Università Politecnica delle Marche

Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Progettazione e implementazione di un gateway per la comunicazione tra dispositivi multiprotocollo in un contesto IoT: il sottosistema per la traduzione dei protocolli e la comunicazione dei dispositivi

Design and implementation of a gateway for the communication of multiprotocol devices in an IoT context: the subsystem for protocol translation and device communication

Relatore

Prof. Domenico Ursino

Candidato

Andrea Perrello

Anno Accademico 2019-2020

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Introduzione

L'Internet of Things è un settore ancora in forte espansione, al quale è rivolto grande interesse tecnologico nel panorama attuale. Sotto certi aspetti, le sfide moderne in cui è coinvolto l'IoT riguardano ancora gli aspetti fondamentali sull'interconnessione tra oggetti. Infatti, benché gli aspetti legati alle "things" siano pressoché ormai consolidati e i dispositivi siano ampiamente noti e diffusi nel mercato, irrisolte e prioritarie rimangono, invece, le problematiche legate alla questione di "internet". Attualmente, i problemi tecnologici non riguardano più l'abilitazione dei dispositivi alla comunicazione, bensì le modalità in cui la comunicazione si attua e i processi che permettono di renderla agevole, fruibile e sicura.

Questi aspetti riflettono l'andamento delle problematiche tecnologiche attuali, secondo le quali il quesito prioritario non risulta più essere la reperibilità dei dati, ma il loro uso intelligente, affidabile e di valore. In questo modo, si evidenzia la necessità di fare luce su un futuro nel quale la possibilità di operare in un ambiente IoT consolidato sarà un bisogno sempre più stringente, e nel quale la comunicazione non debba più essere il problema, ma l'affidabile strumento che lascerà spazio alla gestione delle vere sfide industriali e commerciali, piuttosto che alla definizione dei mezzi per risolverle.

La questione affrontata dalla presente tesi affonda le sue radici nella crescita sempre maggiore dei mezzi e delle tecnologie che abilitano la comunicazione degli oggetti nel mondo IoT. Oltre alla disponibilità sempre maggiore di oggetti intelligenti, accompagnata da un conseguente aumento dei protocolli che ne abilitano la trasmissione delle informazioni, il processo di digitalizzazione globale sta vedendo crescere anche il bisogno di adeguare i vecchi sistemi informatici (*legacy*) alla trasformazione tecnologica in atto, appoggiandosi su tecnologie e standard di comunicazione che li rendano intelligenti e, quindi, utilizzabili in contesti moderni. Di fatto, quella che è una tendenza necessaria sta, però, alimentando l'incremento di "lingue" con le quali i dispositivi comunicano. Persino l'obiettivo di unificare i vari protocolli esistenti, che numerosi standard proposti hanno tentato di raggiungere, si è spesso risolto nella creazione di ulteriori protocolli andati ad aggiungersi a quelli già esistenti.

La presente tesi intende perseguire lo stesso obiettivo, ma senza proporre un nuovo standard o una piattaforma dotata di sistemi stretti di comunicazione e di sviluppo. L'idea è quella di affrontare l'uniformazione in ambito IoT attraverso

l'astrazione e la sintesi di concetti propri dell'informatica e della comunicazione, approfondendo i temi della traduzione dei protocolli e della normalizzazione dei dati come tecniche che permettano di ottenere un linguaggio universale, necessario per raggiungere la vera trasformazione digitale. La trasparenza tecnologica, infatti, si pone come abilitatore verso scenari di fruizione agevole dei dati, assorbendo i problemi della gestione degli stessi e dell'interfacciamento tra i dispositivi e l'utente finale.

Il lavoro svolto in questa trattazione è parte dello sviluppo di un progetto di più ampio respiro che riguarda la definizione di un gateway multiprotocollo, i cui componenti principali sono sommariamente riportati per fornire una visione d'insieme, per poi focalizzarsi sugli aspetti legati alla traduzione e la comunicazione. Il sistema si pone come soluzione al problema della trasparenza tecnologica, al fine di uniformare diverse sorgenti di dati e popolare agevolmente diverse destinazioni per la loro memorizzazione, elaborazione e visualizzazione.

La tesi è suddivisa in otto capitoli, strutturati come di seguito specificato:

- Nel primo capitolo si fornisce una panoramica più approfondità sul mondo dell'Internet of Things. In particolare, vengono esaminati le sue caratteristiche fondamentali, le principali tecnologie abilitanti e la struttura comune ad ogni sistema IoT.
- Nel secondo capitolo si fornisce una descrizione dettagliata ad alto livello dell'architettura del gateway multiprotocollo, riportando informazioni sui concetti alla base del suo design e sulle componenti primarie dalle quali è formato.
- I capitoli dal terzo al sesto coprono le principali fasi dello sviluppo informatico applicate al progetto, ovvero l'analisi dei requisiti, la progettazione, l'implementazione del codice e la sua validazione.
- Il settimo capitolo tratta l'analisi SWOT del sistema sviluppato, mettendo in luce i vari aspetti positivi e negativi e le differenze con sistemi affini già esistenti, al fine di analizzare le possibili strategie di miglioramento.
- L'ultimo capitolo è dedicato alle conclusioni e ai potenziali sviluppi futuri applicabili al gateway multiprotocollo, che permetterebbero di estendere le funzionalità, mitigare i punti critici e migliorare l'esperienza complessiva nell'uso del sistema.

1

Uno sguardo all'Internet of Things

In questo capitolo si fornirà una visione complessiva del mondo dell'Internet of Things. Dopo una breve introduzione sulla sua definizione e sulle sue principali caratteristiche, si mostreranno le sue tipiche applicazioni dal punto di vista funzionale, le tecnologie abilitanti più comuni e le tipiche componenti di un generico sistema IoT. Infine, si fornirà una panoramica sulla trasformazione dell'IoT attraverso l'evoluzione dei suoi paradigmi di elaborazione.

1.1 Caratteristiche e tecnologie abilitanti

Quando si parla di Internet of Things (noto come IoT o *Internet delle Cose*), si parla di sistemi complessi per la produzione, gestione e fruizione dei dati all'interno di una vasta rete di comunicazione. Tale rete, dopo decenni di sviluppo, si è evoluta in un vero e proprio ecosistema in grado di interconnettere hardware, software, oggetti fisici e persone con interazioni complesse e grandi volumi di dati scambiati. A partire dal 1999, anno in cui l'espressione "Internet of Things" è stata coniata per la prima volta dall'ingegnere inglese Kevin Ahston, il numero di utenti, servizi e applicazioni con IoT è cresciuto rapidamente e in molti domini diversi. Lo sviluppo di nuove applicazioni IoT ha permesso di fornire valore in numerosi campi, dalla vita quotidiana personale (domotica e *smart-home*) all'ambiente e alla società (*smart-city* e *smart-building*). A causa delle enormi potenzialità dell'IoT, le richieste di nuovi servizi e applicazioni sono enormi. Numerosi servizi e applicazioni possono essere ulteriormente sviluppati sulla base del gran numero di dispositivi già in grado di comunicare attraverso Internet, anche tramite tecnologie diverse tra loro.

I principi alla base dell'IoT sono stati, infatti, racchiusi all'interno di una serie di caratteristiche ben definite e condivise da tutti i sistemi moderni:

- *Interconnessione*: qualunque piattaforma o dispositivo dotato di interfacce di comunicazione può essere interconnesso all'interno dell'infrastruttura di informazione e comunicazione IoT.
- *Eterogeneità*: i dispositivi nell'IoT sono eterogenei, in quanto basati su diverse piattaforme hardware e reti. Possono interagire con altri dispositivi o piattaforme di servizi attraverso reti diverse.

- *Dinamicità*: lo stato dei dispositivi cambia nel tempo, così come il loro numero, che può variare dinamicamente.
- *Scalabilità*: il numero di dispositivi gestiti e che comunicano con gli altri è almeno di un ordine maggiore rispetto al numero di dispositivi connessi alla rete. In questo senso, la gestione dei dati generati e la loro interpretazione è critica, e necessita di metodi efficienti di gestione dell'informazione e delle strutture dati.
- *Sicurezza*: essendo una connessione di dispositivi in grado di trasmettere informazioni sensibili e protette, l'IoT deve essere approcciato con una progettazione orientata alla sicurezza. Alcuni criteri di sicurezza riguardano la protezione dei dati personali, delle rotte e delle reti di comunicazione.
- *Connettività*: la connettività richiede l'accessibilità delle reti e la compatibilità dei dispositivi in termini di capacità di produrre e consumare dati trasmissibili.

Con l'IoT, la comunicazione via Internet si estende a tutti i dispositivi che ci circondano che siano, per loro natura, in grado di comunicare con l'esterno in modo "intelligente", ovvero capaci di esporre interfacce verso l'ambiente circostante che rispettino ben noti protocolli e tecnologie di trasmissione dell'informazione.

Tuttavia, l'IoT è molto più che comunicazioni tra dispositivi e reti. Le tecnologie abilitanti per l'IoT possono, infatti, essere raggruppate in tre categorie principali:

- *Tecnologie che abilitano i dispositivi all'acquisizione di informazioni*: protocolli di comunicazione e standard, sistemi integrati, servizi di *discovery*[1], reti di dispositivi e reti di reti, tecnologie di telefonia (4G, GPRS, WI-FI, GPS).
- *Tecnologie che abilitano i dispositivi all'elaborazione delle informazioni*: algoritmi di *pre-processing*, tecnologie cloud, super-computer e tecniche di Intelligenza Artificiale.
- *Tecnologie che migliorano la sicurezza e la privacy*: gestione dei dati e delle reti, sistemi di protezione delle informazioni e degli standard GDPR.

Le prime due categorie possono essere comprese congiuntamente, dal momento che le tecnologie che abilitano l'acquisizione sono funzionali a quelle necessarie alla loro elaborazione, ovvero in grado di trasformare semplici "dati" in "informazione", caratteristica fondamentale che differenzia l'IoT dal Internet per come era visto nel passato. La terza categoria è meno funzionale, ma si tratta comunque di un requisito oramai necessario e senza il quale l'interesse nei confronti dell'IoT risulterebbe decisamente meno evidente.

1.2 Applicazioni funzionali

L'IoT vanta un ampio numero di applicazioni in svariati ambiti, sia puramente informatici che a livello di business e fruizione del dato finale per gli utenti. Tali applicazioni (Figura 1.1) possono essere racchiuse in tre principali categorie:

- *Monitoraggio e controllo*: consiste nella raccolta di dati sulle prestazioni di macchinari e dispositivi, sul consumo energetico, sul monitoraggio e il controllo delle

[1] Meccanismo che permette alle applicazioni di accedere ai dati dei dispositivi IoT senza necessità di conoscere la loro effettiva locazione o la struttura dei loro dati.

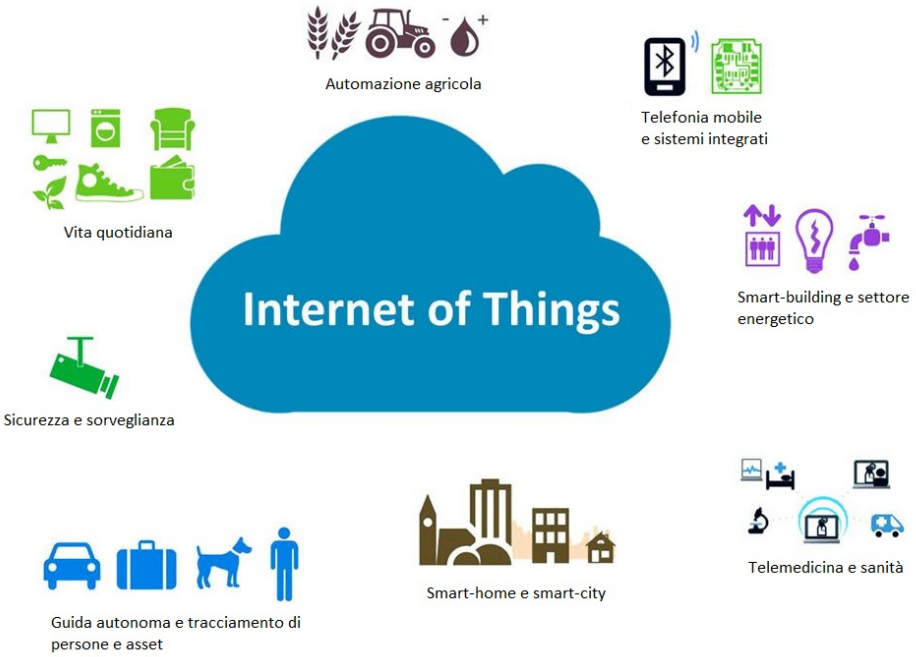

Figura 1.1. Esempio di campi di applicazione dell'IoT

prestazioni in tempo reale. Le applicazioni pratiche legate a questo ambito permettono solitamente un miglioramento dei risultati in un processo produttivo e le ottimizzazioni delle operazioni, portando a costi inferiori e a maggiore produttività. Altri esempi riguardano gli ambiti di domotica intelligente (per il monitoraggio e il controllo remoto della casa) e *smart-driving* (automazione e sicurezza alla guida).

- *Big data e analisi aziendale*: dispositivi e macchine IoT con sensori e attuatori incorporati generano enormi quantità di dati che consentono, attraverso la loro analisi, di prendere decisioni ad alto livello. Tra le varie applicazioni troviamo il supporto al marketing (analisi dei comportamenti dei clienti e del mercato per aumentare la soddisfazione del cliente), il settore pubblico (controllo stradale per la segnalazione di incidenti sulle autostrade, gestione del traffico e fornitura di dati meteorologici) e quello sanitario (controllo della salute attraverso dispositivi indossabili).
- *Condivisione delle informazioni e collaborazione*: questa categoria racchiude ogni altro genere di applicazione funzionale dell'IoT, in quanto riguarda lo scambio di informazioni tra due o più agenti coinvolti nel sistema. La collaborazione può avvenire tra persone ("*human to human*" o *H2H*), tra persone e dispositivi ("*human to machine*" o *H2M*) o direttamente tra dispositivi ("*machine to machine*" o *M2M*).

1.3 Componenti di un sistema IoT

Di seguito verrà descritta la tipica architettura di un sistema IoT, derivata dal confronto tra varie piattaforme proprietarie e *open-source* . In Figura 1.2 vengono evidenziate le comunicazioni tra i componenti generalmente presenti nel sistema, sebbene non tutti siano necessari allo sviluppo di un'architettura IoT.

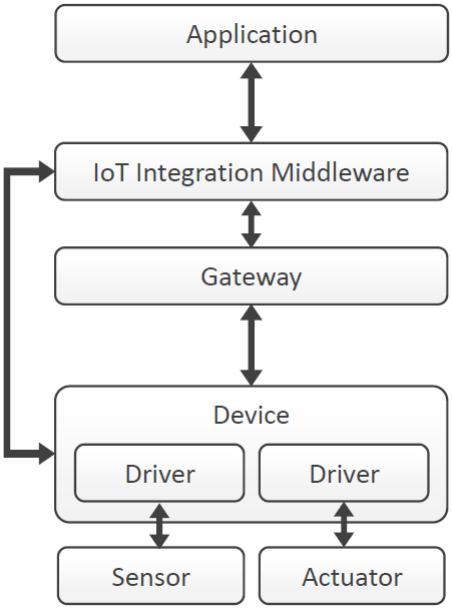

Figura 1.2. Architettura tipica di un sistema IoT

1.3.1 Dispositivi e sensori

Il livello più basso di un sistema IoT è costituito da dispositivi intelligenti integrati con sensori e attuatori. I sensori abilitano l'interconnessione tra mondo digitale e fisico, permettendo alle informazioni di essere collezionate e processate in tempo reale. Per meglio comprendere i vari dispositivi, consideriamo la seguente classificazione:

- *Sensore*: componente hardware utilizzato per misurare parametri dell'ambiente fisico in cui è collocato e tradurli in segnali elettrici (ad esempio, misurare l'umidità e la temperatura di una stanza). Tipicamente, i sensori sono connessi o integrati in un dispositivo più complesso nel quale vengono raccolti i dati inviati. La connessione può essere cablata o senza fili.
- *Attuatore*: componente hardware che può manipolare l'ambiente fisico (ad esempio, emettendo un segnale ottico o acustico). Gli attuatori ricevono comandi dal

dispositivo a cui sono connessi, traducendo un impulso elettrico in un qualche tipo di azione fisica. Essi hanno le stesse modalità di connessione dei sensori.
- *Dispositivo*: componente hardware connesso a sensori e/o attuatori attraverso una connessione cablata o wireless. Generalmente sono richiesti dei software in formato "driver" per fare da interfaccia verso l'esterno, processare i dati dei sensori e controllare gli attuatori. I dispositivi sono, quindi, il punto di contatto tra il mondo digitale e il mondo fisico. Essi possono essere connessi sia tra loro (formando una *black-box* di funzionalità) che ad un altro sistema esterno (ad esempio, un middleware IoT).

1.3.2 Gateway e reti

I sensori producono elevati volumi di dati. Ciò richiede un'infrastruttura di rete (cablata o senza fili) robusta che faccia da mezzo di trasporto per il grande flusso di dati. I dispositivi sono spesso connessi ad un gateway nel caso in cui essi non siano in grado di connettersi direttamente ad altri sistemi (ad esempio, se il dispositivo non può comunicare attraverso un determinato protocollo o per via di limitazioni tecniche). Per risolvere problemi di questo genere, si utilizza un gateway in grado di fornire le tecnologie richieste e le funzionalità che permettano di operare attraverso protocolli differenti, inoltrando la comunicazione tra i dispositivi ad esso connessi e altri sistemi.

Un gateway, in questo modo, ha il compito di supportare i protocolli di comunicazione in entrambi i lati e di tradurre i dati trasportati, se necessario. Un esempio è quello di un dispositivo che comunica con un gateway attraverso un protocollo tipico dell'IoT, come l'MQTT o lo ZigBee: quando riceve un messaggio in formato binario proprietario, il gateway traduce l'informazione in JSON o XML e lo inoltra al sistema interessato attraverso la rete internet.

Allo stesso modo, un gateway può tradurre comandi in altre tecnologie, protocolli e formati di comunicazione supportati dai dispositivi. I gateway possono inoltre essere in grado di eseguire funzioni per l'elaborazione dei dati, come aggregazioni dei dati raccolti dai sensori.

1.3.3 Middleware

Questo componente rende possibile l'elaborazione delle informazioni attraverso analisi, controlli di sicurezza, modellazione dei processi e gestione dei dispositivi. Queste funzionalità sono generalmente messe fornite da alcuni *middleware* IoT dedicati all'integrazione. Essi fanno da intermediari tra il livello di gestione e i dispositivi connessi, ricevendo ed elaborando i loro dati e mettendoli a disposizione di moduli interni. Questi ultimi hanno varie funzioni, tra cui:

- *Attivazione di regole* (definite nei cosiddetti "motori di regole"), basate su condizioni e azioni da svolgere in seguito a determinati eventi (ad esempio, accensione di una luce in seguito al rilevamento della presenza in una stanza attraverso un sensore di prossimità).
- *Fornitura dei dati ricevuti dai sensori ad altre applicazioni* e controllo dei dispositivi per inviare loro comandi da far eseguire dai rispettivi attuatori. Se il

dispositivo non può comunicare direttamente con il *middleware*, ci si serve di un gateway che traduca le informazioni (conversione di protocollo e formato dei dati).
- *Memorizzazione dei dati raccolti* all'interno di basi di dati permanenti (basate su serie temporali o storiche).
- *Visualizzazione dei dati* su interfacce di gestione grafica.

1.3.4 Applicazioni

Numerosi software applicativi utilizzano i *middleware* IoT in svariati settori applicativi, come i trasporti, lo *smart-building* e le *smart-city*, la manifattura, l'agricoltura, l'industria, le catene di distribuzione, la sanità, il turismo, l'ambiente e l'energia. Questi software comunicano con i *middleware* IoT per ottenere dati d'interesse ad alto livello o per effettuare automazioni per il controllo fisico dei dispositivi sul campo (ad esempio: un sistema software che controlla la temperatura di un intero edificio).

1.4 Paradigmi di elaborazione dati

1.4.1 Cloud ed edge computing

Negli ultimi dieci anni, il *cloud computing* ha svolto un ruolo dominante per eseguire elaborazioni di dati complesse e su vasta scala, sfruttando le risorse virtualizzate, l'elaborazione parallela e l'integrazione dei servizi con l'archiviazione scalabile dei dati. Tuttavia, a causa della crescita esplosiva dei dispositivi connessi, che stanno guidando l'intera "società dell'informazione" nell'era dell'IoT, il *cloud computing* sta affrontando sfide crescenti nel supportare una così vasta struttura, in particolare per le applicazioni IoT sensibili al ritardo e al contesto. Poiché un numero crescente di applicazioni IoT viene distribuito per l'elaborazione dei dati in tempo reale, spostare il provisioning dei servizi in prossimità dei dispositivi IoT diventa un potenziale modo per affrontare le sfide del *cloud computing* e promuove l'emergere e lo sviluppo dell'*edge computing*.

Seguendo questa tendenza, diversi concetti e paradigmi, come i *cloudlet*[2], il *fog computing* e il *mobile edge computing*, sono stati proposti negli ultimi anni per soddisfare le esigenze di elaborazione e archiviazione dei dati. A differenza del *cloud computing*, che centralizza l'elaborazione dei dati nel cloud, l'idea alla base dell'*edge computing* è quella di gestire le richieste di elaborazione ed archiviazione dei dati dei dispositivi IoT dall'interno di qualche altra infrastruttura che si trovi in prossimità del dispositivo stesso, riducendo così i tempi di risposta e supportando servizi dipendenti dalla posizione dei dispositivi.

I concetti e i paradigmi dell'*edge computing* hanno mostrato un grande potenziale per supportare applicazioni IoT in tempo reale, ma pochi di essi possono fornire

[2] Con questo termine si intende una banca dati in scala ridotta e orientata alla mobilità, solitamente situata sull'edge.

una soluzione efficiente per l'accesso a servizi multipiattaforma o su richiesta all'interno di dispositivi leggeri. Poiché la maggior parte dei terminali ha scarse capacità di elaborazione e archiviazione, questa necessità diventa così una missione critica, ma anche impegnativa, nell'era dell'IoT, motivando il fatto per cui l'*edge computing* dovrebbe essere ulteriormente sviluppato per migliorare la scalabilità dei dispositivi IoT da questa prospettiva e per soddisfare le sempre crescenti richieste degli utenti.

