedColor.OFF
44
45     job = ctx.cloud.send_ota(ctx.prev_firmware)
46
47     # attendo che il dispositivo esegua il job e si ricolleghi
48     ctx.cloud.receive(timeout=30, filter_action=Action.GET)
49     ctx.cloud.publish(Message(
50         action=Action.GET,
51         response=Response.ACCEPTED,
52         state={
53             **STATE_MANUAL,
54             "clientToken": msg.state["clientToken"],
55             "timestamp": int(time()),
56             "envId": env_id.bytes,
57         }
58     ))
59
60     # su cloud arriva la nuova versione
61     msg = ctx.cloud.receive(timeout=30)
62     assert msg.action == Action.REPORTED_UPDATE
63
64     # la versione firmware deve corrispondere a quella in test
65     assert msg.state["firmwareVersion"][0] == ctx.prev_firmware.version.major
66     assert msg.state["firmwareVersion"][1] == ctx.prev_firmware.version.minor
67
68     # il job ha avuto successo
69     assert ctx.cloud.job_state(job) == JobState.SUCCEEDED

```

---

## A.4 Test ripristino di fabbrica

---

```

1  from time import sleep, time
2
3  import uuid
4
5  from fw_test.context import Context
6  from fw_test.io import LedColor
7  from fw_test.cloud import Action, Response, Message
8
9  from .utils import TEST_AP_CONFIG, STATE_MANUAL
10
11 def test_factory_reset(ctx: Context):
12
13     # avvio la connessione del Raspberry all'AP
14     ctx.wifi.client_connect()
15
16     sleep(1)
17
18     # invio la richiesta provision
19     env_id = uuid.uuid4()
20     response = ctx.api.provision(TEST_AP_CONFIG, env_id)
21     assert response["status"] == "success"
22
23     # communto la raspberry in AP mode
24     ctx.wifi.start_ap(TEST_AP_CONFIG)
25
26     # attendo che il pairing abbia successo
27     msg = ctx.cloud.receive(timeout=30)
28     assert msg.action == Action.GET
29
30     ctx.cloud.publish(Message(
31         action=Action.GET,
32         response=Response.ACCEPTED,
33         state={
34             **STATE_MANUAL,
35             "clientToken": msg.state["clientToken"],
36             "timestamp": int(time()),
37             "envId": env_id.bytes,
38         }

```

```

39     ))
40
41     msg = ctx.cloud.receive(timeout=30)
42     assert msg.action == Action.REPORTED_UPDATE
43
44     assert ctx.io.status_led_color() == LedColor.OFF
45
46     # ora posso fare l'hard reset
47     ctx.io.hard_reset()
48
49     # attendo che arrivi su cluod una REPORTED UPDATE con envId
50     # vuoto (rimozione abbinamento)
51     msg = ctx.cloud.receive(timeout=5)
52     assert msg.action == Action.REPORTED_UPDATE
53     assert msg.state["envId"] == b"\0" * 16
54
55     sleep(1)
56
57     # il LED indica dispositivo resettato
58     assert ctx.io.status_led_color() == LedColor.RED
59

```

---

## A.5 Test ripristino di fabbrica disconnesso

```

1  from time import sleep, time
2
3  import uuid
4
5  from fw_test.context import Context
6  from fw_test.io import LedColor
7  from fw_test.cloud import Action, Message, Response
8
9  from .utils import TEST_AP_CONFIG, STATE_MANUAL
10
11 def test_factory_reset_offline(ctx: Context):
12
13     # avvio la connessione del Raspberry all'AP
14     ctx.wifi.client_connect()
15

```

```

16     sleep(1)

17

18     # invio la richiesta provision
19     env_id = uuid.uuid4()
20     response = ctx.api.provision(TEST_AP_CONFIG, env_id)
21     assert response["status"] == "success"

22

23     # communto la raspberry in AP mode
24     ctx.wifi.start_ap(TEST_AP_CONFIG)

25

26     # attendo che il pairing abbia successo
27     msg = ctx.cloud.receive(timeout=30)
28     assert msg.action == Action.GET

29

30     ctx.cloud.publish(Message(
31         action=Action.GET,
32         response=Response.ACCEPTED,
33         state={
34             **STATE_MANUAL,
35             "clientToken": msg.state["clientToken"],
36             "timestamp": int(time()),
37             "envId": env_id.bytes,
38         }
39     ))
40