1.4.2 Transparent computing

Il *transparent computing* è un paradigma molto vicino ai concetti di *cloud* e *edge computing*; esso ha la potenzialità di con il potenziale per fornire servizi scalabili all'edge per dispositivi IoT leggeri. Infatti, esso è in grado di dividere logicamente il software e l'hardware dei dispositivi IoT per consentire l'esecuzione flessibile di software multipiattaforma e per garantire l'accesso ai servizi su richiesta. Un certo numero di lavori esistenti proposti nell'ultimo decennio ne hanno dimostrato l'efficacia e le prestazioni nella costruzione di computer e tablet scalabili su reti locali (LAN) ad alta velocità o su reti locali senza fili (WLAN).

Tuttavia, entrando nell'era dell'IoT, i nuovi terminali e ambienti di rete portano con sé nuove sfide nell'applicazione del *transparent computing* per la costruzione di piattaforme IoT scalabili, come la progettazione di soluzioni efficienti per l'implementazione del *transparent computing* su sistemi con risorse limitate e la stabilità delle prestazioni in contesti caratterizzati da comunicazioni NB-IoT, wireless e/o con un elevato numero di dispositivi mobili.

2

Descrizione del gateway complessivo

In questo capitolo si fornirà una panoramica descrittiva sul funzionamento e la finalità del gateway multiprotocollo per l'IoT. La descrizione introdurrà concetti e termini chiave preliminari, utili nelle fasi di analisi e di progettazione. Infine, verrà presentato il design ad alto livello di un'architettura semplificata del gateway e sarà riportata una descrizione dei principali moduli e delle tecnologie coinvolte.

2.1 Concetti preliminari

2.1.1 Digital twin

Essendo alla base dell'IoT, i dispositivi sono un elemento chiave nello sviluppo del gateway multiprotocollo. La varietà di dispositivi intelligenti a disposizione è molto vasta e sempre crescente, con un numero di nuovi dispositivi immessi nel mercato pari a circa 38 miliardi nel 2018 (più di tre volte quello stimato nel 2012) e circa 50 miliardi entro la fine del 2020, come mostrato nel grafico in Figura 2.1.

La maggior parte dei produttori di dispositivi intelligenti utilizza un formato proprietario per la rappresentazione dei dati, in unione a un numero di protocolli di comunicazione diversi sempre maggiore. Pertanto, a causa dell'offerta crescente, le combinazioni di strutture dati e modalità di trasmissione degli stessi è diventata enorme, con un aumento proporzionale della necessità di uniformare e semplificare la comunicazione e la grande mole di dispositivi connessi.

Un primo passo verso questa semplificazione consiste nella rappresentazione del *tipo* di dispositivo in esame, definendo le proprietà logiche e/o fisiche che esso possiede. Per ogni tipo di dispositivo è possibile, infatti, astrarre delle proprietà che permettano di rappresentare ogni suo stato nel tempo. Si prenda come riferimento una lampadina: essa può essere accesa o spenta, può avere una determinata intensità luminosa o una certa tonalità di colore, e così via.

In Figura 2.2 è riportato un diagramma di stato che mette in luce un sottoinsieme dei possibili stati finiti di una lampadina a led di tipo RGB *dimmerabile*. La lampadina in esame è rappresentata attraverso il tipo definito col nome "luce" e dotato di tre proprietà:

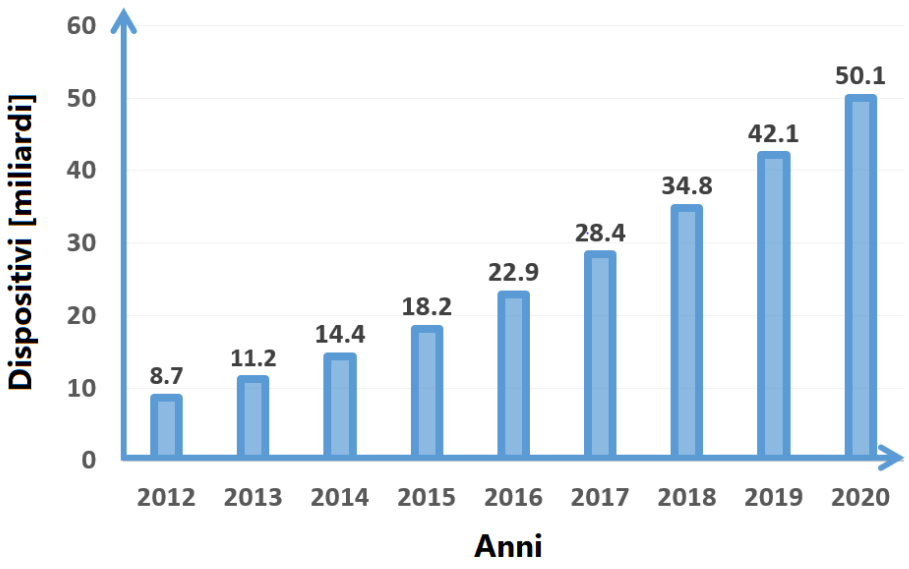

Figura 2.1. Numero di dispositivi IoT connessi dal 2012 al 2020

- *Status*: rappresenta lo stato di accensione o spegnimento della luce attraverso un valore booleano binario: *ON* (vero), *OFF* (falso).
- *Intensità*: quantità associata al dimmer[1] che regola l'intensità luminosa, rappresentata da un valore intero nell'intervallo da 0 a 100 (inteso come percentuale).
- *Colore*: valore esadecimale che esprime la tonalità di colore della luce RGB.

Ogni stato del diagramma rappresenta un preciso stato in cui il dispositivo di tipo luce può trovarsi. Ogni stato è definito in funzione del valore assunto dalle tre proprietà associate, mentre le transizioni tra gli stati rappresentano una variazione di una delle proprietà. Combinando una o più transizioni, si ottiene la transizione da un qualunque stato ad un qualunque altro stato del dispositivo. Tale rappresentazione, espressa mediante un Automa a Stati Finiti Deterministico (ASFD) permette, quindi, di classificare ogni dispositivo sulla base di una collezione di proprietà che definiscono i suoi stati possibili.

Una volta definito il tipo di dispositivo che si intende rappresentare, è possibile costruire il suo *digital twin*, ovvero il suo "gemello digitale". Un *digital twin* è formato da un componente fisico (*physical device*) e un componente virtuale (*virtual device*); essendo l'unione di una rappresentazione digitale e di un dispositivo fisico, il *digital twin* consente di mappare le proprietà reali del dispositivo fisico all'interno di una struttura dati virtuale, il cui schema è predefinito. In questo contesto, tale schema è, per l'appunto, dato dalle proprietà che definiscono il tipo di dispositivo.

[1] Regolatore elettronico utilizzato per controllare la potenza assorbita da un carico.

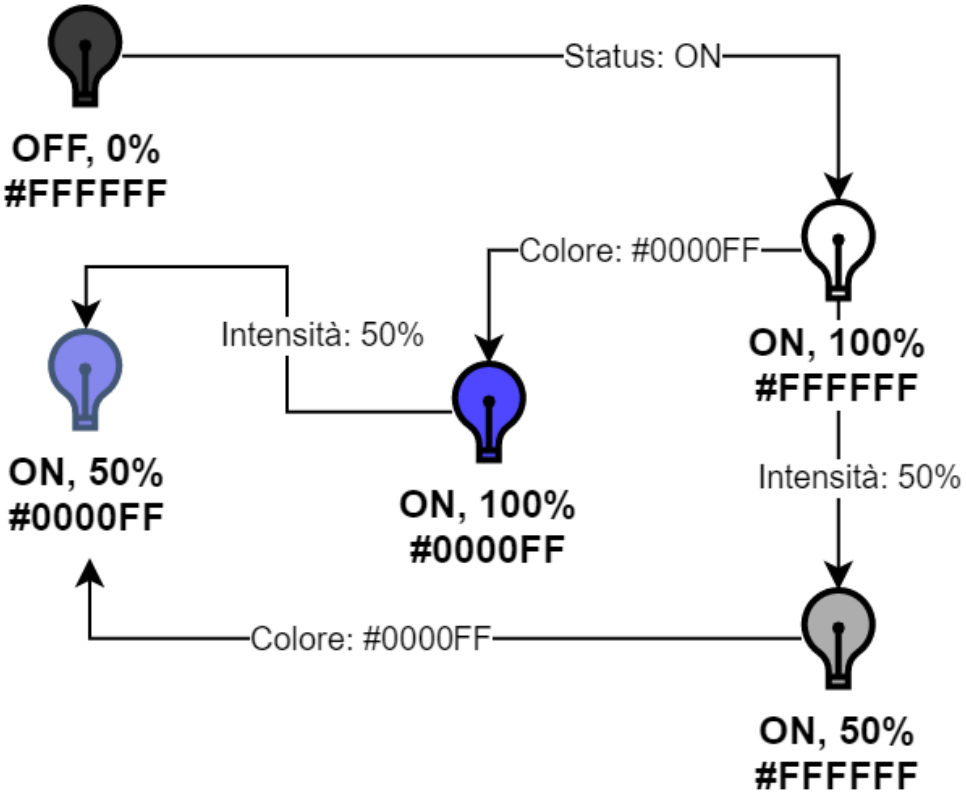

Figura 2.2. Esempio di transizione atomiche degli stati del tipo "luce"

Da notare come, in quanto formalmente definito dall'utente, un tipo non debba necessariamente mappare le stesse esatte proprietà del dispositivo fisico, potendo prevedere vari tipi di manipolazione del dato. Alcuni possibili manipolazioni sono le seguenti:

- *Trasformazione* del formato di una proprietà (ad esempio, si mappa il colore di una luce in formato esadecimale, applicando trasformazioni al dato da formato RGB, XY, etc).
- *Aggregazione* di diverse proprietà fisiche in una proprietà logica unica (ad esempio, una luce con tre proprietà distinte per i canali R, G e B viene mappata logicamente in un unico colore in formato esadecimale).
- *Derivazione* di una proprietà che non esiste nel dispositivo fisico, attraverso delle operazioni su una o più proprietà fisiche esistenti (ad esempio, si ricava lo stato di accensione e spegnimento di una luce dal valore dell'intensità luminosa, rispettivamente, se esso è maggiore di zero o nullo).
- *Filtraggio* del numero di proprietà logiche rispetto a quelle fisiche (ad esempio, non si mappano gli effetti di una luce *smart*, come l'intermittenza, la transizione di colore, etc).

2.1.2 Servizi e richieste

Un altro concetto chiave, strettamente legato al concetto di tipo, è quello di *servizio* esposto da un dispositivo. Nel mondo dell'IoT, una prerogativa principale dei dispositivi è la connettività e la comunicabilità, spesso in ingresso e in uscita rispetto al dispositivo. Sebbene la lettura dei dati (e, quindi, la comunicazione in uscita dai dispositivi) sia piuttosto comune e facilmente realizzabile nei contesti IoT, la varietà di protocolli di comunicazione utilizzati dai dispositivi in commercio rende le automazioni (ovvero le comunicazioni in ingresso ai dispositivi) un contesto più difficile da affrontare. A differenza dei sensori, che offrono la sola lettura del dato, gli attuatori permettono, infatti, delle operazioni attuabili sui dispositivi fisici.

Riprendendo l'esempio del tipo "luce", una lampadina *smart* offre, generalmente, la possibilità di accendere e spegnere il dispositivo da remoto, attraverso una comunicazione con il protocollo utilizzato dal dispositivo. Per alcuni modelli di lampadine, le operazioni disponibili comprendono, anche, la variazione di intensità luminosa e del colore della luce, la regolazione del calore e l'applicazione di effetti di transizione. Questo genere di operazioni possono essere viste come dei "servizi fisici" esposti dal dispositivo, in quanto permettono, su richiesta, di modificare le sue proprietà fisiche.

Più in generale, un "servizio" si può definire come una funzione, associata a un tipo k, che, dato uno stato iniziale s_0 del dispositivo, permette di ottenere uno stato finale s_1 del medesimo. In simboli, dato un insieme $S = \{s0, ..., s_n\}$ di stati possibili per il dispositivo, un servizio si definisce come:

$$f : f_k(s_0) = s_1, \quad s_0, s_1 \in S$$

Si può notare come, ancora una volta, la definizione data di servizio equivalga a quella di "transizione" associata ad un ASFD, con la differenza che la funzione appartiene a una classe, determinata dal tipo del dispositivo. Dividendo i dispositivi considerati per tipo, quindi, si può associare ad ogni tipo un ASFD nel quale gli "stati" sono definiti in funzione dei valori delle proprietà del tipo e le "transizioni" (o combinazioni di esse) rappresentano i servizi richiesti.

In questo modo, essendo lo stato del dispositivo un insieme di valori delle proprietà associate al tipo, ne consegue che l'applicazione di un servizio risulta essere una modifica di tali proprietà. Il risultato della richiesta di un servizio è, quindi, assimilabile all'esecuzione di una funzione che fornisca come risultato una collezione di chiavi (una per ogni proprietà associata al tipo) insieme ai corrispettivi valori (equivalenti al valore desiderato per la proprietà).

2.2 Il gateway multiprotocollo

Il gateway multiprotocollo è un *edge gateway* per l'IoT, con la funzione principale di semplificare la comunicazione tra dispositivi intelligenti presenti sul campo e l'ambiente esterno, sia esso l'utente o un altro sistema. La definizione di "multiprotocollo" deriva dalla capacità di tale gateway di uniformare, con un linguaggio omogeneo, sia le richieste di servizio verso i dispositivi che i dati da essi raccolti.

Per la definizione data di "tipo di dispositivo" e di "servizio", il gateway multiprotocollo è un oggetto che consente di gestire in maniera identica due dispositivi fisici diversi (ovvero, con diverso protocollo di comunicazione e/o proprietà fisiche), qualora essi siano assimilabili concettualmente allo stesso tipo. Lo schema definito dal tipo di dispositivo comune permette, infatti, di mappare le loro diverse proprietà fisiche nelle stesse proprietà logiche e di poter, quindi, applicare gli stessi servizi, definiti in forma agnostica rispetto alla tecnologia e ai protocolli di comunicazione dei dispositivi. In altri termini, l'appartenenza ad uno stesso tipo garantisce la possibilità di collezionare i dati di campo dei due dispositivi in una struttura con lo stesso schema e, quindi, di richiedere gli stessi servizi in modo equivalente, rendendoli difatti "trasparenti" all'utente rispetto alla loro tecnologia.

Considerando le funzionalità precedentemente definite, il gateway multiprotocollo deve prevedere dei moduli in grado di raccogliere le richieste di servizio provenienti dall'utente, di tradurle verso i singoli dispositivi e di raccogliere i loro dati fisici in una rappresentazione uniforme. Dal momento che l'architettura prevede una gestione trasparente dei dispositivi associati, i loro dati devono essere accessibili in modo centralizzato, così come le richieste di servizio, che devono essere effettuabili attraverso un modulo centrale. Le comunicazioni tra l'utente e il sistema devono avvenire attraverso interfacce di facile accesso, mentre quelle interne al sistema devono prevedere la scalabilità dello stesso, permettendo l'aggiunta, su necessità, di nuovi canali di comunicazione senza impattare sugli altri moduli già esistenti.

2.3 Design dell'architettura

Nello sviluppo del design dell'architettura si è optato per un sistema a microservizi distribuiti. Questo genere di architettura è basato sullo sviluppo di "microservizi", ovvero software indipendenti di piccole dimensioni, messi in comunicazione da interfacce (API o scambio di messaggi) ben definite. In questo modo si garantisce l'autonomia dei singoli moduli e una delimitazione chiara delle loro funzionalità, permettendo così di ottenere una maggiore scalabilità del sistema.

I microservizi interessati nello sviluppo dell'architettura in esame sono tre:

- *Hub*: modulo unico e centrale dell'architettura; esso opera come collegamento tra il sistema e gli Add-on.
- *Add-on*: moduli dedicati alla traduzione delle informazioni e alla comunicazione da/verso i dispositivi.
- *Bus*: modulo di comunicazione tra microservizi; esso ricopre il ruolo di *broker* e smistatore dei messaggi scambiati tra gli altri moduli del sistema.

In Figura 2.3 viene riportato un diagramma dell'architettura adottata. Di seguito viene riportata una descrizione più dettagliata dei tre componenti principali coinvolti nella trattazione.

2.3.1 Add-on

Esso rappresenta l'intermediario tra le richieste che provengono dall'utente e i dispositivi fisici, e per i dati provenienti da questi ultimi ed esposti all'utente. In

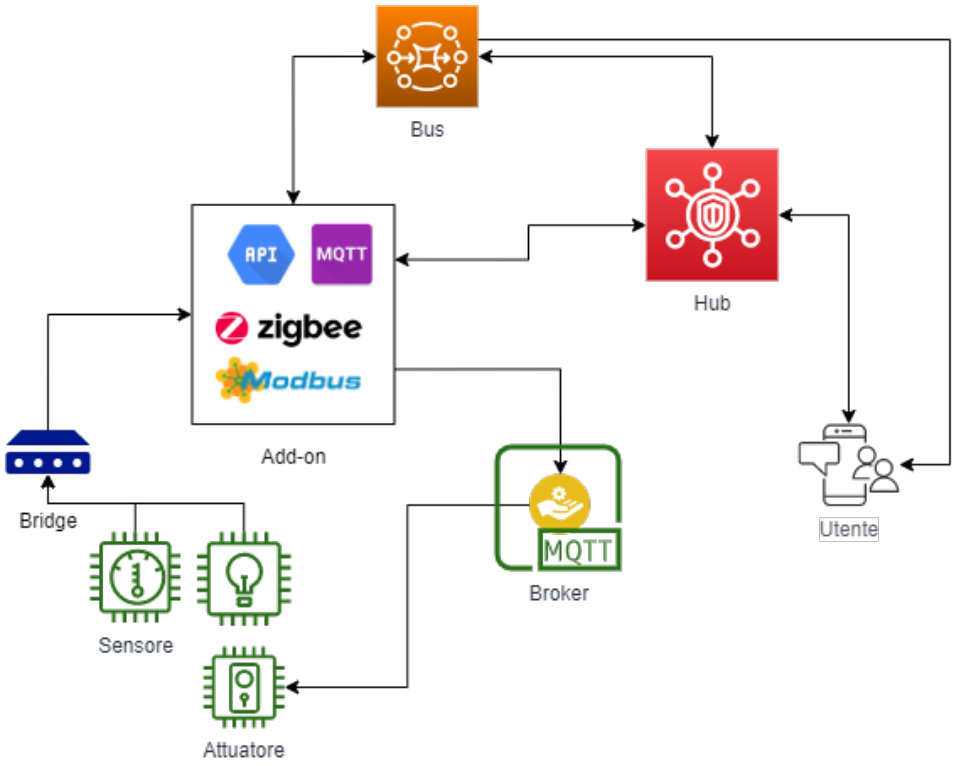

Figura 2.3. Design di architettura del gateway multiprotocollo

particolare, esso costituisce il gestore effettivo dei *digital twin*, ovvero delle connessioni tra dispositivi virtuali e i rispettivi dispositivi fisici. Ogni Add-on è un modulo separato dagli altri, che gestisce autonomamente i dispositivi di una specifica tecnologia, *vendor*[2] o protocollo di comunicazione. Gli Add-on non hanno informazioni sugli altri microservizi eccetto l'Hub, col quale sono in collegamento per ricevere messaggi contenenti le informazioni riguardanti i nuovi dispositivi registrati e l'esecuzione di servizi, e per l'invio di dati elaborati e tradotti a partire dai dati dei dispositivi fisici.

2.3.2 Hub

Esso rappresenta il modulo centrale, dove sono contenute le informazioni degli Add-on e dei dispositivi virtuali accoppiati ad essi. Esso espone API dedicate alla gestione della memoria centrale e alla richiesta di servizi verso i dispositivi. L'Hub assegna ad ogni nuovo Add-on registrato un *AddonID* univoco, e ad ogni dispositivo registrato un *DeviceID* univoco, associato all'*AddonID* dell'Add-on che lo gestisce. La coppia *DeviceID-AddonID* di un dispositivo costituisce la chiave per l'instradamento dei messaggi tra Hub e Add-on attraverso il Bus.

[2] Dall'inglese "venditore", è utilizzato per riferirsi al produttore di una tecnologia o, in modo improprio, alla tecnologia stessa.

I dispositivi registrati hanno sempre tre informazioni:

- *ID*: l'ID univoco del dispositivo nel sistema;
- *Tipo*: il tipo a cui è associato il dispositivo;
- *Metadati*: collezione chiave-valore di metadati associati al dispositivo.

I metadati, in particolare, sono quelli necessari a mappare le caratteristiche fisiche del dispositivo e, quindi, a legare un dispositivo fisico alla sua rappresentazione virtuale.

Ogni volta che all'Hub arriva una richiesta di servizio, indirizzata ad uno specifico dispositivo attraverso il suo ID univoco, dalla memoria centrale dell'Hub viene estratto l'*AddonID* a cui viene accoppiato il *DeviceID*. La richiesta di servizio viene, quindi, trasformata in uno stato desiderato per il dispositivo, ovvero in una combinazione di valori per le proprietà definite per il suo tipo. Attraverso il Bus viene, quindi, inviato un messaggio di innesco del servizio sull'Add-on individuato, il quale conterrà:

- *il DeviceID del dispositivo interessato*; questo viene utilizzato all'interno dell'Add-on per ottenere i metadati, in modo da indirizzare il rispettivo dispositivo fisico.
- *Stato desiderato*: esso viene ottenuto come risultato della funzione associata al servizio richiesto, composto da una collezione chiave-valore delle proprietà da modificare e dei suoi valori desiderati.

2.3.3 Bus

Modulo basato su tecnologia *broker-based*, che fa da intermediario nella comunicazione e nello scambio di messaggi tra i moduli. Esso permette una comunicazione asincrona tra i microservizi e la loro totale separazione nello scambio dei messaggi attraverso il design-pattern *publish-subscribe* su cui si basa. Gli agenti coinvolti nello scambio di messaggi possono, infatti, effettuare due sole operazioni sul cosiddetto "intermediario" (ovvero, il *broker*). Entrambe le operazioni sono legate al concetto di *topic*, ovvero agli argomenti di interesse dai quali si vogliono ricevere messaggi. Esse sono:

- *Sottoscrizione*: procedura di registrazione, sul *broker*, ad un *topic* di interesse e di ricezione dei messaggi ad esso associati.
- *Pubblicazione*: procedura di invio, sul *broker*, di un messaggio associato ad un *topic*.

Quando un agente effettua la pubblicazione di un messaggio su un *topic*, esso viene ricevuto dal *broker*, che lo replica e lo instrada verso gli agenti che si sono sottoscritti a quel *topic*. In questo modo, ogni agente è ignaro dell'identità degli altri agenti durante lo scambio di messaggi, ed è possibile scalare l'applicazione in termine di agenti coinvolti senza intaccare lo scambio di messaggi tra altri agenti già esistenti.

2.4 Tecnologie utilizzate

In questa sezione diamo uno sguardo alle principali tecnologie utilizzate durante la tesi. Esse sono le seguenti:

- *Python*: linguaggio di programmazione utilizzato per lo sviluppo dei moduli software. La scelta del linguaggio è dovuta alla sua potenza nell'ambito dell'elaborazione dei dati, al suo alto livello di programmazione e alla caratteristica di gestire in modo efficiente la *meta-programmazione*[3], il formato JSON (ampiamente utilizzato nello scambio di messaggi del sistema) e i dizionari (collezioni modificabili di dati non ordinati e indicizzati per chiave).
- *Docker*: tecnologia per la *containerizzazione*[4] e *deployment*[5] dei microservizi. Esso permette di impacchettare ogni modulo (e tutte le sue dipendenze) all'interno di un oggetto chiamato "immagine". Ogni immagine è composta da vari *layer*, i quali definiscono le risorse e le operazioni di cui necessita per essere eseguita. L'esecuzione di un'immagine avviene all'interno di un "container", ovvero un ambiente software isolato che utilizza il sistema operativo dell'*host*. Ciò consente l'esecuzione indipendente dei vari moduli e la loro distribuzione su qualunque ambiente Linux (nativo o virtualizzato) in modo scalabile.
- *Flask*: framework leggero, scritto in Python e basato su protocollo Web Server Gateway Interface (WSGI), destinato alla creazione di *web-server* e *web-application*. Esso viene impiegato nello sviluppo delle API esposte dall'Hub.
- *Advanced Message Queue Protocol (AMQP)*: protocollo di comunicazione asincrono che definisce uno standard per lo scambio di messaggi basato su code. Esso fornisce funzionalità di consegna di messaggi e tecniche di instradamento degli stessi attraverso regole basate su *wild-card* e con metodi di *broadcast*, accodamento diretto o *topic-based*. *RabbitMQ* è il *middleware* impiegato per implementare il protocollo AMQP, utilizzato nell'architettura come *broker* e base del Bus di comunicazione.
- *Message Queue Telemetry Transport (MQTT)*: protocollo di comunicazione asincrono e leggero, anch'esso *broker-based*, tipico dei sistemi IoT. Esso è stato ideato per lo scambio di dati con basso impiego di banda (e, quindi, per un consumo ridotto di batteria dei dispositivi sul campo). Come l'AMQP, esso consente la strutturazione di *topic* personalizzati all'interno di un *broker* (in questo caso, *Mosquitto*, un *broker* MQTT *open-source*), sebbene sia meno scalabile per sistemi di grandi dimensioni. Esso viene impiegato come protocollo di riferimento per l'Add-on sviluppato nella presente tesi.

[3] Tecnica di programmazione in cui i software hanno la capacità di trattare essi stessi, o altri software, come loro dati.

[4] Tecnica di gestione del software che permette la virtualizzazione delle applicazioni software.

[5] Fase di rilascio e distribuzione del software.

3

Il sottosistema per la traduzione dei protocolli e la comunicazione dei dispositivi: analisi dei requisiti

In questo capitolo verranno analizzati i requisiti per lo sviluppo del gateway multiprotocollo, attraverso un'analisi delle informazioni riportate discorsivamente nel precedente capitolo. Dopo aver effettuato una raccolta dei requisiti, i principali casi d'uso saranno affinati con una suddivisione più schematica dei requisiti funzionali, riportati in forma tabellare, e di quelli non funzionali.

3.1 Raccolta dei requisiti

La raccolta dei requisiti è basata sull'approfondimento di ogni funzionalità necessaria per lo sviluppo del gateway multiprotocollo, così come individuata nell'architettura proposta nel precedente capitolo, in fase di design della soluzione.

Di seguito sono riportati i casi d'uso individuati, ottenuti dagli scenari necessari per il funzionamento del sistema. Ciascuno di essi sarà descritto più nel dettaglio in fase di analisi dei requisiti funzionali, con l'aggiunta di eventuali dettagli implementativi.

3.1.1 Casi d'uso

In un diagramma dei casi d'uso, come quello mostrato in Figura 3.1, sono coinvolti due attori, ovvero:

- *Utente*: rappresenta qualunque utente finale o sistema esterno che utilizza le funzionalità del gateway multiprotocollo.
- *Sistema*: rappresenta l'effettivo insieme di moduli software che compongono il gateway e che espongono le funzionalità coinvolte nei casi d'uso raccolti.

Il diagramma dei casi d'uso relativo al gateway multiprotocollo, oggetto della presente tesi, viene riportato nella Figura 3.1. Come si evince nella Figura 3.1, è presente un caso d'uso, denominato "Invoca API dell'Hub", che compare unicamente come estensione di altri casi d'uso più specifici. Dal momento che nella presente tesi non ci occuperemo in dettaglio dello sviluppo dell'infrastruttura dell'Hub e del sistema per la generazione di API REST, possiamo ridurre lo scenario dei casi d'uso come riportato in Figura 3.2.

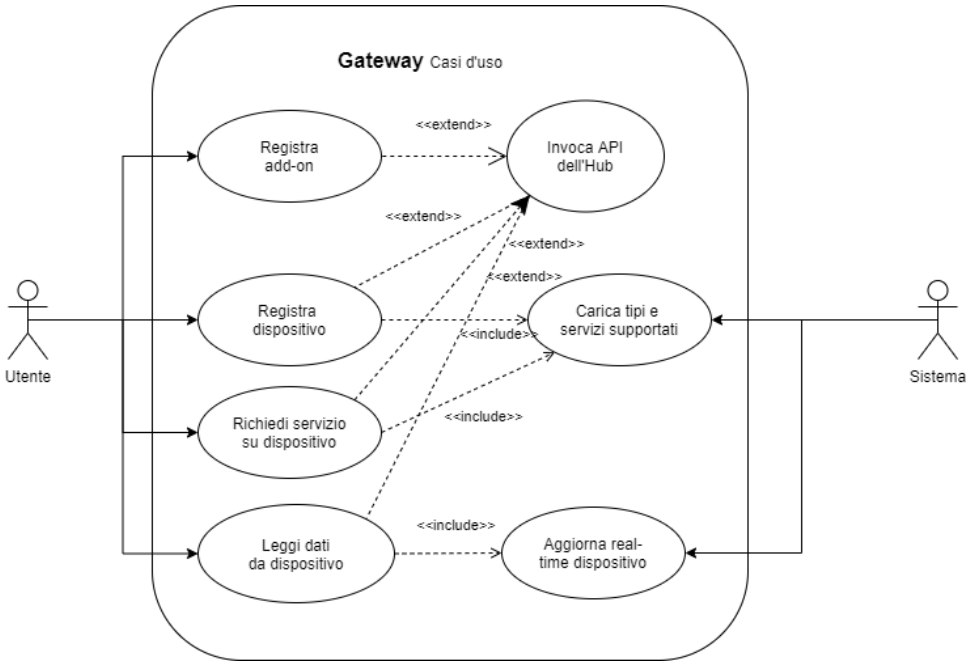

Figura 3.1. Casi d'uso del gateway multiprotocollo

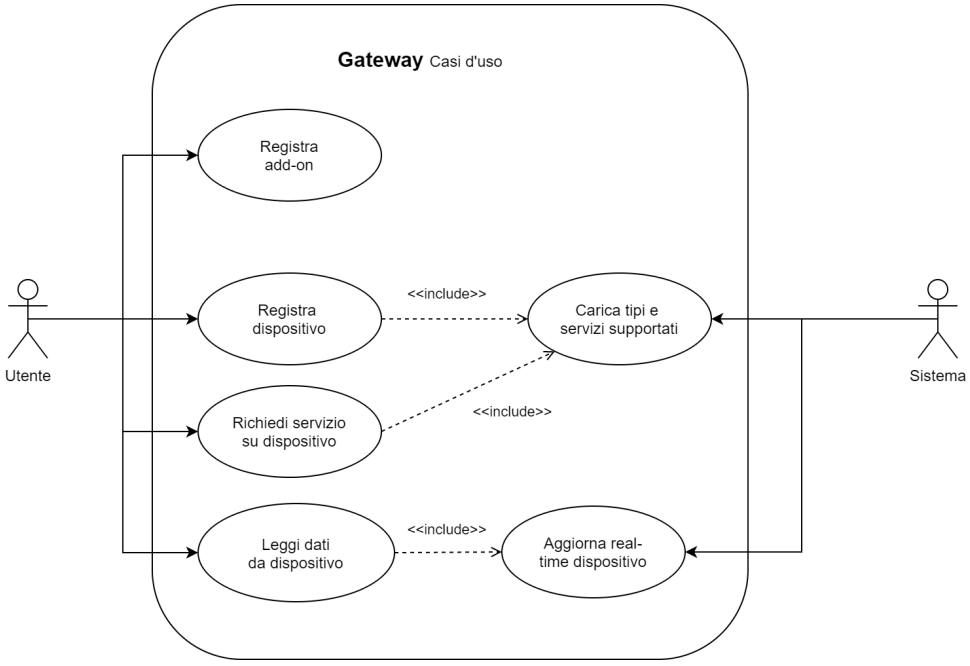

Figura 3.2. Casi d'uso del gateway multiprotocollo relativi alla presente tesi

3.2 Requisiti funzionali

I casi d'uso definiti sono stati utilizzati per individuare i requisiti funzionali necessari per approfondire la fase di analisi. Tali requisiti, elencati e classificati nella Tabella 3.1, riportano i seguenti parametri:

- *Codice*: identificatore alfanumerico del requisito funzionale, utilizzato per fare riferimento al requisito durante lo sviluppo o per identificarlo come dipendenza nella tabella dei requisiti.
- *Epica*: nome autodescrittivo della funzionalità; rappresenta una *user story*[1] (che non può essere frammentata in parti più piccole senza perdere parte della sua funzionalità).
- *Dipendenze*: elenco dei codici di identificazione che individuano altre funzionalità necessarie per lo sviluppo del requisito.

I requisiti sono ordinati tenendo conto delle dipendenze, sia funzionali che logiche, di ciascuno di essi. La loro disposizione permette, così, di avere un'idea chiara del funzionamento del sistema, emulando un tipico scenario applicativo, partendo dalla registrazione di ogni risorsa necessaria, fino al loro impiego in un contesto pratico.

I requisiti funzionali relativi al gateway multiprotocollo sono riportati nella Tabella 3.1.

CODICE	EPICA	DIPENDENZE
F01	Caricamento di tipi e servizi supportati	...
F02	Registrazione di un Add-on	F01
F03	Registrazione di un dispositivo	F02
F04	Richiesta di servizio su un dispositivo registrato	F03
F05	Aggiornamento real-time dei dati di un dispositivo	F03
F06	Lettura dei dati in memoria di un dispositivo	F05

Tabella 3.1. Elenco dei requisiti funzionali

Di seguito viene riportata una descrizione di dettaglio per ciascuno di essi.

3.2.1 Caricamento dei tipi e dei servizi

In fase di inizializzazione, a partire da una configurazione esterna al codice, l'Hub deve caricare la struttura dei tipi di dispositivo supportati. Ogni tipo di dispositivo dichiarato deve riportare lo schema delle proprietà standard che lo definiscono. Ciascuna proprietà di un tipo deve essere identificata da un nome e dalla struttura dati con cui essa viene rappresentata. Nella dichiarazione del tipo deve essere possibile definire opzionalmente la lista degli identificativi dei servizi ad esso associati.

[1] Dall'inglese, è una descrizione informale e in linguaggio naturale di una o più funzionalità di un sistema software.

Per ciascun servizio associato ad un tipo deve esistere una e una sola funzione in grado di ricevere in input i dati del dispositivo ed eventuali parametri opzionali. Le informazioni dei tipi di dispositivo e dei servizi associati devono essere salvate nel sistema di memorizzazione dell'Hub.

3.2.2 Registrazione di un Add-on

La registrazione di un Add-on è possibile attraverso una chiamata alle API dell'Hub. In fase di registrazione, l'Add-on deve definire il proprio nome univoco e ricevere in risposta un identificativo chiave, necessario per effettuare le future richieste verso l'Hub. Durante l'inizializzazione, l'Add-on deve recuperare dall'Hub le informazioni sui tipi supportati e sui dispositivi registrati che sono stati associati all'Add-on. Le informazioni dell'Add-on devono essere salvate nel sistema di memorizzazione dell'Hub.

3.2.3 Registrazione di un dispositivo

La registrazione di un nuovo dispositivo è possibile attraverso una chiamata alle API dell'Hub. Per registrare il dispositivo, è necessario fornire un nome identificativo, il tipo di dispositivo supportato dal sistema ad esso associato e indicare l'Add-on della tecnologia che lo supporta. Inoltre, deve essere presente un'informazione sui metadati, ovvero i dati necessari alla configurazione e all'identificazione del dispositivo fisico. Al termine del processo, l'Hub deve comunicare l'avvenuta registrazione del dispositivo all'Add-on a cui esso è stato associato. Le informazioni del dispositivo devono essere salvate nel sistema di memorizzazione dell'Hub.

3.2.4 Richiesta di servizio su un dispositivo

La richiesta di esecuzione di un servizio su uno dei dispositivi registrati è possibile attraverso una chiamata alle API dell'Hub. In fase di invocazione di un servizio, è necessario fornire i seguenti attributi:

- *Target IDs*: lista degli identificativi dei dispositivi registrati nel sistema sui quali invocare il servizio.
- *Nome*: nome del servizio da richiedere, corrispondente al nome di uno dei servizi registrati nel sistema.
- *Parametri*: dati parametrici necessari all'esecuzione del servizio.

Il sistema deve recuperare la funzione associata al nome del servizio da richiedere. Per ciascun dispositivo su cui invocare il servizio, il sistema deve recuperare l'identificativo dell'Add-on a cui è collegato il dispositivo ed eseguire la funzione associata al servizio. I dati risultanti dall'esecuzione della funzione devono essere inviati, attraverso il Bus di comunicazione, all'Add-on che gestisce il dispositivo. Alla ricezione del messaggio, l'Add-on deve elaborare i dati della richiesta in funzione del tipo e dei metadati del dispositivo, del suo stato attuale e della tecnologia/protocollo relativo all'Add-on, traducendoli in un'azione sul dispositivo fisico.

3.2.5 Aggiornamento in real-time dei dati

Durante l'esecuzione, l'Add-on deve aggiornare, in tempo reale, i dati dei dispositivi che controlla, attraverso un ciclo di sincronizzazione. Per ogni dispositivo controllato, deve essere attivo un processo parallelo che, con frequenza stabilita, esegua una funzione di sincronizzazione che tenga conto delle informazioni del dispositivo e del suo stato attuale. La funzione di sincronizzazione deve leggere i dati dal dispositivo fisico associato, attraverso istruzioni legate alla tecnologia e/o al protocollo relativo. I dati letti durante ogni ciclo di sincronizzazione devono essere tradotti in valori per le proprietà del tipo di dispositivo, compatibili con le strutture dati ad esso associate.

I dati così standardizzati devono essere inviati all'Hub, attraverso un messaggio sul Bus, e salvati all'interno di un registro degli stati. Quest'ultimo deve contenere un'occorrenza per ogni dispositivo registrato nel sistema e un campo di memoria per ogni proprietà associata al tipo di dispositivo.

3.2.6 Lettura dei dati di un dispositivo

La lettura dei dati di un dispositivo (ovvero, il suo ultimo stato registrato) dalla memoria centrale è possibile attraverso una chiamata alle API dell'Hub. Per leggere lo stato, è necessario fornire l'identificativo del dispositivo interessato. Il sistema deve recuperare le informazioni relative all'identificativo del dispositivo presenti nel registro degli stati dell'Hub. Tali informazioni devono essere formattate tramite delle coppie chiave-valore, con una chiave per ogni proprietà associata al tipo e il relativo valore, così come memorizzato durante l'ultimo aggiornamento ricevuto dall'Add-on a cui è associato il dispositivo. Il contenuto della risposta deve specificare anche il tempo in cui è stato registrato l'ultimo aggiornamento dello stato. del dispositivo.

3.3 Requisiti non funzionali

Sono stati individuati due requisiti non funzionali che il sistema deve rispettare e dei quali è necessario tenere conto nelle successive fasi di progettazione e validazione:

- *Scalabilità*: l'aggiunta di nuovi tipi di dispositivi e di nuove tecnologie deve essere possibile senza intaccare il funzionamento degli altri componenti del sistema.
- *Trasparenza*: il sistema deve standardizzare il formato dei dati di dispositivi appartenenti allo stesso tipo e permettere la chiamata di servizi legati ad un certo tipo di dispositivo in modo equivalente e indipendente dalla tecnologia del dispositivo reale.

4

Il sottosistema per la traduzione dei protocolli e la comunicazione dei dispositivi: progettazione

In questo capitolo verrà illustrata la progettazione del nostro sottosistema. Tale attività conta di due fasi. La prima fase si occuperà di progettare le interazioni tra i vari moduli. La seconda fase prevederà, invece, la progettazione delle componenti strutturali; al tal fine verrà utilizzato il diagramma delle classi.

4.1 Progettazione dei comportamenti

Per la prima fase della progettazione, verranno realizzati i diagrammi dei comportamenti tra gli attori coinvolti nei moduli ottenuti in fase di analisi. La progettazione dei comportamenti per ogni epica può essere divisa in due categorie, ciascuna caratterizzata da un diverso approccio. Le categorie sono:

1. *Progettazione orientata alle sequenze*: le epiche F01-F03 e F06 presentano una scarsa quantità di stati a disposizione e prediligono un flusso di dati lineare e sequenziale; esse saranno progettate attraverso un diagramma delle sequenze.
2. *Progettazione orientata agli stati*: le epiche F04 e F05 riguardano il ciclo di vita dello stesso componente (Add-on), essendo caratterizzate da una dipendenza dagli stati dello stesso; esse saranno progettate attraverso un unico diagramma degli stati in aggiunta al diagramma delle sequenze.

Di seguito saranno riportati i diagrammi che mettono in luce i comportamenti adottati tra gli attori di moduli e sotto-moduli del sistema. La progettazione di ogni modulo sarà accompagnata da una discussione delle relative modalità adottate e da una sintetica spiegazione dei corrispondenti diagrammi elaborati.

4.1.1 Caricamento di tipi e servizi

La prima progettazione orientata alle sequenze è quella dell'epica F01, riguardante il modulo dell'Hub. Esso, in fase preliminare, si suddivide in tre sotto-moduli:

- *Hub*: il modulo principale, opera come controllore ad alto livello delle operazioni.
- *ServiceManager*: il gestore dei servizi; raccoglie le informazioni sui servizi registrati.

- *Storage*: il gestore della memoria persistente.

L'Hub carica il file di configurazione relativo ai tipi di dispositivo supportati, elaborando il contenuto al fine di generare una lista di tipi. In un secondo momento, avviene la lettura dei file situati in un percorso di configurazione dedicato ai servizi. Ogni servizio è associato ad una cartella, che ha lo stesso nome del servizio. All'interno di tale cartella sono situati due file:

- *Script*: contiene la procedura principale da eseguire alla richiesta del servizio; tale procedura riceve in ingresso i dati contenuti nel registro degli stati appartenenti al dispositivo (per disporre di servizi basati sullo stato attuale) nonché eventuali parametri opzionali definiti in fase di richiesta del servizio (per disporre di servizi parametrici).
- *File di configurazione*: metadati in formato JSON riguardanti informazioni e modalità di esecuzione del servizio, tra cui la firma della funzione principale contenuta nello script.

In Figura 4.1 è rappresentato il diagramma delle sequenze della funzionalità. Per ogni tipo rilevato, l'Hub richiede una verifica dei servizi associati al ServiceManager; essa restituisce il contenuto dei servizi e un valore booleano sulla sua validità. Se i servizi configurati ed associati al tipo sono validi, l'Hub memorizza tutto il risultato dell'elaborazione nello Storage.

4.1.2 Registrazione di un Add-on

L'epica F02 riguarda la registrazione di un Add-on. Tale operazione avviene con una richiesta da parte dell'Add-on che deve registrarsi verso le RESTful API dell'Hub.

Nella Tabella 4.1 è riportata la descrizione dell'interfaccia della richiesta. Nelle Tabelle 4.2 e 4.3 sono riportate, invece, le descrizioni della risposta alla richiesta e dei relativi codici HTTP. Infine, in Figura 4.2, è rappresentato il diagramma delle sequenze della relativa funzionalità.

Metodo: POST
Indirizzo: /addons

Parametro	Tipo	Locazione	Descrizione	Necessario
name	Stringa	Body	Nome univoco dell'Add-on	SI
config_dir	Stringa	Body	Percorso delle configurazioni	SI
meta	Oggetto	Body	Informazioni specifiche per la tecnologia	NO

Tabella 4.1. Registrazione di un Add-on: parametri della richiesta

L'Add-on comunica le informazioni all'Hub per effettuare un auto-censimento. L'Hub verifica dapprima l'esistenza di informazioni legate al nome univoco dell'Add-on, con un controllo sullo Storage. Se l'Add-on non è mai stato censito, l'Hub verifica l'esistenza del percorso di configurazione all'interno del file system su cui esso è in esecuzione. Se queste verifiche hanno successo, l'Hub salva le informazioni ricevute nello Storage e restituisce l'URI della risorsa creata.

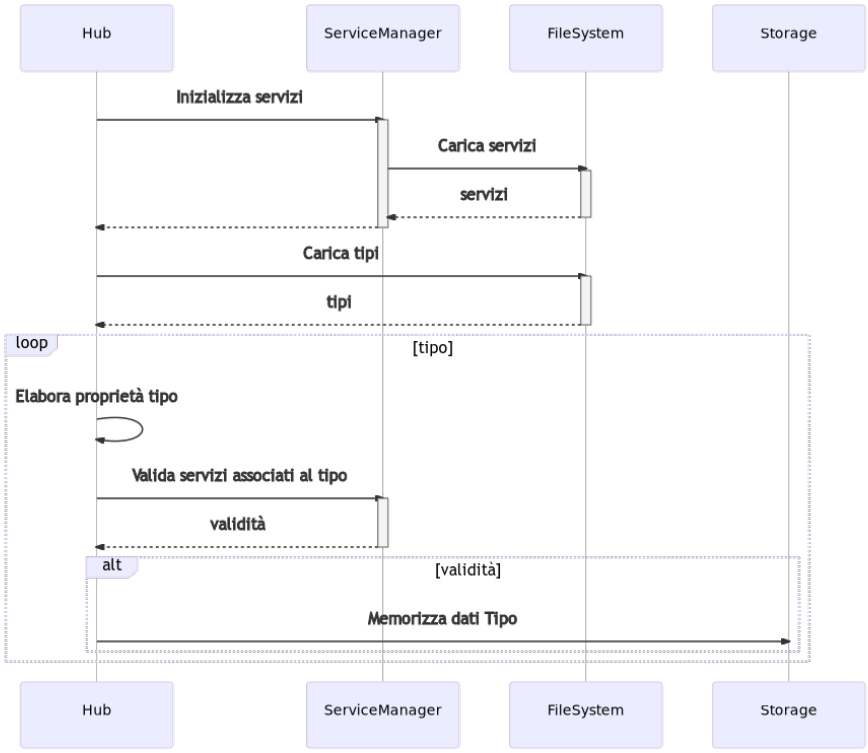

Figura 4.1. Diagramma delle sequenze relativo al caricamento di tipi e servizi

Parametro	Tipo	Locazione	Descrizione
location	Stringa	Header	URL della risorsa creata

Tabella 4.2. Registrazione di un Add-on: parametri della risposta

Codice	Descrizione
204	Dispositivo registrato con successo
403	Nome del dispositivo già esistente nel sistema
500	Errore interno o inaspettato

Tabella 4.3. Registrazione di un Add-on: codici HTTP della risposta

4.1.3 Registrazione di un dispositivo

L'epica F03 riguarda la registrazione di un dispositivo. Tale operazione avviene con una richiesta da parte dell'utente verso le RESTful API dell'Hub.