41     sleep(1)

42

43     assert ctx.io.status_led_color() == LedColor.OFF

44

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

47

48     # attendo 5 secondi
49     sleep(5)

50

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

53

54     sleep(1)

55

56     # il LED indica dispositivo resettato

```

```
57     assert ctx.io.status_led_color() == LedColor.RED  
58
```

---

## A.6 Test termoregolazione

```
1  from time import sleep, time  
2  import uuid  
3  
4  from fw_test.context import Context  
5  from fw_test.io import LedColor  
6  from fw_test.cloud import Message, Response, Action  
7  from fw_test.cloud.state import SYSTEM_STATUS_HEATING, SYSTEM_STATUS_LOAD_ACTIVE  
8  
9  from .utils import STATE_MANUAL, TEST_AP_CONFIG  
10  
11  
12 START_SET_POINT = 60  
13 SET_POINT_INCREMENT = 20  
14  
15 def test_thermoregulation(ctx: Context):  
16     # avvio la connessione del Raspberry all'AP  
17     ctx.wifi.client_connect()  
18  
19     sleep(1)  
20  
21     # invio la richiesta provision  
22     env_id = uuid.uuid4()  
23     response = ctx.api.provision(TEST_AP_CONFIG, env_id)  
24     assert response["status"] == "success"  
25  
26     # communto la raspberry in AP mode  
27     ctx.wifi.start_ap(TEST_AP_CONFIG)  
28  
29     msg = ctx.cloud.receive(timeout=30)  
30     assert msg.action == Action.GET  
31  
32     ctx.cloud.publish(Message(  
33         action=Action.GET,  
34         response=Response.ACCEPTED,
```

```

35         state={
36             **STATE_MANUAL,
37             "clientToken": msg.state["clientToken"],
38             "timestamp": int(time()),
39             "envId": env_id.bytes,
40             "manualSetPoint": START_SET_POINT
41         }
42     ))
43
44     sleep(1)
45
46     assert ctx.io.status_led_color() == LedColor.OFF
47     assert not ctx.io.is_load_active()
48
49     ctx.cloud.flush()
50
51     for _ in range(SET_POINT_INCREMENT + 1):
52         ctx.io.press_plus()
53         sleep(0.1)
54
55     # i LED sono rossi
56     assert ctx.io.status_led_color() == LedColor.RED
57
58     # il radiatore deve scalare ora
59     assert ctx.io.is_load_active()
60
61     msg = ctx.cloud.receive(timeout=5)
62     assert msg.action == Action.REPORTED_UPDATE
63     assert msg.state["manualSetPoint"] == START_SET_POINT + SET_POINT_INCREMENT * 5
64     assert msg.state["systemStatus"] & SYSTEM_STATUS_HEATING != 0
65     assert msg.state["systemStatus"] & SYSTEM_STATUS_LOAD_ACTIVE != 0
66
67     # attendo che la tastiera si spenga
68     sleep(5)
69     assert ctx.io.status_led_color() == LedColor.OFF
70
71     for _ in range(SET_POINT_INCREMENT + 1):
72         ctx.io.press_minus()
73         sleep(0.1)
74
75     # i LED sono azzurri

```

```

76     assert ctx.io.status_led_color() == LedColor.CYAN
77
78     # il radiatore non deve più scaldare
79     assert not ctx.io.is_load_active()
80
81     msg = ctx.cloud.receive(timeout=5)
82     assert msg.action == Action.REPORTED_UPDATE
83     assert msg.state["manualSetPoint"] == START_SET_POINT
84     assert msg.state["systemStatus"] & SYSTEM_STATUS_HEATING == 0
85     assert msg.state["systemStatus"] & SYSTEM_STATUS_LOAD_ACTIVE == 0
86

```

---

## A.7 Test standby

```

1  from time import sleep, time
2  import uuid
3
4  from fw_test.context import Context
5  from fw_test.io import LedColor
6  from fw_test.cloud import Message, Response, Action, PacketType
7
8  from .utils import STATE_MANUAL, TEST_AP_CONFIG
9
10
11 def test_standby(ctx: Context):
12     # avvio la connessione del Raspberry all'AP
13     ctx.wifi.client_connect()
14
15     sleep(1)
16
17     # invio la richiesta provision
18     env_id = uuid.uuid4()
19     response = ctx.api.provision(TEST_AP_CONFIG, env_id)
20     assert response["status"] == "success"
21
22     # communto la raspberry in AP mode
23     ctx.wifi.start_ap(TEST_AP_CONFIG)
24
25     # attendo che il pairing abbia successo

```