Nella Tabella 4.4 è riportata la descrizione dell'interfaccia della richiesta. Nelle Tabelle 4.5 e 4.6 sono riportate, invece, le descrizioni della risposta alla richiesta e dei relativi codici HTTP. Infine, in Figura 4.3, è rappresentato il diagramma delle sequenze della relativa funzionalità.

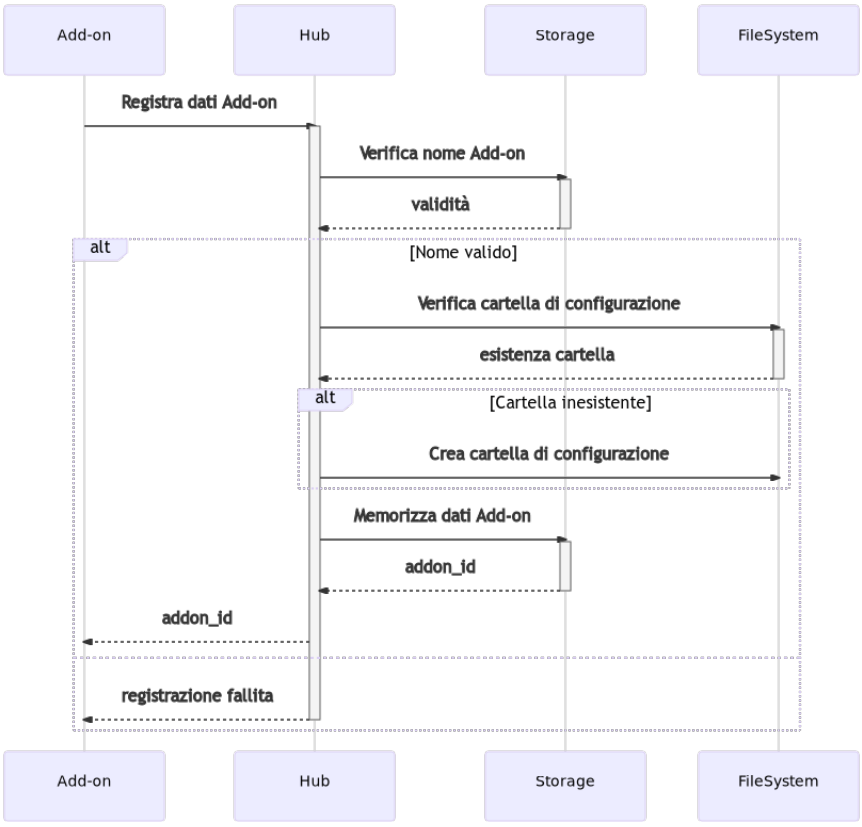

Figura 4.2. Diagramma delle sequenze relativo alla registrazione di un Add-on

Metodo: POST
Indirizzo: /devices

Parametro	Tipo	Locazione	Descrizione	Necessario
name	Stringa	Body	Nome univoco del dispositivo	SI
type	Stringa	Body	Tipo associato al dispositivo	SI
addon	Stringa	Body	Nome dell'Add-on associato	SI
meta	Oggetto	Body	Dati di riferimento del dispositivo fisico	NO

Tabella 4.4. Registrazione di un dispositivo: parametri della richiesta

L'Hub verifica che il nome non corrisponda a nessun altro dispositivo già associato e che il tipo e l'Add-on di riferimento esistano all'interno dello Storage. Se tali verifiche hanno successo, l'Hub salva le informazioni ricevute nello Storage, dal quale viene estratto il *DeviceID* univoco generato. L'Hub crea, quindi, un nuovo stato nel registro degli stati, inizializzato in funzione delle proprietà legate al tipo di dispositivo associato; il nuovo stato creato è anch'esso identificato dal medesimo *DeviceID*. Al termine delle operazioni, l'Hub comunica, mediante Bus, la registra-

zione del dispositivo all'Add-on a cui è stato associato, comunicando il *DeviceID* generato, e restituisce l'URI della risorsa creata.

Queste operazioni prevedono un sotto-modulo aggiuntivo dell'Hub, denominato StateRegistry, incaricato di gestire le informazioni del registro degli stati.

Parametro	Tipo	Locazione	Descrizione
location	Stringa	Header	URL della risorsa creata

Tabella 4.5. Registrazione di un dispositivo: parametri della risposta

Codice	Descrizione
204	Add-on registrato con successo
403	Nome dell'Add-on già esistente nel sistema
500	Errore interno o inaspettato

Tabella 4.6. Registrazione di un dispositivo: codici HTTP della risposta

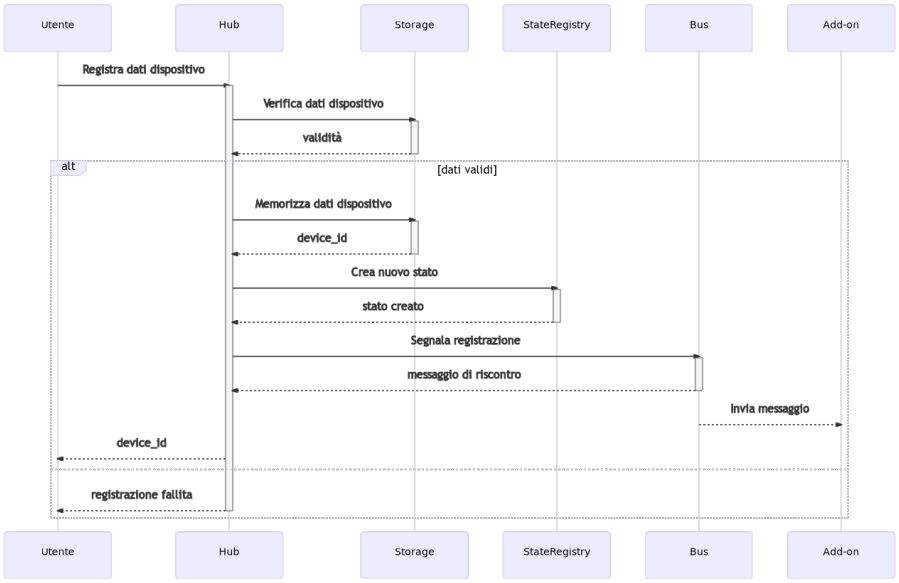

Figura 4.3. Diagramma delle sequenze relativo alla registrazione di un dispositivo

4.1.4 Ciclo di vita di un Add-on e modulo traduttore

Nel diagramma di stato in Figura 4.4 è riportato il ciclo di vita di un Add-on. Esso prevede le seguenti fasi:

- verifica della connessione verso l'Hub e verso il Broker;
- registrazione delle informazioni sull'Hub;
- caricamento dei dispositivi registrati;
- avvio del loop di esecuzione;
- sincronizzazione dei dati dei dispositivi gestiti;
- ricezione di un messaggio dall'Hub; questa fase prevede le seguenti sottofasi:
  - elaborazione di un servizio richiesto;
  - aggiunta di un nuovo dispositivo.
- verifica della condizione di arresto.

Tra i processi dell'Add-on riportati nel diagramma, due sono di particolare interesse per la progettazione del modulo di traduzione. Essi possono essere scomposti, a loro volta, nei seguenti sotto-processi:

- *Elaborazione di una richiesta di servizio*:
  - ricezione di uno stato standard dall'Hub;
  - traduzione dello stato in operazioni fisiche per il dispositivo;
  - esecuzione delle operazioni sul dispositivo.
- *Sincronizzazione di un dispositivo*:
  - esecuzione di operazioni per il rilevamento dei dati fisici del dispositivo;
  - traduzione dei dati fisici in uno stato standard;
  - invio dello stato all'Hub.

Come si può notare, i sotto-processi in cui si scompongono i due processi sono gli stessi, ma disposti in ordine inverso. Da considerare, inoltre, che le modalità di traduzione di uno stato in una o più operazioni dipendono unicamente dalla tecnologia (o protocollo) e dal tipo che caratterizzano il dispositivo. Il "linguaggio" di traduzione per un dispositivo si può, quindi, classificare in base al suo tipo e alla sua tecnologia. Associando ad ogni classe (ovvero alla combinazione di tipo e tecnologia) una serie di funzioni in ingresso (per tradurre dal linguaggio macchina al linguaggio standard) e in uscita (per tradurre, viceversa, dal linguaggio standard al linguaggio macchina), si ottiene il modulo di traduzione.

In altri termini, il traduttore è un *router* di richieste da e verso l'Hub; esso associa ad ogni richiesta di servizio una classe di funzioni in uscita, e ad ogni lettura dei dati dal campo una classe di funzioni in ingresso. Dal momento che ogni Add-on rappresenta una tecnologia e che i tipi definiti nell'Hub rappresentano ciascuno un tipo di dispositivo, è possibile modellare un Add-on come un collettore di classi di funzioni mappate su ogni tipo, come nel diagramma in Figura 4.5.

Ogni tipo al secondo livello dell'albero rappresenta una classe di funzioni, la quale è dotata di funzioni in uscita (se la traduzione avviene da un valore fisico a uno stato, ovvero dal dispositivo all'Hub) e in ingresso (se la traduzione avviene da uno stato al valore fisico, ovvero dall'Hub al dispositivo). Se il tipo di dispositivo non è sufficiente per distinguere tra due classi di funzioni, è possibile registrare un

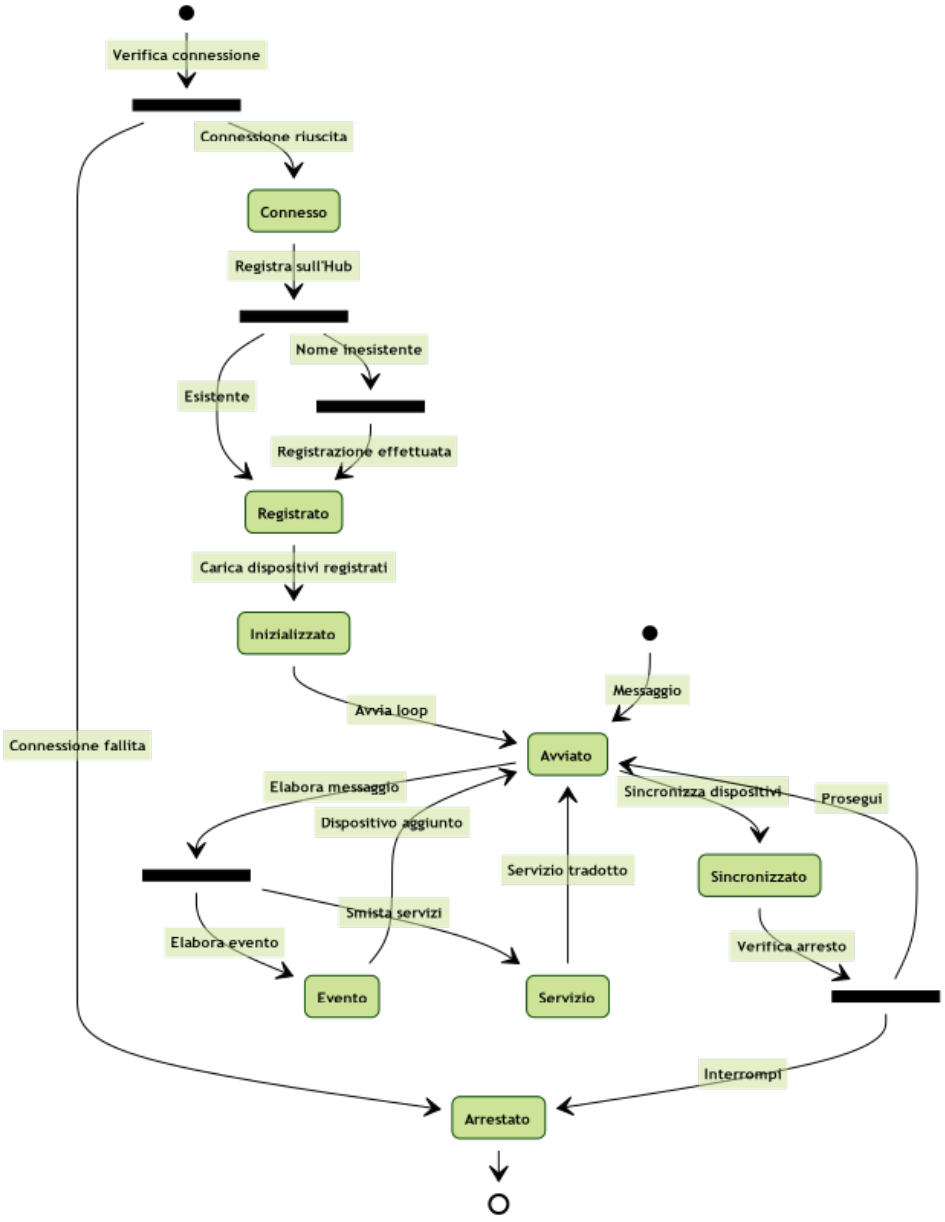

Figura 4.4. Diagramma di stato relativo al ciclo di vita di un Add-on

numero adeguato di metadati nel dispositivo; essi verranno utilizzati internamente all'Add-on per disambiguare la sotto-classe di funzioni da mappare.

Avendo così definito le classi di funzioni in ingresso e in uscita associate ad ogni tipo e per ogni Add-on, tradurre uno stato ricevuto in una richiesta di servizio significa, quindi, effettuare l'instradamento dello stato desiderato dall'Hub verso

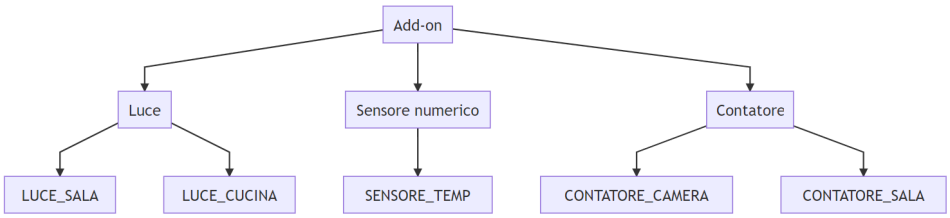

Figura 4.5. Grafo dei dispositivi e dei corrispondenti tipi gestiti da un Add-on

la classe di funzioni associata al dispositivo ed applicare le sue funzioni in uscita. Viceversa, la traduzione di un dato fisico del dispositivo rilevato dal campo equivale ad applicare le funzioni in ingresso della classe associata al dispositivo e indirizzare lo stato risultante all'Hub.

4.1.5 Elaborazione di una richiesta di servizio: Hub

La prima parte dell'epica F04 (epica F04.1) riguarda l'elaborazione di una richiesta di servizio dal lato dell'Hub. Tale operazione avviene con una richiesta da parte dell'utente verso le RESTful API dell'Hub.

Nella Tabella 4.7 è riportata la descrizione dell'interfaccia della richiesta. La Tabella 4.8 riporta, invece, la descrizione dei codici HTTP relativi alla risposta. Infine, in Figura 4.6, è rappresentato il diagramma delle sequenze della relativa funzionalità.

Metodo: PUT
Indirizzo: /services/{service_name}

Parametro	Tipo	Locazione	Descrizione	Necessario
service_name	Stringa	URI	Nome univoco del servizio	SI
targets	Lista	Body	ID dei dispositivi su cui richiedere il servizio	SI
parameters	Oggetto	Body	Parametri del servizio	NO

Tabella 4.7. Elaborazione di una richiesta di servizio: parametri della richiesta

L'Hub effettua un controllo sul nome del servizio per verificarne l'esistenza all'interno del ServiceManager. Se questa verifica ha successo, l'Hub recupera le informazioni dei dispositivi target, effettuando una ricerca basata sull'indice degli ID. Se tutti i dispositivi recuperati appartengono ad un tipo per il quale è supportato il servizio richiesto, l'Hub predispone la funzione configurata nel file di script del servizio e mappa tale funzione su tutti i dispositivi target. Per ognuno di tali dispositivi, viene recuperato lo stato attuale del dispositivo dallo StateRegistry; questi parametri vengono, quindi, utilizzati, insieme a quelli passati nella richiesta, come argomenti della funzione predisposta. Per ogni dispositivo, il risultato della funzione applicata viene, quindi, inviato all'Add-on che lo gestisce attraverso il Bus.

Codice	Descrizione
204	Servizio richiesto con successo
404	Nome del servizio inesistente
500	Errore interno o inaspettato

Tabella 4.8. Elaborazione di una richiesta di servizio: codici HTTP della risposta

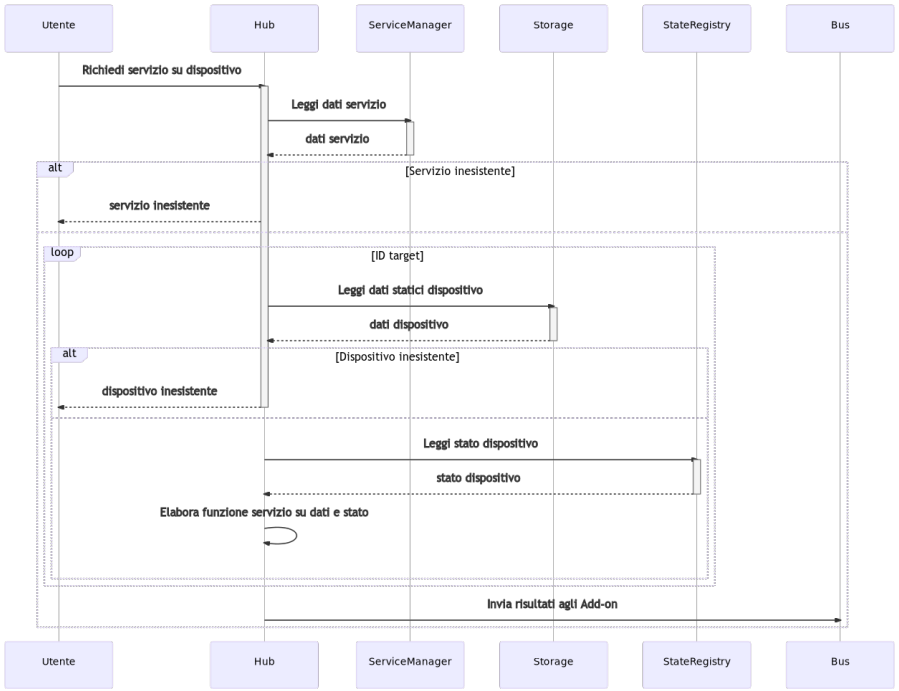

Figura 4.6. Diagramma delle sequenze relativo all'elaborazione di una richiesta di servizio sull'Hub

4.1.6 Elaborazione di una richiesta di servizio: Add-on

La seconda parte dell'epica F04 (epica F04.2) riguarda l'elaborazione di una richiesta di servizio dal lato dell'Add-on.

Ogni Add-on è in ascolto sul Bus rispetto alle richieste di servizio elaborate dall'Hub. Quando esso riceve una richiesta di servizio per un dispositivo con ID, spacchetta il contenuto del messaggio, individuando le coppie di proprietà e relativi valori da applicare al dispositivo. Per ogni dispositivo, l'Add-on recupera la relativa istanza della classe di funzioni in uscita che gestiscono il suo tipo, chiamata DeviceInstance.

Per ogni proprietà contenuta nel messaggio ricevuto, l'Add-on invoca la relativa funzione in uscita associata alla DeviceInstance del dispositivo target, passando per argomento il valore della proprietà. Dal momento che la DeviceInstance incapsula i dati specifici del dispositivo, l'esecuzione della funzione con argomento si applica proprio a questi dati. L'elaborazione della funzione invocata traduce, quindi, il valore

per la proprietà in un processo sul dispositivo fisico, attraverso la tecnologia o il protocollo di comunicazione associato.

Al termine dell'invocazione, la DeviceInstance aggiorna il proprio stato interno sulla base della proprietà modificata e del risultato della funzione invocata.

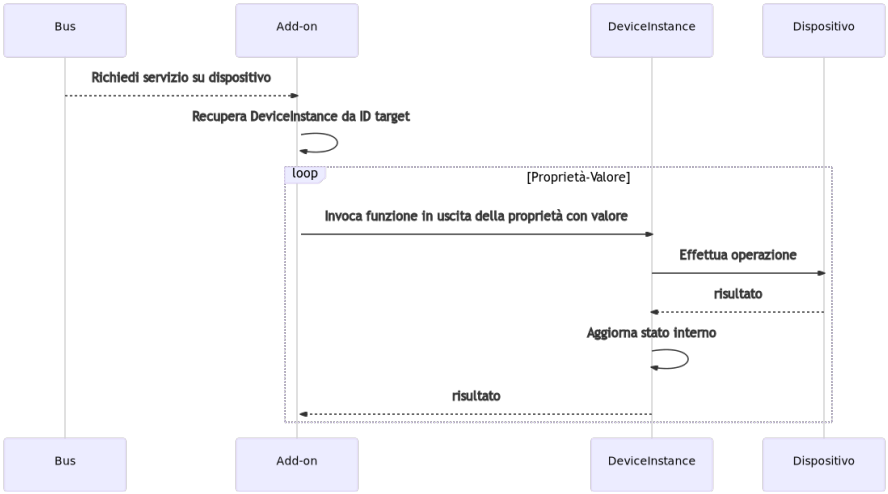

Figura 4.7. Diagramma delle sequenze relativo all'elaborazione di una richiesta di servizio su un Add-on

4.1.7 Sincronizzazione di un dispositivo

L'epica F05 riguarda la sincronizzazione dei dati di un dispositivo all'interno di un Add-on.

Il termine "sincronizzazione" si riferisce impropriamente al processo di aggiornamento dei dati attraverso una procedura atta a far coincidere i dati di un dispositivo fisico col suo *digital twin*. Questa procedura può essere, infatti, di due tipi, in funzione della modalità di ricezione e lettura dei dati dai dispositivi:

- *Polling*: Comunicazione sincrona. L'Add-on interroga il dispositivo periodicamente e riceve i dati in risposta con una procedura bloccante. Fanno parte di questa categoria tutti i protocolli di comunicazione sincroni (REST, ModBus, etc).
- *Event-driven*: Comunicazione asincrona. L'Add-on riceve i dati attraverso un messaggio inviato in un evento generato dal dispositivo. Fanno parte di questa categoria tutti i protocolli di comunicazione asincroni (MQTT, AMQP, etc).

La procedura di *polling* prevede che l'Add-on interroghi periodicamente i suoi dispositivi. Per fare ciò, il suo stato di loop prevede un ciclo temporizzato di verifica, durante la quale inviare ad ogni sua DeviceInstance una richiesta di lettura dei dati dai dispositivi fisici. La richiesta, per ogni DeviceInstance, si converte nell'invocazione delle sue funzioni in ingresso, nel qual caso all'interno di essa si effettua la

lettura dal dispositivo fisico secondo la tecnologia o il protocollo di comunicazione stabilito e la conversione dei valori letti nel formato dei dati definito dalle proprietà del dispositivo.

Il numero di funzioni in ingresso non deve necessariamente coincidere col numero di proprietà che caratterizzano il tipo di dispositivo; la funzione in ingresso può anche essere una sola, all'interno della quale viene effettuata la lettura di tutte le proprietà. Le funzioni in ingresso salvano, quindi, i dati letti all'interno della DeviceInstance, che fornisce il suo stato attuale all'Add-on al termine della richiesta di sincronizzazione. In seguito, l'Add-on invia i dati collezionati in formato normalizzato all'Hub, attraverso un messaggio sul Bus.

La procedura *event-driven* segue un procedimento meno articolato. Durante l'inizializzazione di una DeviceInstance, l'Add-on si mette in ascolto sui canali di comunicazione degli eventi definiti dal protocollo di comunicazione che gestisce, in funzione dei dati statici definiti per il dispositivo, e associa una *callback* ad ogni evento. Il numero di callback definite non deve coincidere necessariamente col numero di proprietà definite per il tipo di dispositivo; la funzione di callback può anche essere una sola per tutte le proprietà.

Per ogni messaggio ricevuto per l'evento, la funzione di *callback* estrae il contenuto delle proprietà secondo il protocollo di comunicazione utilizzato, convertendo i valori letti nel formato dei dati definito dalle proprietà del dispositivo. Anche in questo caso, l'Add-on invia i dati collezionati in formato normalizzato all'Hub, attraverso un messaggio sul Bus.

Nella presente tesi è stata considerata la sola procedura *event-driven* per la gestione di un Add-on basato sul protocollo MQTT e, pertanto, non sarà approfondita quella di *polling*. Nella Figura 4.8 è rappresentato il diagramma delle sequenze della funzionalità.

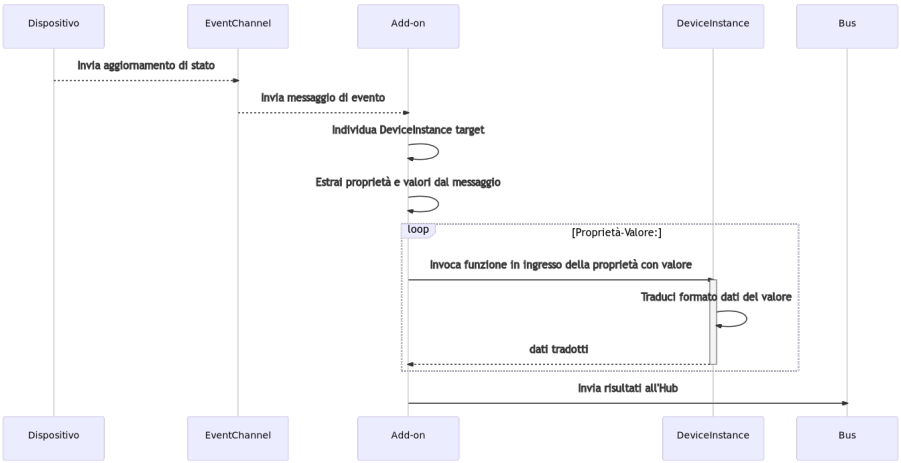

Figura 4.8. Diagramma delle sequenze relativo alla sincronizzazione dei dati di un dispositivo con metodo *event-driven*

4.1.8 Lettura dei dati di un dispositivo

L'epica F06 riguarda la lettura dei dati di un dispositivo contenuti nella memoria centrale. Tale operazione avviene con una richiesta da parte dell'utente verso le RESTful API dell'Hub.

Nella Tabella 4.9 viene riportata la descrizione dell'interfaccia della richiesta. Nelle Tabelle 4.10 e 4.11 sono riportate, invece, le descrizioni della risposta alla richiesta e dei relativi codici HTTP. Infine, in Figura 4.9, è rappresentato il diagramma delle sequenze della relativa funzionalità.

Metodo: GET
Indirizzo: /devices/{device_id}

Parametro	Tipo	Locazione	Descrizione	Necessario
device_id	Stringa	URI	ID univoco del dispositivo	SI

Tabella 4.9. Lettura dei dati di un dispositivo: parametri della richiesta

L'Hub esegue un controllo su base indice (*DeviceID*) riguardo l'esistenza del dispositivo all'interno dello Storage. Se questa verifica ha successo, l'Hub recupera le informazioni registrate nello Storage (per i dati statici) e nello StateRegistry (per i dati dinamici legati allo stato corrente del dispositivo, come ricevuto dall'Add-on di riferimento). Al termine dell'operazione, l'Hub restituisce in formato JSON tutti i dati raccolti.

Parametro	Tipo	Locazione	Descrizione
id	Stringa	Body	ID univoco del dispositivo
name	Stringa	Body	Nome univoco del dispositivo
type	Stringa	Body	Tipo associato al dispositivo
addon	Stringa	Body	Nome dell'Add-on associato
meta	Oggetto	Body	Dati di riferimento del dispositivo fisico
status	Oggetto	Body	Stato delle proprietà del dispositivo

Tabella 4.10. Lettura dei dati di un dispositivo: parametri della risposta

Codice	Descrizione
200	Lettura effettuata con successo
404	ID del dispositivo inesistente
500	Errore interno o inaspettato

Tabella 4.11. Lettura dei dati di un dispositivo: codici HTTP della risposta

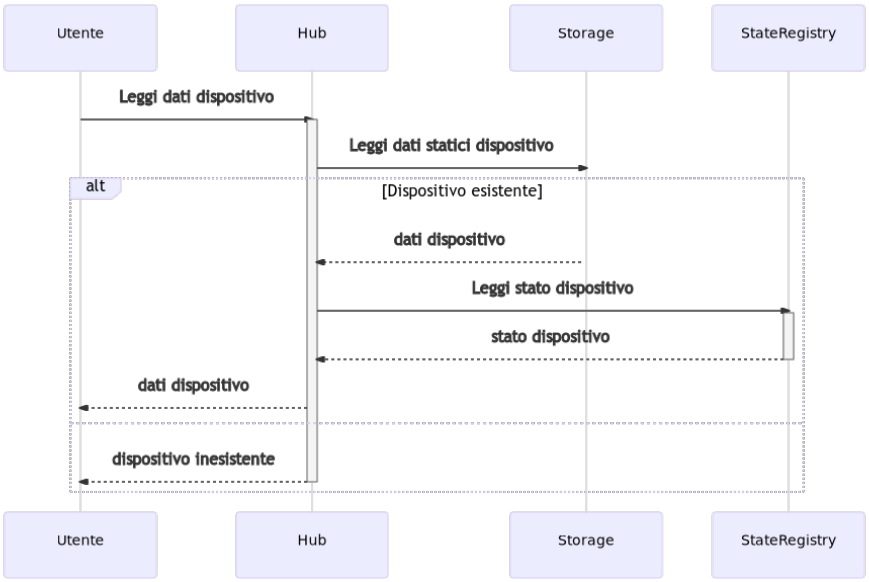

Figura 4.9. Diagramma delle sequenze relativo alla lettura dei dati di un dispositivo

4.2 Progettazione strutturale

Durante la seconda fase della progettazione, l'attenzione sarà focalizzata sulla struttura dei moduli coinvolti nelle principali sequenze su cui si concentra la presente tesi. Le parti d'interesse sono quelle che appartengono al sottosistema per la traduzione dei protocolli e la comunicazione con i dispositivi. Per questo, la progettazione strutturale riguarderà soltanto le epiche F04.2 (Figura 4.7) e F05 (Figura 4.8), assieme al diagramma di stato del ciclo di vita dell'Add-on.

4.2.1 Progettazione delle classi

Sarà, di seguito, riportata la stesura delle classi di progettazione che riguardano le due funzionalità individuate. Le classi di progettazione sono ottenute individuando le entità riportate nei diagrammi delle sequenze delle epiche F04.2 e F05. Tali entità sono le seguenti:

- *Bus*: classe dedicata alla comunicazione con il modulo Bus e il broker che gestisce lo scambio di messaggi tra Hub e Add-on.
- *Addon*: classe principale, destinata alla gestione di tutte funzionalità dell'Add-on.
- *DeviceInstance*: classe che rappresenta il gemello digitale di un dispositivo fisico all'interno di un Add-on. Essa deve essere estesa in classi figlie, una per ogni tipo di dispositivo supportato dall'Add-on, ovvero la classe *MqttLight* (per la gestione delle luci) e la classe MqttSensor (per la gestione dei sensori numerici).
- *EventChannel*: Classe di gestione del *client* dedicato alla ricezione di eventi dal dispositivo e all'invio di operazioni verso il dispositivo. Essa deve essere

implementata nella classe *MqttConnection*, che gestisce la specifica tecnologia di comunicazione in esame.

L'unica entità non considerata come classe di progettazione è l'entità "Device", la quale, rappresentando il dispositivo fisico, non costituisce un elemento del sistema software. In Figura 4.10 è riportato il diagramma delle classi associato alle classi di progettazione precedentemente descritte.

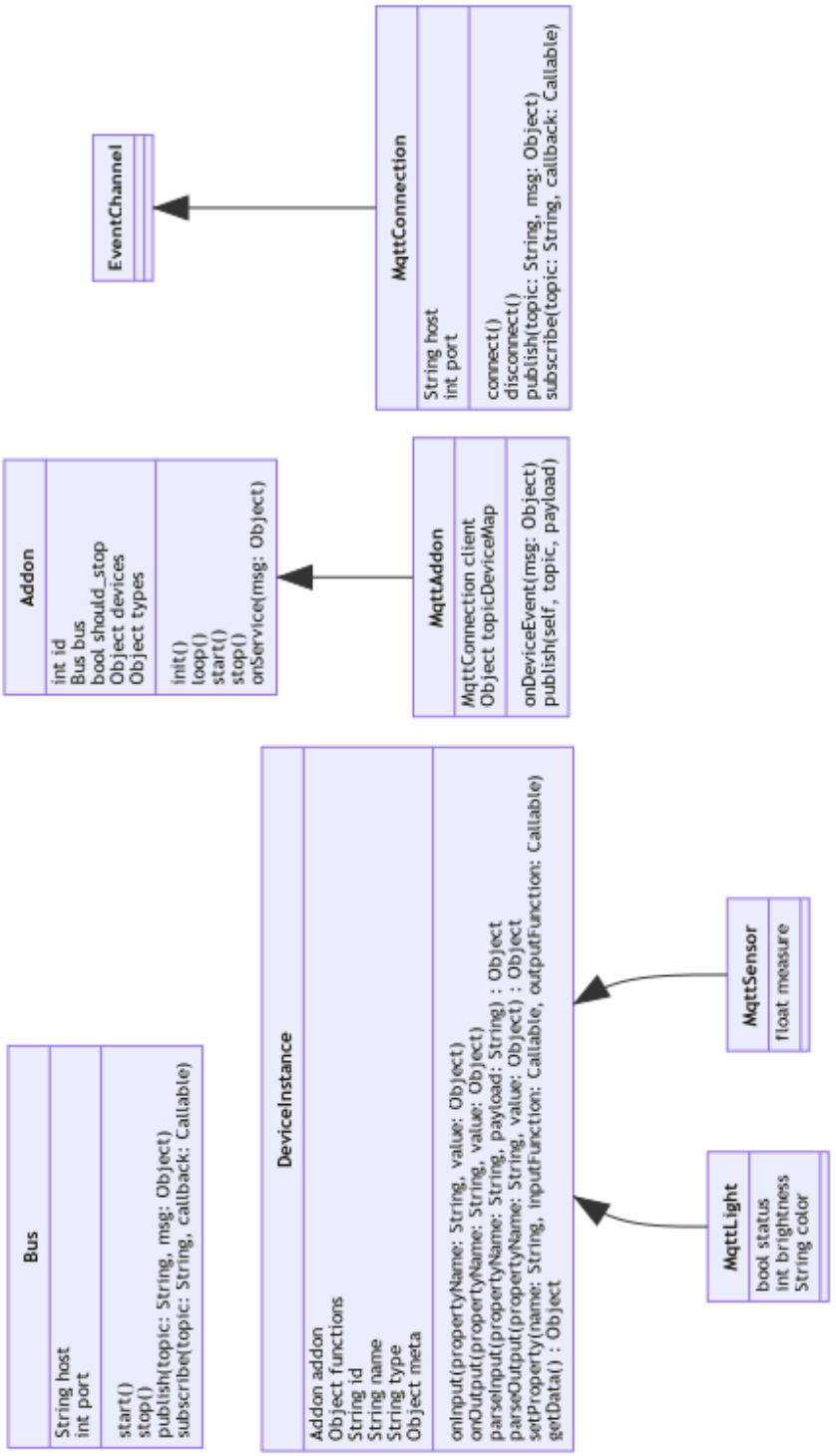

Figura 4.10. Diagramma delle classi di progettazione

5

Il sottosistema per la traduzione dei protocolli e la comunicazione dei dispositivi: implementazione

In questo capitolo verranno implementate, in linguaggio Python, le classi di progettazione ottenute nel precedente capitolo, suddivise per classi di libreria e classi riguardanti il modulo di traduzione per l'MQTT, con una sezione finale dedicata allo script principale per l'esecuzione del modulo. Ogni parte di codice sarà corredata da una descrizione del suo funzionamento e da alcuni dettagli implementativi.

5.1 Classi di libreria

La prima categoria di classi implementate è quella di libreria, ovvero classi di progettazione genitori e non astratte (come, invece, nel caso della classe di progettazione `EventChannel`). Tali classi hanno, infatti, la funzionalità di classi base per l'implementazione di altre classi e funzionalità, rappresentando, quindi, la libreria di sviluppo del software in questione.

5.1.1 Bus

La prima classe di libreria è quella del Bus, la cui implementazione è riportata nel Listato 5.1. Questa classe ha la funzione di *wrapper*, ovvero di modulo di "rivestimento", che, tramite il *design-pattern delegation*, semplifica le operazioni di comunicazione con il modulo "rivestito".

In particolare, il Bus ha la sola funzione di *wrapper* per il modulo `AmqpConnection`: questa è una classe di gestione della connessione per il protocollo AMQP (in particolare, implementata attraverso la libreria `Pika`, che è un'implementazione in Python di *AMQP 0-9-1*).

```
1           from .amqp import AmqpConnection
2
3           import logging
4           LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
5           '-35s %(lineno) -5d: %(message)s')
6           LOGGER = logging.getLogger(__name__)
7           logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
8
9
10          class Bus:
11
12                  def __init__(self, host: str, port: int):
```

```
13                      self._host: str = host
14                      self._port: int = port
15                      self._client: AmqpConnection = AmqpConnection(host, port)
16
17              def __repr__(self):
18                      return f"{self._host}:{self._port}"
19
20              def start(self):
21                      self._client.connect()
22                      LOGGER.info(f"Bus at '{self}' started")
23
24              def stop(self):
25                      self._client.disconnect()
26                      LOGGER.info(f"Bus at '{self}' stopped")
27
28              def publish(self, topic: str, message: object):
29                      self._client.publish(topic, message)
30
31              def subscribe(self, topic: str, callback):
32                      self._client.subscribe(topic, callback)
33                      LOGGER.info(f"Bus at '{self}' subscribed to topic '{topic}'")
```

Listato 5.1. Implementazione della classe `Bus`

La classe `Bus` espone semplici interfacce per l'avvio e l'arresto della connessione AMQP (metodi `start` e `stop`) e lo scambio di messaggi su determinati *topic* (metodi `publish` e `subscribe`).

5.1.2 DeviceInstance

La seconda classe di libreria è `DeviceInstance`, la cui implementazione è riportata nel Listato 5.2. Questa classe riporta metodi per la conversione di dati in *input* e *output* rispetto all'Add-on. In particolare, le classi che estendono da `DeviceInstance` sono quelle che devono rappresentare le modalità in cui ogni tipo di dispositivo traduce i dati nel protocollo dell'Add-on a cui afferisce.

Ogni classe figlia può, infatti, utilizzare il metodo `set_property` per registrare le proprietà che gestisce il proprio tipo di riferimento. Ad ogni proprietà si possono, quindi, associare una funzione d'ingresso (per i tipi "passivi", come i sensori) e una funzione d'uscita (per i tipi "attivi", come gli attuatori). In questo modo, i metodi `parse_input` e `parse_output` possono essere utilizzati per invocare la conversione nel verso corretto, così come determinato nelle funzioni d'ingresso o uscita registrate.

```
1       class DeviceInstance:
2
3               def __init__(self, addon, **kwargs):
4                       self._addon = addon
5                       self._info = kwargs
6                       self._functions = dict(input=dict(), output=dict())
7
8               @property
9               def id(self):
10                      return self._info['id']
11
12              @property
13              def type(self):
14                      return self._info['type']
15
16              @property
17              def name(self):
18                      return self._info['name']
19
20              @property
21              def meta(self):
22                      return self._info.get('meta', {})
23
24              def parse(self, direction: str, property_name: str, data: object) -> object:
25                      function = self._functions[direction].get(property_name)
26                      if function is not None:
27                              value = function(data)
28                              if value is not None:
29                                      return value
30
```

```
31                  def parse_input(self, property_name: str, payload: str) -> bool:
32                          value = self.parse('input', property_name, payload)
33                          should_update = value is not None
34                          if should_update:
35                                  self.on_input(property_name, value)
36                          return should_update
37
38                  def on_input(self, property_name, value):
39                          setattr(self, property_name, value)
40
41                  def on_output(self, property_name, value):
42                          pass
43
44                  def parse_output(self, property_name: str, value: object) -> bool:
45                          value = self.parse('output', property_name, value)
46                          should_update = value is not None
47                          if should_update:
48                                  self.on_output(property_name, value)
49                          return should_update
50
51                  def set_property(self, name, input_function, output_function=None):
52                          self._functions['input'][name] = input_function
53                          if output_function is not None:
54                                  self._functions['output'][name] = output_function
55
56                  @property
57                  def data(self):
58                          return {property_name: getattr(self, property_name)
59                                  for property_name in self._functions['input']}
```

Listato 5.2. Implementazione della classe `DeviceInstance`

5.1.3 Addon

L'ultima classe di libreria implementata è la classe `Addon`, riportata nel Listato 5.3. Questa classe racchiude al proprio interno tutto il ciclo di vita di base di un Add-on e le istanze dei *client* utilizzate per la comunicazione attraverso il Bus e con le API dell'Hub.

Il costruttore della classe `Addon` riceve come argomento il proprio identificativo di riferimento e un dizionario che mappa ogni tipo supportato con le rispettive funzioni di ingresso e uscita. La classe espone due metodi per l'avvio e l'arresto dell'esecuzione (rispettivamente, `start` e `stop`). Il metodo di avvio invoca i metodi necessari per l'esecuzione del ciclo di vita dell'Add-on, inizializzando i client necessari. In questa fase, viene effettuata la sottoscrizione agli eventi di richieste di servizio e, attraverso le API dell'Hub, vengono caricati i dispositivi associati. L'invocazione del metodo `loop` esegue, quindi, un ciclo bloccante sull'Add-on.

```
1           import json
2           import time
3           import typing
4
5           from .device import DeviceInstance
6           from .amqp import Bus
7           from .utils import HubAPI
8           from .configs import *
9
10          import logging
11          LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
12          '-35s %(lineno) -5d: %(message)s')
13          LOGGER = logging.getLogger(__name__)
14          logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
15
16
17          class Addon:
18
19                  def __init__(self, id, **kwargs):
20                          self._id: str = id
21                          self._bus: Bus = None
22                          self._api: HubAPI = None
23                          self._should_stop: bool = False
24                          self._devices: typing.Dict[int, DeviceInstance] = None
25                          self._types: typing.Dict[str, typing.Callable] = kwargs
26
```

```
27                 def start(self):
28                         self._bus = Bus(AMQP_HOST, AMQP_PORT)
29                         self._bus.subscribe(f"{self._id}.service", self.on_service)
30                         self._bus.start()
31                         self._api = HubAPI(HUB_HOST, HUB_PORT)
32                         self._api.init()
33                         LOGGER.info("Getting addon device from Hub..")
34                         self._devices = dict()
35                         for device_data in self._api.get_devices_by_addon(self._id).values():
36                                 device_type = device_data['type']
37                                 device_instance_class = self._types.get(device_type)
38                                         if device_instance_class is None:
39                                                 continue
40                                         device = device_instance_class(self, **device_data)
41                                         self._devices[device.id] = device
42                                         LOGGER.info(f">> Added {device_type} '{device.name}' (id: {device.id})")
43                         self.on_start()
44                         self.loop()
45                         self.on_stop()
46
47                 def loop(self):
48                         while not self._should_stop:
49                                 time.sleep(.1)
50
51                 def stop(self):
52                         self._should_stop = True
53                         self._bus.stop()
54
55                 def on_service(self, _client, _user_data, msg):
56                         payload = json.loads(msg.payload.decode('utf-8'))
57                         device = self._devices.get(payload['device_id'])
58                         if device is not None:
59                                 properties = payload['properties']
60                                 for property_name, property_value in properties.items():
61                                         device.parse_output(property_name, property_value)
62
63                 def on_init(self):
64                         pass
65
66                 def on_start(self):
67                         pass
68
69                 def on_stop(self):
70                         pass
71
72                 def send_device_data(self, device_id, data):
73                         device = self._devices.get(device_id)
74                         if device is not None:
75                                 data['id'] = str(device.id)
76                                 data['type'] = device.type
77                                 self._bus.publish(f"hub.device", json.dumps(data))
```

Listato 5.3. Implementazione della classe `Addon`

Alla ricezione di una richiesta di servizio, il metodo di *callback* denominato `on_service` definisce l'estrazione della `DeviceInstance` target e l'invocazione, per ogni proprietà registrata da quest'ultima, della funzione d'uscita per convertire e inviare i dati. I metodi `on_init`, `on_start` e `on_stop` permettono, con un procedimento di *override*[1], di specificare procedure specifiche che devono avvenire, rispettivamente, al termine dell'inizializzazione, prima del ciclo bloccante e dopo l'arresto del ciclo.