```

26     msg = ctx.cloud.receive(timeout=60)
27     assert msg.action == Action.GET
28
29     # invio una GET accepted al dispositivo device
30     ctx.cloud.publish(Message(
31         action=Action.GET,
32         response=Response.ACCEPTED,
33         state={
34             **STATE_MANUAL,
35             "clientToken": 0,
36             "timestamp": int(time()),
37             "envId": env_id.bytes,
38             "version": 1,
39             "manualSetPoint": 150,
40         }
41     ))
42
43     sleep(1)
44
45     # I led devono essere spenti
46     assert ctx.io.status_led_color() == LedColor.OFF
47     assert ctx.io.is_load_active()
48
49     ctx.io.press_minus(6)
50
51     assert ctx.io.status_led_color() == LedColor.MAGENTA
52     assert not ctx.io.is_load_active()
53
54     sleep(1)
55
56     ctx.io.press_minus(6)
57
58     assert ctx.io.status_led_color() != LedColor.MAGENTA
59     assert ctx.io.is_load_active()
60

```

---

## A.8 Test funzionamento offline

---

```
1 from time import sleep, time
2
3 import uuid
4
5 from fw_test.context import Context
6 from fw_test.io import LedColor
7 from fw_test.cloud import Action, Message, Response
8
9 from .utils import TEST_AP_CONFIG, STATE_MANUAL
10
11 def test_offline_working(ctx: Context):
12     # avvio la connessione del Raspberry all'AP
13     ctx.wifi.client_connect()
14
15     sleep(1)
16
17     # invio la richiesta provision
18     env_id = uuid.uuid4()
19     response = ctx.api.provision(TEST_AP_CONFIG, env_id)
20     assert response["status"] == "success"
21
22     # communto la raspberry in AP mode
23     ctx.wifi.start_ap(TEST_AP_CONFIG)
24
25     # attendo che il pairing abbia successo
26     msg = ctx.cloud.receive(timeout=30)
27     assert msg.action == Action.GET
28
29     ctx.cloud.publish(Message(
30         action=Action.GET,
31         response=Response.ACCEPTED,
32         state={
33             **STATE_MANUAL,
34             "clientToken": msg.state["clientToken"],
35             "timestamp": int(time()),
36             "envId": env_id.bytes,
37         }
38     ))
```