5.2 Classi del modulo

La seconda categoria di classi implementate comprende quelle specifiche per il modulo di traduzione dedicato al protocollo MQTT. Le classi appartenenti a questa categoria sono tutte quelle che estendono classi di libreria.

[1] Operazione di riscrittura di un metodo ereditato da una classe padre.

5.2.1 MqttLight e MqttSensor

Le prime classi del modulo sono `MqttLight` e `MqttSensor`, riportate, rispettivamente, nel Listato 5.5 e nel Listato 5.6.

Queste due classi devono estendere la classe di libreria `DeviceInstance`. Tuttavia, dal momento che entrambe condividono dei comportamenti in comune, tipici dell'implementazione MQTT adottata, si è sviluppata un'ulteriore classe figlia di `DeviceInstance`, che fosse genitore di entrambe, chiamata `MqttDevice`, riportata nel Listato 5.4.

```
1           import re
2
3           from lib import DeviceInstance
4
5
6           class MqttDevice(DeviceInstance):
7
8                   def __init__(self, addon, **kwargs):
9                           super().__init__(addon, **kwargs)
10
11                  @property
12                  def properties(self) -> dict:
13                          return self.meta['properties']
14
15                  def on_output(self, property_name, value):
16                          topic = self.properties[property_name].get('command_topic')
17                          if topic is not None:
18                                  self._addon.publish(topic, value)
```

Listato 5.4. Implementazione della classe padre `MqttDevice`

La classe `MqttLight` si occupa della gestione e della traduzione, tramite protocollo MQTT, dei dispositivi di tipo "luce". Nel costruttore, la classe registra le proprietà `status`, `brightness` e `color`. Per ogni proprietà registrata, nella classe sono definiti due metodi di traduzione (in ingresso e in uscita) e i metodi `getter` e `setter` di Python.

Per ogni proprietà, i metodi di traduzione verificano, se necessario, la validità del formato del valore da tradurre e lo convertono nel formato standard dell'Hub (se la funzione è di input) o nel formato accettato dal dispositivo MQTT (se la funzione è di output).

```
1
2           class MqttLight(MqttDevice):
3
4                   def __init__(self, addon, **kwargs):
5                           super().__init__(addon, **kwargs)
6                           self._status: bool = None
7                           self._brightness: int = None
8                           self._color: str = None
9
10                          self.set_property('status', self._status_input, self._status_output)
11                          self.set_property('brightness', self._brightness_input, self._brightness_output)
12                          self.set_property('color', self._color_input, self._color_output)
13
14                  @property
15                  def _status_payload_on(self) -> str:
16                          return self.meta['properties']['status']['payload_on']
17
18                  @property
19                  def _status_payload_off(self) -> str:
20                          return self.meta['properties']['status']['payload_off']
21
22                  @property
23                  def status(self) -> bool:
24                          return self._status
25
26                  @status.setter
27                  def status(self, value):
28                          self._status = value
29
30                  def _status_input(self, value: str) -> bool:
31                          if value in [self._status_payload_on, self._status_payload_off]:
```

```
32                              return value == self._status_payload_on
33
34              def _status_output(self, value: bool) -> str:
35                      return self._status_payload_on if value else self._status_payload_off
36
37              @property
38              def _brightness_scale(self) -> int:
39                      return self.meta['properties']['brightness'].get('scale', 100)
40
41              @property
42              def brightness(self) -> int:
43                      return self._brightness
44
45              @brightness.setter
46              def brightness(self, value):
47                      self._brightness = value
48
49              def _brightness_input(self, value: str) -> int:
50                      value = int(value)
51                      if 0 <= value <= self._brightness_scale:
52                              return round((value/self._brightness_scale)*100)
53
54              def _brightness_output(self, value: int) -> int:
55                      return round((value / 100) * self._brightness_scale)
56
57              @property
58              def color(self) -> str:
59                      return self._color
60
61              @color.setter
62              def color(self, value):
63                      self._color = value
64
65              def _color_input(self, value: str) -> str:
66                      if re.search(r'^(?:[0-9a-fA-F]{1,2}){3}$', value):
67                              return value
68
69              def _color_output(self, value: str) -> str:
70                      return value
```

Listato 5.5. Implementazione della classe padre `MqttLight`

La classe `MqttSensor` si occupa della gestione e della traduzione, mediante protocollo MQTT, dei dispositivi di tipo "sensore" numerico. L'implementazione è analoga a quella della classe `MqttLight`, ma più semplice, dal momento che è prevista una sola funzione d'ingresso (essendo un tipo "passivo").

```
1       class MqttSensor(MqttDevice):
2
3               def __init__(self, addon, **kwargs):
4                       super().__init__(addon, **kwargs)
5                       self._measure: float = None
6                       self.set_property('measure', self._measure_input)
7
8               @property
9               def measure(self):
10                      return self._measure
11
12              @measure.setter
13              def measure(self, value):
14                      self._measure = value
15
16              def _measure_input(self, value: str) -> float:
17                      return float(value)
```

Listato 5.6. Implementazione della classe `MqttSensor`

5.2.2 MqttAddon

L'ultima classe di modulo implementata è `MqttAddon`, riportata nel Listato 5.7. La classe estende `Addon` ed effettua l'*override* dei metodi `on_start` e `on_stop` per sovrascrivere il comportamento durante le fasi del suo ciclo di vita.

In fase di avvio, l'Add-on crea la connessione MQTT (la classe di progettazione `MqttConnection` è stata sostituita con l'uso della classe analoga già esistente

`mqtt.Client` della libreria `Paho-MQTT`). Per ogni `DeviceInstance` registrata e per ogni proprietà di quest'ultima, l'Add-on si mette in ascolto dello `state_topic` definito per la proprietà (ovvero il topic per la ricezione di dati riguardanti la proprietà del dispositivo fisico) e mappa quest'ultimo nella coppia "DeviceID-proprietà".

Per ogni topic su cui l'Add-on si mette in ascolto, viene associato come *callback* il metodo `on_device_event`. Alla ricezione di un messaggio di evento, questo metodo permette di risalire alla `DeviceInstance` da cui è originato e al nome della proprietà associata (attraverso la precedente mappatura eseguita). Il metodo permette, così, di invocare la funzione di input della `DeviceInstance` per la proprietà interessata, con argomento uguale al valore ricevuto nel messaggio di evento.

Al termine del processo, l'Add-on connette il *client* MQTT e avvia il loop di ascolto dei messaggi in ingresso. In fase a di arresto, infine, l'Add-on arresta il loop di ascolto dei messaggi e disconnette il *client* MQTT.

```
1           import paho.mqtt.client as mqtt
2
3           from lib import Addon
4
5           from .configs import *
6           from .device import MqttDevice, MqttLight, MqttSensor
7
8           import logging
9           LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
10          '-35s %(lineno) -5d: %(message)s')
11          LOGGER = logging.getLogger(__name__)
12          logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
13
14
15          class MqttAddon(Addon):
16
17          def __init__(self):
18                  super().__init__(ADDON_ID, light=MqttLight, sensor=MqttSensor)
19                  self._client: mqtt.Client = None
20                  self._topic_device_map = dict()
21
22          def on_device_event(self, _client, _user_data, msg):
23                  device_id, property_name = self._topic_device_map[msg.topic]
24                  device = self._devices.get(device_id)
25                  if device is not None:
26                          assert isinstance(device, MqttDevice)
27                          payload = msg.payload.decode('utf-8')
28                          try:
29                                  if device.parse_input(property_name, payload):
30                                  self._send_device_data(device_id, device.data)
31                          except Exception as e:
32                                  LOGGER.error(e)
33
34          def on_start(self):
35                  self._client = mqtt.Client()
36                  LOGGER.info(f"Connecting MQTT client at broker '{MQTT_HOST}:{MQTT_PORT}'..")
37                  self._client.connect(MQTT_HOST, MQTT_PORT)
38                  for device in self._devices.values():
39                          assert isinstance(device, MqttDevice)
40                          for property_name, property_data in device.properties.items():
41                                  topic = property_data['state_topic']
42                                  self._topic_device_map[topic] = (device.id, property_name)
43                                  self._client.message_callback_add(topic, self.on_device_event)
44                                  self._client.subscribe(topic)
45                  self._client.loop_start()
46                  LOGGER.info(f"MQTT client started")
47
48          def on_stop(self):
49                  self._client.loop_stop()
50                  self._client.disconnect()
51                  LOGGER.info(f"MQTT client disconnected")
52
53          def publish(self, topic, payload):
54                  self._client.publish(topic, payload)
```

Listato 5.7. Implementazione della classe `MqttAddon`

5.3 Script di avvio

Nel Listato 5.8 è riportato lo script di avvio per l'esecuzione del modulo implementato.

Lo script crea un'istanza della classe `MqttAddon` e invoca, all'interno di un blocco per la gestione delle eccezioni, il suo metodo `start` per avviarlo. Alla cattura di un'eccezione, corrispondente ad un interruzione da tastiera o all'uscita del sistema, viene invocato il metodo `stop` per l'arresto dell'Add-on, e l'esecuzione dello script termina.

```
1           from module import MqttAddon
2
3
4           def main():
5                   addon = MqttAddon()
6                   try:
7                           addon.start()
8                   except (KeyboardInterrupt, SystemExit):
9                           addon.stop()
10
11
12          if __name__ == '__main__':
13                  main()
```

Listato 5.8. Implementazione dello script di avvio

6

Il sottosistema per la traduzione dei protocolli e la comunicazione dei dispositivi: validazione

In questo capitolo sarà eseguito il processo di validazione del codice implementato per il modulo di traduzione. Dopo una prima fase di impostazione dell'ambiente di esecuzione, saranno mostrate la configurazione del modulo e la sua esecuzione. Infine, saranno riportate le schermate dei test effettuati per la verifica del corretto funzionamento della soluzione, in conformità con i requisiti previsti.

6.1 Ambiente di esecuzione

Per l'esecuzione del modulo sviluppato e dei test di validità si è scelto di costruire un ambiente containerizzato con tecnologia Docker. Questo sistema permette di incapsulare ogni servizio necessario all'interno di un *container*.

Di seguito saranno mostrati i passaggi seguiti per la configurazione dell'ambiente Docker impiegato, descrivendo i principali servizi che verranno utilizzati in fase di test della soluzione.

6.1.1 Docker

Il primo passaggio per la configurazione dell'ambiente è l'avvio di Docker sulla macchina che ospiterà i servizi in esecuzione. Docker permette, infatti, di utilizzare istruzioni interpretabili dal suo *demone* per manipolare l'ambiente di esecuzione, tra cui l'avvio di *container*.

Ogni *container* da eseguire nell'ambiente Docker necessita di un'*immagine*, ovvero un file, costituito da più strati ottenuti da istruzioni macchina, utilizzato per eseguire codice nel *container* e fornire, al tempo stesso, tutte le risorse software di cui necessita, come interpreti, librerie, ma anche organizzazioni specifiche del *file system*, e tutto ciò che è correlabile alla virtualizzazione del proprio ambiente.

Un'immagine può essere costruita, attraverso l'istruzione `docker build`, a partire da un codice sorgente e da un file chiamato "Dockerfile". Un esempio, riportato nel Listato 6.1, mostra il file utilizzato per la costruzione dell'immagine dei moduli Addon e Hub.

```
FROM python:3.7-slim

RUN mkdir -p /src
WORKDIR /src
COPY . /src

RUN pip3 install -r requirements.txt

CMD ["python", "-u", "./app.py"]
```

Listato 6.1. Dockerfile per la costruzione dell'immagine del modulo `MqttAddon`

Uno dei vantaggi di Docker è la portabilità dell'ambiente di rilascio del software, purché le immagini dei *container* siano state costruite da un'istanza Docker eseguita sullo stesso tipo di architettura hardware (o, comunque, purché le immagini siano compatibili con l'architettura di destinazione). Per l'ambiente utilizzato, è stato impiegato il programma "Docker Desktop", che permette l'esecuzione di un ambiente Docker virtualizzato all'interno del sistema operativo Windows.

6.1.2 Docker-compose

Normalmente, l'avvio di un *container* richiede l'esecuzione, da linea di comando, di istruzioni verbose e, per lo più, difficilmente leggibili all'aumentare dei parametri.

Un strumento molto utile, contenuto all'interno di Docker e chiamato "docker-compose", permette di eseguire *container* in modo molto più comodo e facilmente configurabile. Nel Listato 6.2 è riportato il file `docker-compose.yml` utilizzato per l'esecuzione di tutti i *container* necessari all'ambiente di esecuzione.

Il file, in formato YAML, racchiude le configurazioni di tutti i *container*, con la possibilità di specificare ulteriori automatismi per la generazione e la gestione di volumi (partizioni di memoria dedicate ad uno o più *container*) e reti (sotto-reti virtuali utilizzate dai *container* per la connessione).

```
version: '3.8'
services:
  portainer:
    container_name: portainer
    image: 'portainer/portainer-ce:latest'
    restart: always
    ports:
      - '9000:9000'
      - '8000:8000'
    volumes:
      - '/var/run/docker.sock:/var/run/docker.sock'
      - 'portainer_data:/data'
    networks:
      - gateway
  mosquitto:
    container_name: mosquitto
    hostname: mosquitto
    image: 'eclipse-mosquitto'
    ports:
      - '3883:1883'
      - '9001:9001'
    networks:
      - gateway
  nodered:
    container_name: nodered
    image: 'nodered/node-red'
    ports:
      - '1880:1880'
    volumes:
      - 'node_red_data:/data'
    networks:
      - gateway
  influxdb:
    container_name: influxdb
    hostname: influxdb
    image: 'influxdb'
    ports:
      - '8086:8086'
```

```
39              networks:
40                - gateway
41            grafana:
42              container_name: grafana
43              hostname: grafana
44              image: 'grafana/grafana'
45              ports:
46                - '3000:3000'
47              networks:
48                - gateway
49            hub:
50              container_name: hub
51              hostname: hub
52              image: 'hub:v1.0.0'
53              ports:
54                - '5000:5000'
55              networks:
56                - gateway
57            mqtt_addon:
58              container_name: mqtt_addon
59              hostname: mqtt_addon
60              image: 'mqtt_addon:v1.0.0'
61              networks:
62                - gateway
63          volumes:
64            portainer_data:
65            node_red_data:
66          networks:
67            gateway:
```

Listato 6.2. File di docker-compose per l'avvio dell'ambiente in Docker

Tramite l'istruzione `docker-compose`, eseguita sul file di configurazione, e andando alla pagina `http://localhost:9000` da browser, è possibile accedere all'interfaccia di Portainer (uno dei *container* avviati, nonché interfaccia grafica per la gestione di Docker). Il risultato ottenuto è mostrato in Figura 6.1, dove sono riportati tutti i *container* in esecuzione. Per tali container viene riportata, di seguito, una breve descrizione:

- *Portainer*: interfaccia grafica per la gestione di Docker.
- *Node-Red*: applicazione web per la programmazione di applicazioni *event-driven* per l'IoT.
- *InfluxDB*: database "time series" utilizzato dall'Hub per la memorizzazione di serie temporali dei dati dei dispositivi.
- *Grafana*: applicazione web per la costruzione di dashboard in tempo reale.
- *Mosquitto*: broker MQTT utilizzato per lo scambio di messaggi tra i moduli del gateway multiprotocollo.
- *Hub*: servizio dell'Hub sviluppato per il gateway multiprotocollo.
- *MqttAdddon*: servizio dell'Add-on per l'MQTT sviluppato per il gateway multiprotocollo.

6.1.3 Node-Red

Solitamente è difficile trovare sul mercato dispositivi che nativamente comunichino in protocollo MQTT. Generalmente, quelli che lo supportano sono dispositivi *home-made*, oppure piattaforme che rendono disponibili i dati dei dispositivi che essi gestiscono attraverso messaggi provenienti da un broker MQTT.

Per emulare il comportamento di una piattaforma con dispositivi che ricevono comandi e pubblicano dati attraverso il protocollo MQTT, si è optato per una prototipazione rapida degli stessi utilizzando Node-Red. Esso, infatti, permette di sviluppare un sistema ad eventi attraverso una programmazione a blocchi interconnessi e basata sul flusso dei dati.

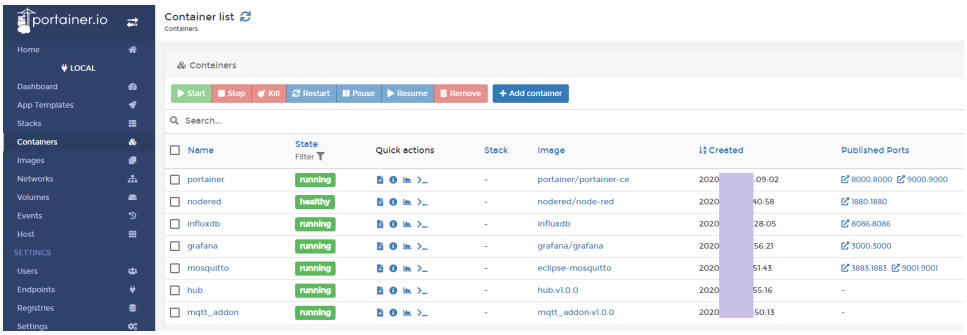

Figura 6.1. Schermata di Portainer per la visualizzazione dei container Docker in esecuzione

In Figura 6.2 è riportata la schermata di Node-Red del flusso di dati creato per simulare l'esistenza di una luce e di un sensore MQTT. Per ogni proprietà della luce, sono stati associati due topic (lo "state topic", per la ricezione delle variazioni di stato, e il "command topic", per l'invio di comandi di modifica della proprietà). Ogni proprietà ha, inoltre, associato un elemento grafico per la costruzione della *dashboard* del dispositivo, che ne permette sia la visualizzazione che il controllo. Questo fa sì che si possano emulare le variazioni di stato del dispositivo utilizzando l'elemento grafico associato ad ogni proprietà, al quale conseguirà la pubblicazione del dato aggiornato sul relativo topic. Analogamente, la ricezione di un evento sul topic di comando permette la modifica del dato a livello grafico sull'interfaccia.

Quanto detto vale anche per il sensore di temperatura, che ha, però, un solo topic associato (quello di stato), e un elemento grafico il cui valore non è controllabile da interfaccia (ma solo visualizzabile). Inoltre, in questo caso, il valore è prodotto automaticamente e casualmente ad una frequenza stabilita.

Il risultato finale della costruzione dell'interfaccia grafica dei dispositivi prototipati è riportato in Figura 6.3.

6.1.4 Grafana

Per la visualizzazione dei dati si è optato per l'uso di Grafana, applicazione web che permette la costruzione rapida di interfacce per la rappresentazione di dati in tempo reale. Grafana, infatti, dispone di un connettore dati per InfluxDB. Attraverso la scrittura di query (in linguaggio *InfluxQL* o con l'editor di Grafana), si dispone di numerosi widget per la rappresentazione del dato, sia in forma di serie temporale che come singolo valore, e di strumenti per la trasformazione dei dati stessi prima della loro visualizzazione.

In Figura 6.4 viene riportato un esempio di query strutturata attraverso l'editor di Grafana, per la lettura dei dati della proprietà "status" dalla misurazione "light" di InfluxDB (la serie associata ai dati delle luci). In Figura 6.5 viene mostrato il risultato della query, assieme a quello delle altre query effettuate per rappresentare, in una forma alternativa a quella di Node-Red, i dati dei dispositivi che si intendono visualizzare.

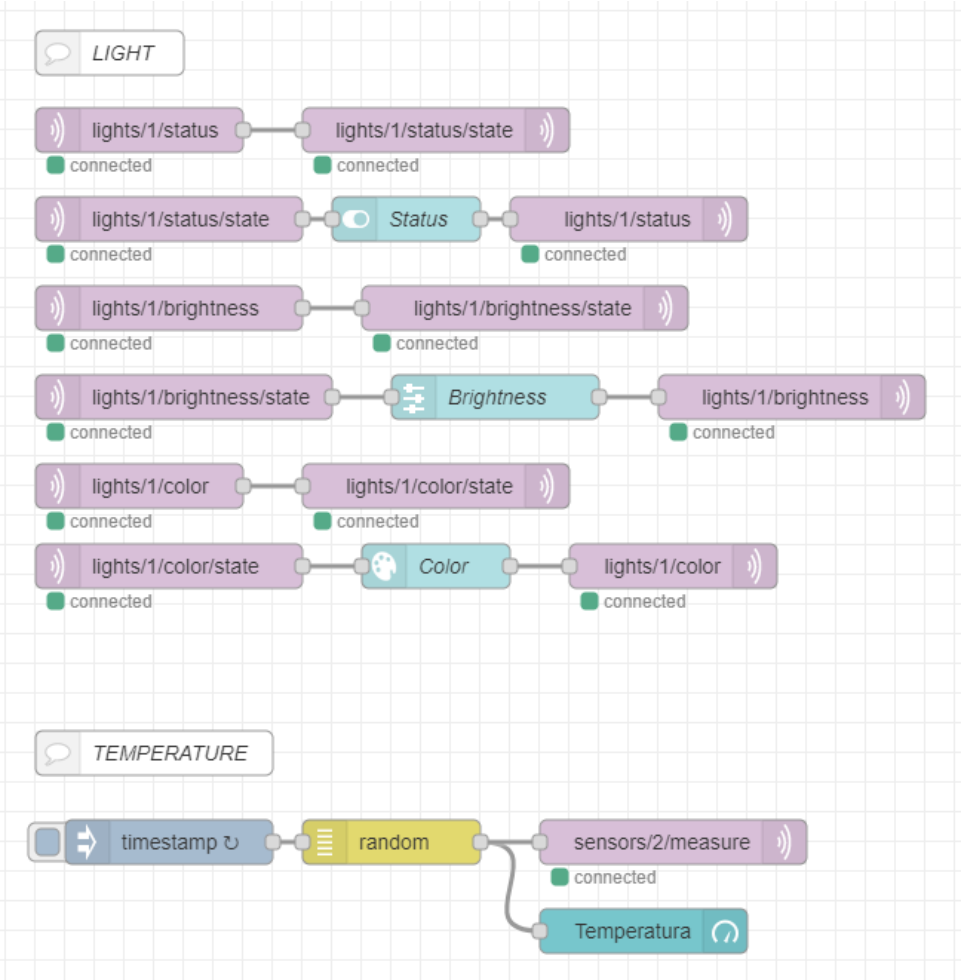

Figura 6.2. Schermata di Node-Red per la prototipazione dei dispositivi MQTT

6.2 Configurazione e avvio dell'Add-on

In questa sezione verranno mostrate le modalità di configurazione e di avvio dell'Add-on MQTT, *containerizzato* all'interno di Docker.

6.2.1 File di configurazione

Nel Listato 6.3 è riportato il file in formato YAML utilizzato come configurazione dei dispositivi MQTT da registrare. Come si può notare, i primi due dispositivi riportati appartengono all'Add-on `mqtt`, mentre l'ultimo appartiene ad un altro Add-on chiamato `modbus`.

```
1           - name: LUCE_SALA
2             type: light
```

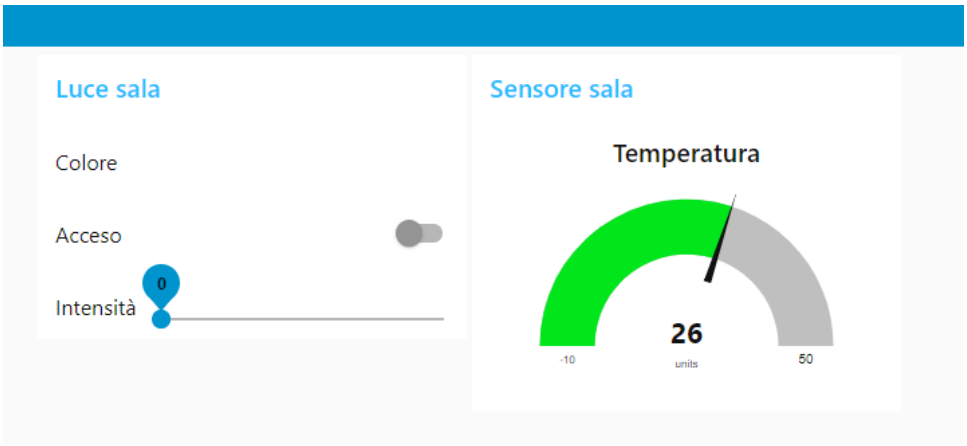

Figura 6.3. Dashboard di Node-Red per l'interazione con i dispositivi MQTT prototipati

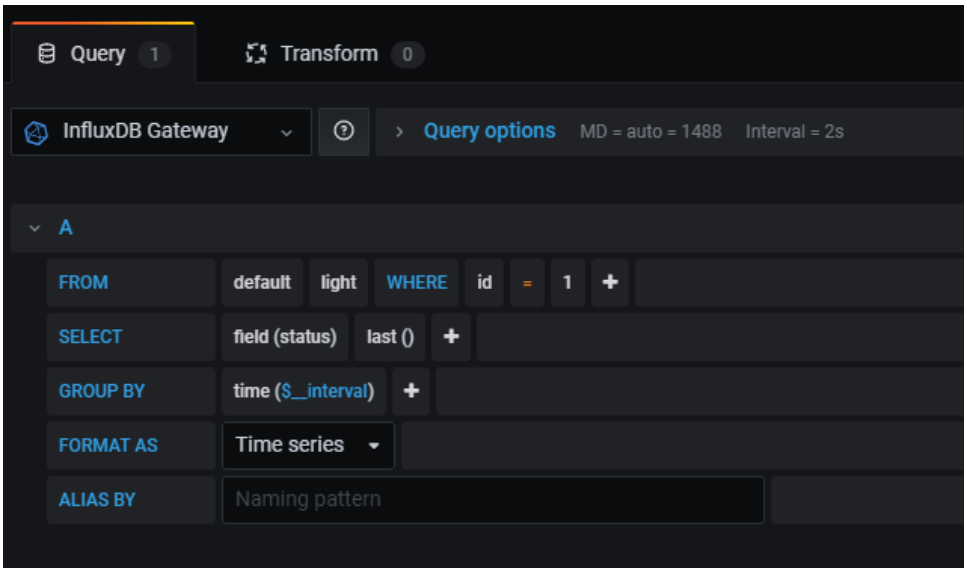

Figura 6.4. Query InfluxQL con l'editor di Grafana per la lettura dello stato di una luce

```
      addon: mqtt
      meta:
        host: mosquitto
        port: 3883
        properties:
          status:
            state_topic: lights/1/status/state
            command_topic: lights/1/status
            payload_on: 'True'
            payload_off: 'False'
          brightness:
            state_topic: lights/1/brightness/state
            command_topic: lights/1/brightness
            scale: 255
          color:
            state_topic: lights/1/color/state
            command_topic: lights/1/color

    - name: TEMPERATURA_SALA
      type: sensor
```

Figura 6.5. Dashboard di Grafana per la visualizzazione dei dati di InfluxDB memorizzati dall'Hub

```
23              addon: mqtt
24              meta:
25                host: mosquitto
26                port: 3883
27                properties:
28                  measure:
29                    state_topic: sensors/2/measure
30
31            - name: LUCE_BAGNO
32              type: light
33              addon: modbus
34              meta:
35                host: 192.168.1.10
36                port: 443
37                slave_id: 1
38                address: 526
```

Listato 6.3. File di configurazione del modulo per la definizione dei dispositivi di test

Il comportamento desiderato dall'Add-on è che, al suo avvio all'interno del container Docker, esso legga e registri i dati solo dei dispositivi che hanno come valore di Add-on soltanto quello corrispondente al proprio ID, ovvero `mqtt`.

Nel file di configurazione, per ogni dispositivo appartenente all'Add-on di riferimento, vengono riportati i seguenti ulteriori parametri:

- *Name*: il nome univoco e *human-friendly* del dispositivo.
- *Type*: il tipo di dispositivo, corrispondente a uno dei tipi supportati dall'Add-on.
- *Meta*: struttura a dizionario, contenente:
  - *Host*: l'indirizzo del broker a cui afferisce il dispositivo;
  - *Port*: la porta del broker a cui afferisce il dispositivo;
  - *Properties*: un dizionario, contenente, per ogni proprietà del tipo di dispositivo, una chiave, i topic di riferimento e ulteriori parametri utili.

6.2.2 Esecuzione del container

In Figura 6.6 viene riportato il log del modulo in esecuzione all'interno del suo container Docker.

Come si può notare, il servizio si connette al Bus e all'Hub, da cui estrae le informazioni. In particolare, vengono registrati solo i dispositivi per l'Add-on `mqtt`, e viene effettuata la connessione di un client al broker Mosquitto presente nella stessa rete a cui è collegato il container dell'Add-on.

```
INFO    2020-        :04:28,274 lib.bus             subscribe       36  : Bus at 'rabbitmq:5672' subscribed to topic 'mqtt.service'
INFO    2020-        :04:28,379 lib.bus             start           23  : Bus at 'rabbitmq:5672' started
INFO    2020-        :04:28,385 lib.utils.hub       init            32  : Hub API initialized at 'hub:5000'
INFO    2020-        :04:28,385 lib.addon           start           33  : Getting addon device from Hub..
INFO    2020-        :04:28,385 lib.addon           start           41  : >> Added light 'LUCE_SALA' (id: 1)
INFO    2020-        :04:28,385 lib.addon           start           41  : >> Added sensor 'TEMPERATURA_SALA' (id: 2)
INFO    2020-        :04:28,385 module.addon        on_start        36  : Connecting MQTT client at broker 'mosquitto:1883'..
INFO    2020-        :04:28,387 module.addon        on_start        46  : MQTT client connected
INFO    2020-        :04:28,388 lib.addon           start           43  : Addon 'mqtt': started
```

Figura 6.6. Schermata di avvio dello script di esecuzione da linea di comando

6.3 Test dei risultati

Per testare il corretto funzionamento del modulo implementato e verificarne la validità, è necessario eseguire un test per ciascuno dei requisiti funzionali implementati, ovvero le funzionalità F04.2 e F05.

I test saranno effettuati a partire da uno stato iniziale noto dei dispositivi presi in esame, e si cercherà di verificare che, in seguito alla procedura applicata, si ottenga lo stato finale desiderato.

La visualizzazione dello stato iniziale, dello stato finale ed di eventuali stati intermedi durante i test sarà verificata nella dashboard di Grafana, che rappresenta graficamente gli ultimi stati delle proprietà di ogni dispositivo, così come essi sono stati memorizzati dall'Hub.

6.3.1 Invocazione di servizi

Il primo test effettuato è l'invocazione di servizi sui dispositivi. Per praticità e chiarezza nell'esecuzione del test, non sarà utilizzata l'API dell'Hub per l'invocazione del servizio, bensì si passerà direttamente dal Bus di comunicazione, sul quale saranno inviati i messaggi così come verrebbero generati dall'Hub.

Il test per l'invocazione dei servizi prenderà come riferimento il dispositivo di tipo "luce" e, come scenario, il diagramma riportato in Figura 2.2. Tale diagramma espone, infatti, un classico esempio di scenario pratico in cui una luce viene portata da uno stato iniziale ad uno stato finale desiderato attraverso l'invocazione di servizi.

Lo stato iniziale scelto è quello della luce spenta, con intensità nulla e colore bianco. Lo stato in questione è riportato in Figura 6.7, dove è mostrata la dashboard di Grafana del dispositivo registrato col nome di "LUCE SALA" al suo stato iniziale.

La prima richiesta di servizio è quella di accensione, che porti la luce da spenta ad accesa con intensità massima. Nel Listato 6.4 viene riportato il contenuto del messaggio per l'attivazione del servizio, inviato sul topic `mqtt.service` in protocollo AMQP.

```
1          # Topic: mqtt.service
2          {"device_id": 1, "properties": {"status": true, "brightness": 100}}
```

Listato 6.4. Contenuto della richiesta di servizio: luce accesa con intensità al 100%

Il risultato della richiesta effettuata viene mostrato in Figura 6.8. Come si può vedere, lo stato di accensione registrato è passato da "Spenta" ad "Accesa", ed il valore percentuale di intensità è stato portato a 100.

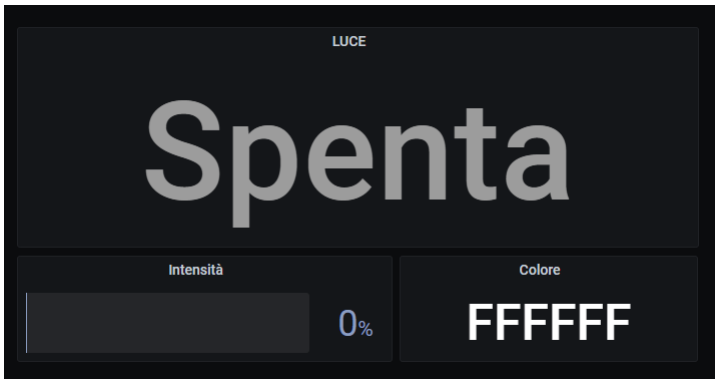

Figura 6.7. Visualizzazione dei dati sulla dashboard Grafana: stato iniziale

Figura 6.8. Visualizzazione dei dati sulla dashboard Grafana: prima richiesta di servizio

La seconda richiesta è, invece, quella di regolazione a intensità media con luce blu. Nel Listato 6.5 viene riportato il contenuto del messaggio per l'attivazione del servizio.

```
# Topic: mqtt.service
{"device_id": 1, "properties": {"brightness": 50, "color": "0000FF"}}
```

Listato 6.5. Contenuto della richiesta di servizio: luce blu con intensità al 50%

Il risultato della richiesta è mostrato in Figura 6.9, dove si può notare che lo stato di accensione è rimasto "Accesa", mentre l'intensità è stata ridotta al 50% e il colore (in rappresentazione RGB esadecimale) è cambiato da bianco a blu.

La soluzione implementata permette, quindi, di:

- *Indirizzare le richieste* al dispositivo desiderato, indipendentemente dall'Add-on di appartenenza (scalabilità) e traducendo l'azione da un messaggio in formato standard (trasparenza).
- *Alterare lo stato* dei dispositivi fisici con variazione delle singole proprietà o con combinazioni di due o più proprietà.

Figura 6.9. Visualizzazione dei dati sulla dashboard Grafana: seconda richiesta di servizio

- *Registrare lo stato* ottenuto dal dispositivo in memoria centrale in formato standard (trasparenza).

Pertanto, il test della funzionalità F04.2 è completato con successo, e i requisiti non funzionali di scalabilità e trasparenza sono soddisfatti.

6.3.2 Sincronizzazione di una modifica del dispositivo fisico

Il secondo test effettuato riguarda la sincronizzazione automatica dei dispositivi dell'Add-on in seguito ad una modifica del dispositivo fisico.

Questa volta è stato preso come dispositivo di riferimento il sensore registrato. Dal momento che il flusso sviluppato in Node-Red emula periodicamente una variazione casuale della temperatura, che viene pubblicata sul topic di stato del dispositivo virtuale, la verifica consiste nell'accertarsi che, ad una modifica del dispositivo sulla UI di Node-Red, questa risulti anche nella dashboard di Grafana (e, quindi, nell'ultimo dato memorizzato dall'Hub).

Le Figure 6.10 e 6.11 mostrano il risultato ottenuto in uno stesso istante di tempo. In questo caso, la soluzione implementata permette di:

- *Leggere una modifica fisica* del dispositivo e intercettarla in tempo reale.
- *Registrare lo stato* ottenuto dal dispositivo in memoria centrale in formato standard (trasparenza).

Pertanto, anche il test della funzionalità F05 è terminato con successo, e la procedura di validazione può considerarsi completata.

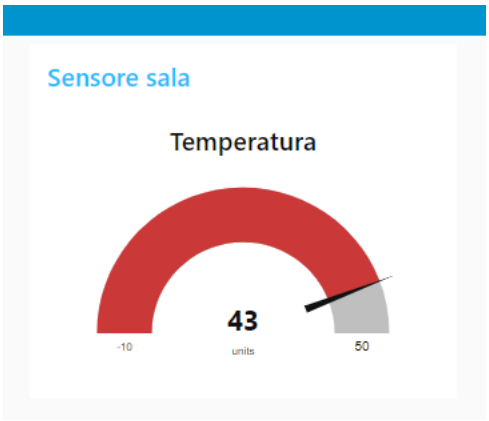

Figura 6.10. Simulazione di dati ottenuti dal campo attraverso la dashboard di Node-Red

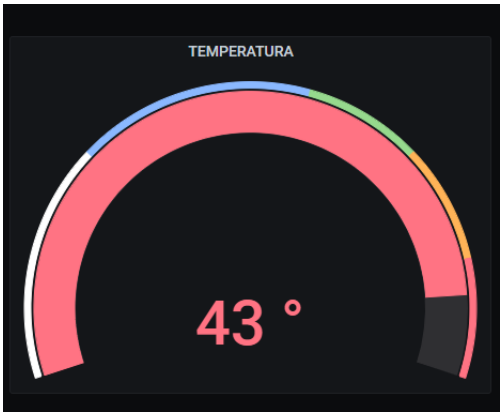

Figura 6.11. Visualizzazione dei dati ottenuti dal campo attraverso la dashboard Grafana

7

Analisi SWOT

In questo capitolo sarà effettuata l'analisi SWOT del sistema realizzato, valutando i punti di forza, le debolezze, le opportunità e le minacce ad esso relativi. Verrà, infine, confrontato il sistema con altri sistemi affini già presenti nel mercato, mettendo in luce analogie e differenze rispetto al sistema realizzato.

7.1 Analisi SWOT

L'analisi SWOT è uno strumento di pianificazione strategica, utilizzato per identificare e valutare i fattori, interni ed esterni ad un progetto o impresa, che possono avere un impatto sul raggiungimento dell'obiettivo.

Il nome dell'analisi è l'acronimo dei quattro punti in cui essa si suddivide, che sono:

- *Strenghts*: i punti di forza interni del progetto o dell'impresa.
- *Weaknesses*: i punti di debolezza interni del progetto o dell'impresa; sono, quindi, gli aspetti da migliorare.
- *Opportunities*: le opportunità esterne che, se adeguatamente sfruttate, offrono al progetto o all'impresa dei vantaggi sul mercato.
- *Threats*: le minacce esterne a cui è esposto il progetto o l'impresa, dai cui effetti negativi è necessario tutelarsi.

Nella Figura 7.1 è riportato un diagramma che schematizza l'analisi SWOT del sistema realizzato. Ciascun punto dell'analisi verrà approfondito in una sezione dedicata di seguito.

7.1.1 Punti di forza

Il gateway multiprotocollo permette una gestione dei dispositivi trasparente rispetto all'utente, ovvero in grado di nascondere la tecnologia sottostante ed esponendo un'interfaccia *human-friendly*, in cui i dati esposti sono strutturati in modo comprensibile all'uomo.

Figura 7.1. Analisi SWOT del sistema realizzato.

Data la separazione tra i moduli di traduzione e il collettore centrale dei dati, il software risulta anche agnostico rispetto alle tecnologie, garantendo il suo funzionamento a prescindere dalla tipologia di dispositivo e l'implementazione di qualunque genere di protocollo di comunicazione.

Lo sviluppo di traduttori è completamente personalizzabile, permettendo di generare autonomamente nuovi moduli di comunicazione, anche con tecnologie *custom*.

Il sistema è facilmente integrabile con qualunque piattaforma esterna o locale, permettendo la lettura di dati e l'invocazione di servizi da software personali o di terze parti attraverso *webhook*, richieste REST o messaggi AMQP.

L'architettura a microservizi permette, infine, la distribuzione fisica dei moduli, consentendo di ospitare parti diverse del sistema in ambienti separati (purché accessibili tra loro), all'interno di macchine reali, virtuali o ibride.

7.1.2 Punti di debolezza

La comunicazione verso il modulo centrale da parte dei traduttori avviene tramite scambio di messaggi che sono agnostici rispetto al linguaggio di programmazione dei moduli, tuttavia lo sviluppo di SDK per i traduttori in linguaggi diversi evidenzia la necessità di manutenere coerentemente ognuno di essi rispetto agli altri ad ogni modifica da apportare.

Il sistema espone API per la gestione ma non presenta un'interfaccia grafica, rendendo complesso l'utilizzo della piattaforma per un utente casuale e non tecnico.

Anche l'installazione del sistema è orientata ad un'utenza con conoscenze tecniche, dal momento che non è disponibile un metodo semplificato di avvio dei moduli all'interno dell'ambiente di esecuzione.

La creazione di moduli di traduzione da parte di uno sviluppatore richiede conoscenze più ampie rispetto al funzionamento dell'intero sistema, essendo l'SDK un modulo software di facilitazione all'integrazione, piuttosto che un *framework* ben definito.

7.1.3 Opportunità

Il gateway multiprotocollo può adattarsi facilmente sia ad ambienti domestici che a contesti industriali. Tuttavia, è proprio nell'industria che può collocarsi come prodotto in grado di fornire benefici e di trarre profitto economico. Il sistema può essere adattato, integrato ed esteso alle esigenze di un qualunque cliente appartenente ad un settore economico che possa trarre vantaggio dall'IoT.

Mettendo a disposizione d'uso libero l'SDK per lo sviluppo di moduli di traduzione e un manuale di utilizzo, il sistema può ricevere il supporto tecnico dell'intera *community* informatica, la quale può contribuire con nuove estensioni e microservizi all'espansione delle sue funzionalità.

Dalla flessibilità del sistema, in termini di integrazione, possono svilupparsi numerose collaborazioni con partner aziendali che integrino la piattaforma col proprio sistema o che forniscano connettori per il loro prodotto, risultando in una partnership produttiva in grado di generare numerosi fronti di estensione nel mercato.

7.1.4 Minacce

Lo scambio di messaggi tra i vari microservizi rappresenta un punto strategico per il sistema e, al tempo stesso, un potenziale rischio per l'immissione di terze parti malevole e attacchi informatici. La comunicazione tra moduli, quindi, è sensibile ad attacchi informatici, che potrebbero compromettere il corretto funzionamento del sistema e permettere la lettura di dati sensibili.

A livello di mercato, la richiesta di soluzioni verticali sulle tecnologie e le piattaforme utilizzate all'interno delle imprese potrebbe ridurre considerevolmente la domanda di un sistema multiprotocollo. Il sistema, infatti, può essere adattato ad ogni specifica esigenza, ma non rappresenta una piattaforma IoT dotata di completa autonomia e interoperabilità. La grande flessibilità del sistema, dunque, può comportare una riduzione della richiesta di mercato se il sistema stesso non è accompagnato da ulteriori middleware per l'analisi e l'elaborazione dei dati.

7.2 Analisi di sistemi affini

Di seguito saranno analizzate le caratteristiche di alcune piattaforme IoT open-source già esistenti, destinate alla domotica e all'automazione.

7.2.1 Home Assistant

Home Assistant è un sistema *open-source* (concesso con licenza Apache 2.0) destinato all'automazione e alla domotica. Dal punto di vista dell'architettura, Home Assistant presenta una serie di componenti logici che comunicano tramite scambio di messaggi, come mostrato in Figura 7.2.

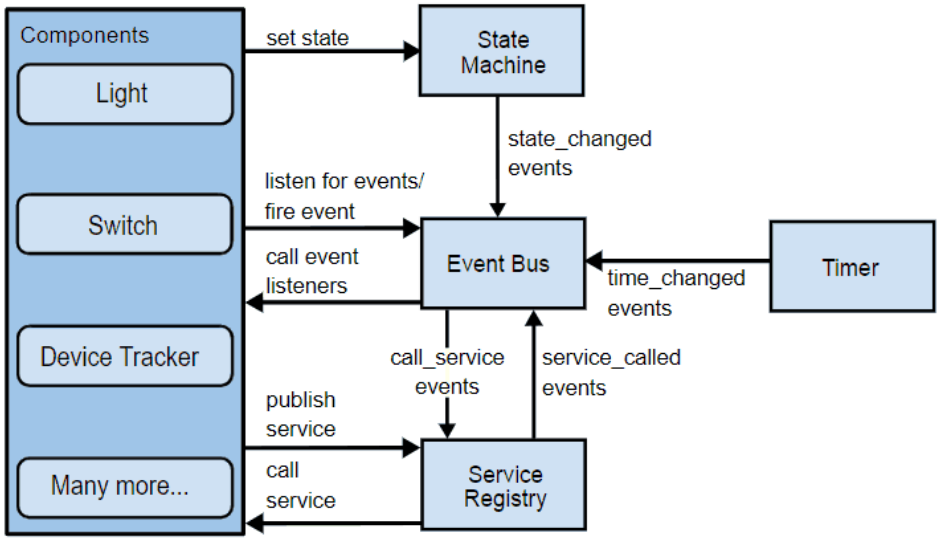

Figura 7.2. Architettura di Home Assistant

Installazione

L'installazione è semplice; può avvenire sia attraverso Docker che scrivendo una copia del sistema operativo (HassBian o Hass.io) all'interno di una SD-card ed eseguendola in una Raspberry Pi. Gli aggiornamenti sono gestiti in modo semplice, attraverso semplici click sull'interfaccia web.