```
39
40     sleep(1)
41
42     assert ctx.io.status_led_color() == LedColor.OFF
43
44     # ora stoppo l'access-point
45     ctx.wifi.stop_ap()
46
47     # riavvio il dispositivo
48     ctx.io.reset()
49
50     sleep(20)
51
52     # il LED indica dispositivo resettato
53     assert ctx.io.status_led_color() == LedColor.YELLOW
54
55     ctx.cloud.flush()
56     ctx.wifi.start_ap(TEST_AP_CONFIG)
57
58     # quando il RE si riconnette deve fare subito una GET
59     msg = ctx.cloud.receive(timeout=5)
60     assert msg.action == Action.GET
61
```

---

# 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, 23, 26, 33, 66, 67

**ARM** Architettura hardware RISC sviluppata dall’azienda ARM. La sua versione Core M è particolarmente utilizzata all’interno di processore embedded low-power.. 19, 21

**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. 27, 42, 43, 86

**cloud** Insieme di tecnologie che permettono di archiviare ed elaborare i dati in rete, anziché su un dispositivo locale. 1, 26, 27, 29, 34, 35, 53, 68, 87

**consumatore** L’utente finale del sistema. È suo obbligo utilizzare il prodotto per gli scopi e nei modi dichiarati dal produttore. 3

**Fil Pilote** Standard francese per il comando di radiatori elettrici in maniera centralizzata. Consiste in un secondo cavo in cui in base alla modalità arriva un segnale a 230V AC che codifica 4 modalità di funzionamento: COMFORT (assenza di segnale), ECO (segnale 230AC comple-

**to), OFF (solo componente positiva della sinusoide 230V), ANTIGELO (solo componente negativa della sinusoide 230V).** 19, 25

**firmware** La componente software che viene eseguita su un dispositivo embedded. 3, 4, 14, 15, 19–22, 30, 33, 35, 37, 41, 47, 49–54, 67–69

**gateway** Dispositivo hardware che mette in comunicazione due reti di tipologie differenti, ad esempio una rete TCP/IP e dispositivi RF. 67

**Git** Sistema di controllo versione 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. 46

**i2c** Protocollo di comunicazione seriale su due segnali elettrici (SDA ed SCL) per la comunicazione fra chip embedded, quali microcontrollori e sensori. 20

**IoT Core** Servizio serverless di broker MQTT gestito da AWS. 23, 27–29, 35, 42, 43, 86

**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). . 41, 42, 44, 52–54

**microcontrollore** Componente elettronico che integra in un unico chip processore, memoria RAM, memoria flash, gestione delle periferiche di input/output GPIO. Tipicamente offre prestazioni molto limitate ma bassi costi e bassi consumi energetici. 10, 19, 22

**Modbus** Protocollo di comunicazione master/slave fra dispositivi embedded. Può funzionare sia su interfaccia fisica seriale UART che su stack di rete TCP/IP. 68

**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. 23, 27, 35, 41–43, 53, 85, 87

**OpenTherm** Protocollo standard definito dal consorzio OpenTherm per la comunicazione fra termostati e generatori di calore (principalmente caldaie). È un protocollo master/slave dove il master è il termostato ambiente (o una centralina di controllo) e lo slave il generatore di calore. L'interfaccia fisica è un bus composto da due cavi che è in grado oltre a portare i dati di fornire alimentazione al dispositivo master (termostato).. 68

**produttore** Colui che pone il proprio marchio sul prodotto. È suo obbligo garantire le qualità del prodotto dichiarate, e risponde al consumatore in caso di non conformità o danni. 5, 13

**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, 26, 86

**topic MQTT** Stringa testuale che identifica un canale di comunicazione sul quale un client MQTT può inviare o ricevere messaggi. 27, 29, 43, 44, 85

**Wi-Fi** Protocollo radio che consente la comunicazione su rete TCP/IP senza cavi. 2, 7, 18–24, 28, 31–33, 37, 39, 41, 44, 45, 50, 51, 53, 55–57, 59–61, 66, 68



# Acronimi

**ADC** Analog to Digital Converter. 20, 21

**AP** Access Point. 20, 21, 24, 33, 37, 39, 41, 44, 45, 51–57, 59, 61

**AWS** Amazon Web Services. 23, 24, 26–28, 35, 42–44, 86

**BLE** Bluetooth Low Energy. 33, 66, 68

**CA** Certification Authority. 27

**CI** Continuos Integration. 42, 46, 47

**CU** Connection Unit. 67

**GPIO** General Purpose Input Output. 20, 21, 37, 39–42, 86

**HMI** Human Machine Interface. 23, 24, 34, 69

**HTTP** Hyper Text Transfer Protocol. 21, 23, 24, 45

**IoT** Internet of Things. 1, 87

**JSON** JavaScript Object Notation. 28, 44, 50, 86

**LED** Light Emitting Diode. 2, 12, 18, 19, 23, 25, 32, 34, 41, 53–61

**NCM** Network Connection Manager. 23, 24

**NTC** Negative Temperature Coefficient. 18, 19, 23, 24

**OTA** Over The Air update. 10, 13, 21, 22, 33, 41, 43, 52, 53

**PWM** Pulse Width Modulation. 20

**RE** Radiatore Elettrico. 2, 17, 18, 29, 33–40, 43, 44, 47, 50–61, 65

**RED** Radio Equipment Device. 7

**RF** Radio Frequenza. 67, 68, 86

**RGBW** Red Green Blue White. 18, 19, 23, 59

**RTC** Real Time Clock. 21

**RTOS** Real Time Operating System. 10, 21

**SDK** Software Development Kit. 21

**SIL** System Integrity Level. 8

**SNTP** Simple Network Time Protocol. 21

**SPI** Serial Peripheral Interface. 20, 69

**STA** Station. 20, 21, 24, 39, 41, 44

**TLS** Transport Layer Security. 21, 27, 43

**UART** Universal Asynchronous Receive and Transmit. 20, 22, 39, 41, 42, 68, 86

**USB** Universal Serial Bus. 39, 40

**VMC** Ventilazione Meccanica Controllata. 1, 33, 85

**YAT** Yet Another Thermostat. 68, 69