Componenti aggiuntivi

I componenti aggiuntivi per la traduzione possono essere installati attraverso dei *repository*, per i quali la community collabora attivamente per lo sviluppo di nuovi moduli d'integrazione.

Interfaccia

Home Assistant offre un'interfaccia web per la gestione del sistema, simile a quella riportata in Figura 7.3. La vista predefinita mostra tutti i dispositivi e le automazioni configurate, con possibilità di personalizzazione attraverso l'aggiunta di schede e temi diversi.

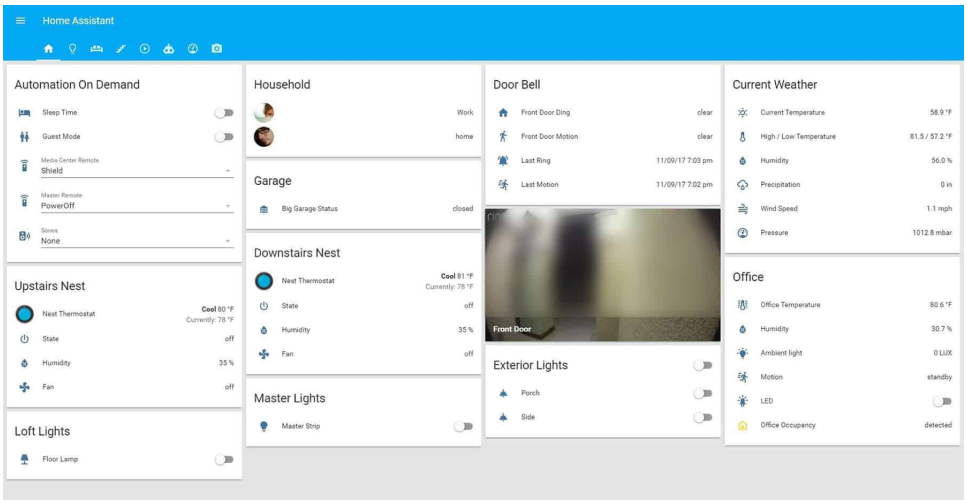

Figura 7.3. Interfaccia web di Home Assistant: gestione del sistema

Configurazione

La prima volta che viene eseguito, il sistema cerca di rilevare tutti i dispositivi connessi alla rete e li aggiunge all'interfaccia. Questa modalità è pratica se si hanno pochi dispositivi, ma per implementazioni più complesse risulta essere piuttosto limitato. Se si desidera eseguire più personalizzazioni, è necessario apportare alcune modifiche ai file di configurazione, scritti in formato YAML.

Regole di automazione

Home Assistant presenta molti modi per creare e modificare le regole di automazione. L'interfaccia web fornisce una sezione per la generazione di regole in modo semplificato, come mostrato in Figura 7.4. L'interfaccia permette di generare automaticamente regole in formato YAML, scritte con una struttura interpretabile dal motore interno di Home Assistant. Le regole, tuttavia, possono anche essere scritte direttamente nel file di configurazione, senza utilizzare l'interfaccia.

Altri modi per generare regole sono *Node-Red,* per il quale esistono nodi costruiti ad-hoc per la comunicazione con gli eventi generati da Home Assistant, e *Appdaemon*, che sfrutta la flessibilità e la potenza di Python per la scrittura di regole più complesse.

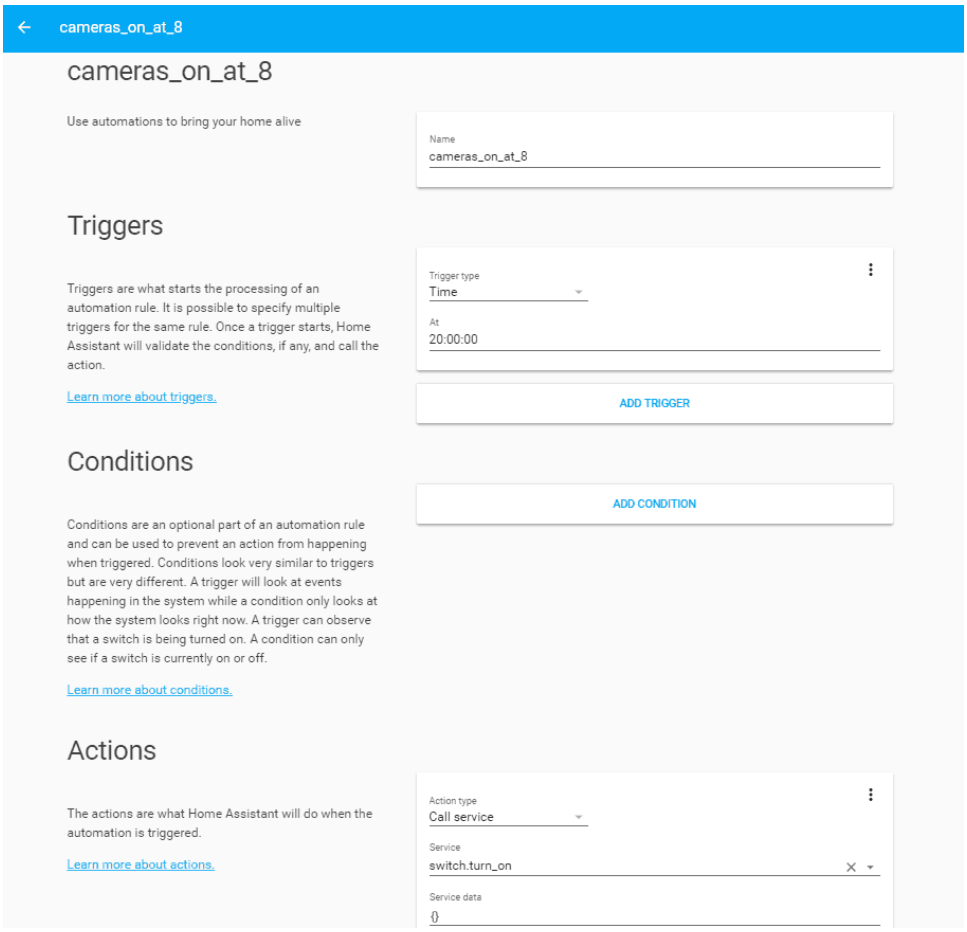

Figura 7.4. Interfaccia web di Home Assistant: costruzione di regole di automazione

7.2.2 OpenHAB

OpenHAB è sviluppato in Java, basato principalmente sul framework Eclipse SmartHome, ed è un software modulare che può essere esteso tramite "Add-on".

Installazione

OpenHAB funziona su Windows, MacOS e Linux. L'installazione può essere effettuata su Raspberry Pi attraverso il sistema operativo OpenHAB (openHABian). In Figura 7.5 è riportata l'architettura software della piattaforma.

Componenti aggiuntivi

I componenti aggiuntivi di openHAB offrono un'ampia gamma di funzionalità, dalle interfacce utente alla capacità di interagire con un numero elevato e crescente di dispositivi.

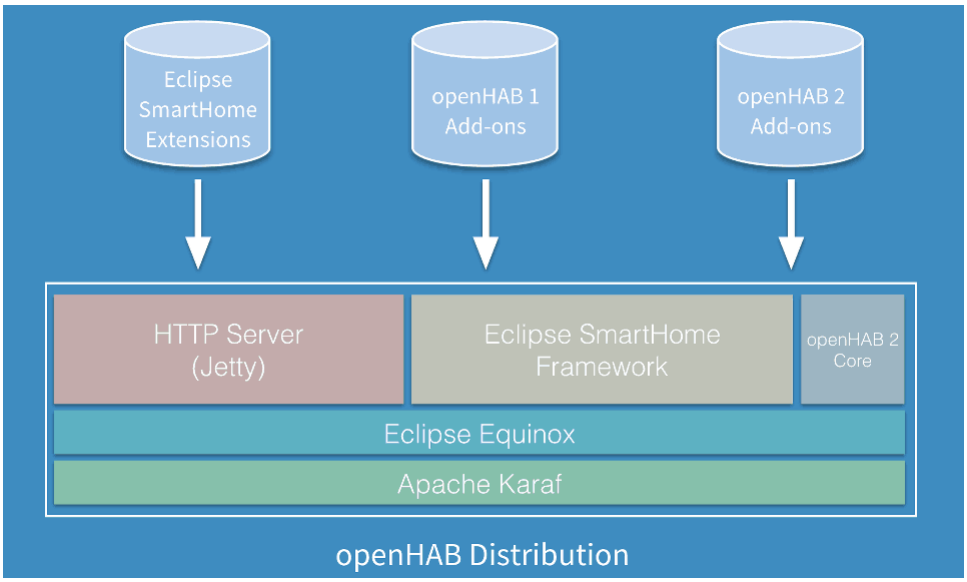

Figura 7.5. Architettura di OpenHAB

Interfaccia

La configurazione iniziale di OpenHAB viene eseguita principalmente utilizzando l'interfaccia web, riportata in Figura 7.6. Essa permette di accedere alla configurazione dei dispositivi e alla gestione del sistema. Purtroppo tale interfaccia non copre tutti i parametri configurabili, quindi è necessario ricorrere a file di configurazione testuali.

Configurazione

Le configurazioni di OpenHAB possono essere gestite attraverso file in formato Xtend, un linguaggio di *scripting* derivato da Java, molto potente, con molte strutture e funzioni complesse disponibili, ma di difficile utilizzo.

Regole di automazione

Anche le routine di automazione (chiamate "regole") possono essere sviluppate attraverso il linguaggio Xtend. Tuttavia, esiste anche un altro modo più semplice, attraverso l'uso di Blockly, una libreria JavaScript per lo sviluppo a blocchi visivi, come mostrato in Figura 7.7.

Applicazione mobile

OpenHAB fornisce, inoltre, un'applicazione Android e una per iOS, che permettono di controllare il server, di ricevere notifiche tramite connessione al cloud, di modificare elementi tramite tag NFC e di inviare comandi vocali.

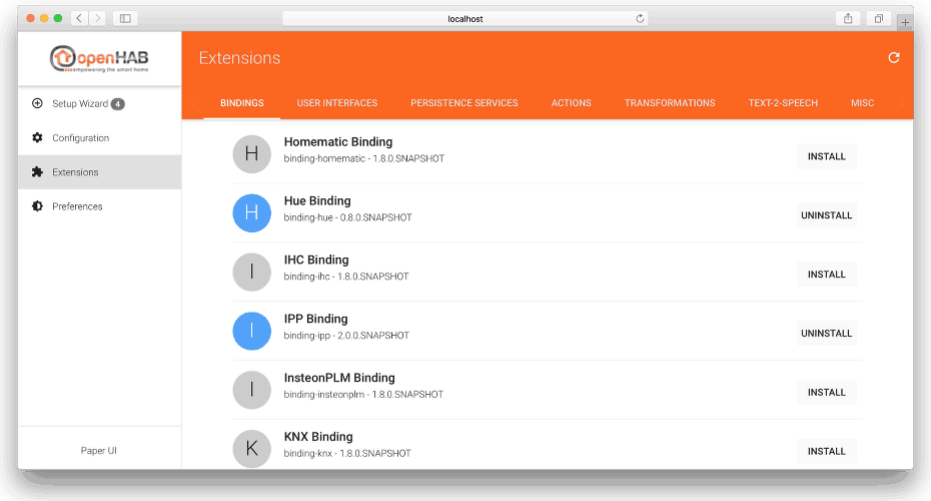

Figura 7.6. Interfaccia web di OpenHAB

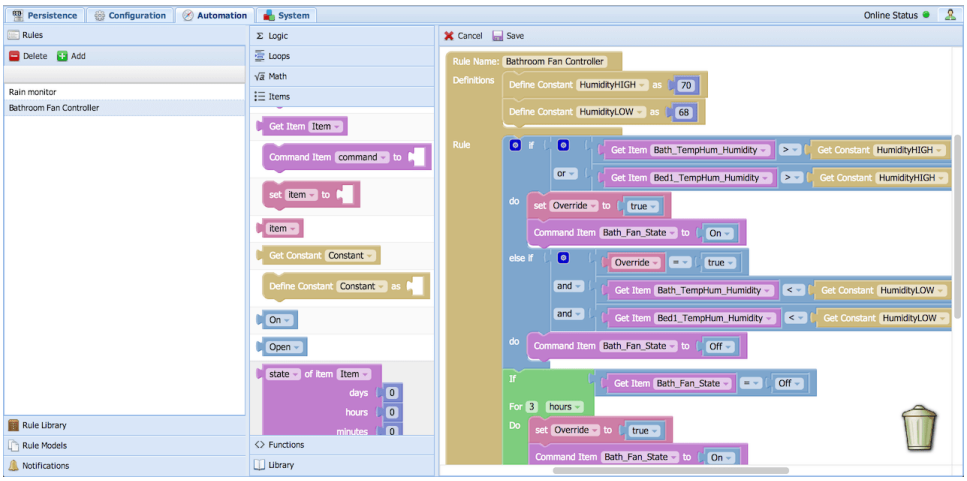

Figura 7.7. Interfaccia di Blockly per la costruzione di regole di automazione

7.3 Analisi strategica

In seguito all'analisi SWOT effettuata sul sistema sviluppato, e a quella dei sistemi affini già esistenti, è possibile confrontare alcune caratteristiche chiave, sia dal punto di vista degli aspetti positivi e negativi riscontrati, sia dal punto di vista dell'esperienza dell'utente.

7.3.1 Analisi della matrice SWOT

A valle dell'analisi SWOT, la matrice costruita può essere utilizzata per individuare le possibili strategie da attuare al fine di migliorare il sistema a livello funzionale e per gestire i punti di forza e i punti deboli a livello di mercato, come esposto di seguito.

Strategie strenght – opportunities

L'elevata personalizzazione del sistema, la sua alta integrabilità con altre piattaforme e l'agnosticismo software rispetto alle tecnologie impiegate possono essere punti chiave su cui concentrarsi per l'inserimento in contesti industriali e in partnership aziendali, attraverso la personalizzare del sistema sulle esigenze del cliente e l'adattamento delle funzionalità a qualunque piattaforma o dispositivo di un *vendor*.

Strategie weaknesses – opportunities

Rendere *open-source* gli SDK e ottenere il supporto della *community* nello sviluppo di librerie e *wrapper* verso vari linguaggi per applicarle a contesti domestici può sopperire alla necessità di mantenere la coerenza tra gli SDK da parte del singolo sviluppatore. Per fare ciò, tuttavia, è necessario fornire una documentazione completa e tecnica del sistema realizzato.

Strategie strenghts – threats

L'alta integrabilità del sistema con altre piattaforme può essere sfruttata per minimizzare il rischio di calo della richiesta verso soluzioni così tanto flessibili. Il sistema, infatti, potrebbe essere sfruttato per la sola integrazione a basso livello, svolgendo, quindi, il ruolo di connettore tra la tecnologia e un altro middleware più verticale e specifico rispetto alle richieste dell'acquirente.

7.3.2 Confronto con i sistemi affini

Il paragone del sistema realizzato con quelli affini già esistenti permette di mettere in luce ulteriori aspetti critici rispetto alla concorrenza sul mercato. Il confronto può essere effettuato sui principali aspetti analizzati delle piattaforme simili, come riportato di seguito.

Installazione

Il sistema realizzato può essere installato su un'infrastruttura Docker, come mostrato durante le fasi di sviluppo e validazione. Tuttavia, rispetto ai sistemi affini, manca la possibilità di essere eseguito sul sistema operativo Linux, fattore che potrebbe incidere sulla sua semplicità di utilizzo e sulla sua versatilità in contesti domestici.

Componenti aggiuntivi

L'integrazione di componenti aggiuntivi non è gestita attraverso dei *repository* dedicati, ma può essere effettuata da qualunque sviluppatore che conosca il sistema realizzato. I moduli di traduzione possono adattarsi a qualunque necessità e la loro aggiunta nel sistema equivale alla costruzione di un'immagine e all'inserimento di quest'ultima nel file di Docker-compose. Tuttavia, dal momento che lo sviluppo si basa su Docker, i *repository* dedicati, presenti in sistemi affini, possono essere sostituiti dall'uso di classici *repository* Docker, pubblici o privati.

Interfaccia

Il sistema realizzato non presenta un'interfaccia, dal momento che fornisce già tutte le risorse per poter realizzare interfacce personalizzate e proprietarie. Un'interfaccia minimale di base, tuttavia, potrebbe risultare utile nelle fasi di installazione e configurazione del sistema, riportando graficamente numerose operazioni disponibili solo attraverso la modifica di file di configurazione.

Configurazione

La configurazione del sistema non necessita di operazioni troppo complesse, tranne in fase di gestione dei dispositivi. Le configurazioni di questi ultimi, infatti, devono riportare una struttura in accordo con quella utilizzata all'interno dei relativi moduli di traduzione. Come anticipato nella sezione di interfaccia, la semplificazione nella configurazione dei dispositivi attraverso una GUI può ridurre la difficoltà di integrazione.

Regole di automazione

Nello sviluppo del sistema, Node-Red è stato utilizzato come strumento per la definizione di regole di automazione. Tuttavia, qualunque sistema in grado di ricevere e inviare dati nel protocollo AMQP può essere utilizzato per definire automazioni, attraverso la ricezione di messaggi dal sistema (per verificare le condizioni di attivazione delle regole) e l'invio di messaggi verso il sistema (per invocare i servizi relativi alle automazioni definite).

8

Conclusioni e uno sguardo al futuro

In questa tesi è stato analizzato lo sviluppo di un generico gateway multi-protocollo destinato a contesti IoT. Sono stati analizzati nel dettaglio i concetti fondamentali alla base del suo funzionamento e, in particolare, le modalità di traduzione e normalizzazione dei protocolli di comunicazione.

In seguito alla descrizione del comportamento desiderato e delle sue principali finalità, sono stati analizzati i requisiti funzionali e non funzionali del sistema in oggetto, con attenzione dedicata alla lettura, alla memorizzazione e alla traduzione dei dati dei dispositivi, attraverso tecniche e metodi propri dell'IoT e dello sviluppo software.

Ogni funzione è stata analizzata e progettata dal punto di vista delle interazioni e dei flussi di dati, fino alla sua strutturazione in classi, che sono state implementate secondo tecniche efficaci di programmazione. Le parti di codice sviluppate sono state, quindi, testate all'interno di un ambiente di esecuzione dedicato, il cui allestimento è stato propriamente descritto. Il funzionamento desiderato del sistema è stato, quindi, validato secondo le specifiche prefissate.

Infine, sono state mostrate le caratteristiche del gateway multiprotocollo sviluppato, effettuando una comparazione di queste con quelle di sistemi affini già esistenti sul mercato. Utilizzando l'analisi SWOT del sistema sono state, quindi, proposte delle strategie di miglioramento del progetto.

Nell'ottica di miglioramenti e sviluppi futuri, il gateway multiprotocollo si presta a numerosi possibili scenari di estensione delle funzionalità e miglioramento dell'esperienza utente complessiva. Tra di essi, uno dei primi è, certamente, l'implementazione di ulteriori interfacce all'interno del pacchetto di API fornito. Ciò permetterebbe di ampliare lo spettro di applicabilità del sistema, fornendo funzioni per la gestione avanzata dei dispositivi, la registrazione di servizi e l'integrazione di un sistema di utenze e accessi controllati.

Inoltre, sarebbe possibile integrare un sistema che permetta l'ascolto di ogni evento significativo, come la creazione o la rimozione di dispositivi, la ricezione di allarmi e notifiche, e molto altro ancora. In questo modo, sarebbe possibile, per terze parti, sviluppare applicazioni che interagiscano in tempo reale con gli eventi del gateway. Un ulteriore sviluppo, quindi, sarebbe il passaggio dei dati *hot* e *cold* verso una piattaforma centrale, attraverso la quale amministrare uno o più gateway multiprotocollo. Tale piattaforma, distribuita sul *cloud*, andrebbe ad arricchire l'e-

sperienza utente attraverso un'interfaccia di gestione interattiva, in grado di rendere più agevole la fruizione del sistema.

Un altro potenziale sviluppo riguarda la creazione di sistemi di *auto-provisioning* per add-on più avanzati. Essi, in base alla tecnologia gestita, potrebbero fornire sistemi di *discovery* e di registrazione automatica dei dispositivi rilevati nella rete. Riguardo l'evoluzione degli add-on è, infine possibile lo sviluppo di un'altra categoria di moduli affine, che gestisca, però, l'interfaccia dei dati verso altri sistemi, piuttosto che la traduzione dei dispositivi reali. Questa variante degli add-on permetterebbe di integrare nativamente le piattaforme IoT più comuni (AWS, Azure, etc) e consentirebbe l'acceso alle informazioni attraverso sistemi di comunicazione noti (SMTP, Telegram, etc), favorendo, così, i principi di integrabilità, trasparenza e scalabilità verso tecnologie e sistemi attuali.

Riferimenti bibliografici

1. U. Breitenbücher, M. Falkenthal, J. Guth, F. Leymann, and L. Reinfurt. Comparison of IoT platform architectures: A field study based on a reference architecture. In *2016 Cloudification of the Internet of Things (CIoT)*.
2. M. Burhan, B. Khan, B. Kim, and R. Rehman. IoT elements, layered architectures and security issues: A comprehensive survey. *Sensors*, 18(9):2796, 2018.
3. H. Derhamy, J. Eliasson, and J. Delsing. IoT interoperability—on-demand and low latency transparent multiprotocol translator. *IEEE Internet of Things Journal*, 4(5):1754–1763, 2017.
4. M. Jung, J. Weidinger, C. Reinisch, W. Kastner, C. Crettaz, A. Olivieri, and Y. Bocchi. A transparent IPv6 multi-protocol gateway to integrate Building Automation Systems in the Internet of Things. In *2012 IEEE International Conference on Green Computing and Communications*, pages 225–233. IEEE, 2012.
5. G. Keramidas, N. Voros, and M. Hübner. *Components and Services for IoT Platforms*. Springer, 2016.
6. I. Lee and K. Lee. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. *Business Horizons*, 58(4):431–440, 2015.
7. J. Macias, H. Pinilla, W. Castellanos, J. Alvarado, and A. Sánchez. Design and Implementation of a Multiprotocol IoT Gateway. *arXiv preprint arXiv:2001.08171*, 2020.
8. S. Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.
9. K. Patel, S. Patel, et al. Internet of things-IOT: definition, characteristics, architecture, enabling technologies, application & future challenges. *International journal of engineering science and computing*, 6(5), 2016.
10. J. Ren, H. Guo, C. Xu, and Y. Zhang. Serving at the edge: A scalable IoT architecture based on transparent computing. *IEEE Network*, 31(5):96–105, 2017.
11. C. Sosa-Reyna, E. Tello-Leal, and D. Lara-Alabazares. Methodology for the model-driven development of service oriented IoT applications. *Journal of Systems Architecture*, 90:15–22, 2018.
12. E. Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.