

RELAZIONE DI PROGETTO DI "SMART CITY E
TECNOLOGIE MOBILI"

Smart Agriculture: l'evoluzione IoT nel settore agricolo

Numero del gruppo:

89

Componenti del gruppo:

Francesco Vignola

Marco Baldassarri

Indice

1	Introduzione	3
2	Stato dell'arte	5
3	Analisi dei requisiti	8
3.1	Requisiti Funzionali	8
3.2	Requisiti Non Funzionali	10
4	Analisi di Mercato	12
4.1	Drone	12
4.2	Gateway	12
4.3	IoT Devices	13
4.4	AWS IoT Core e NoSQL database	14
5	Progettazione	16
5.1	Architettura del Sistema	16
5.1.1	Diagrammi dei Casi d'Uso	17
5.1.2	Interazione dei Componenti	21
5.2	Clients	24
5.3	Comunicazione	25
5.3.1	Introduzione a MQTT	26
5.3.2	Comunicazione MQTT	28
5.3.3	Introduzione a BLE	31
5.3.4	Comunicazione BLE:	32
5.4	Micro-irrigation	35
5.4.1	Comportamento	35
5.4.2	Struttura	37
5.4.3	Interazione	40
5.5	Circuito elettronico	41
5.5.1	Partitore di Tensione	41
5.5.2	Rilevazione dell'intensità luminosa	43
5.5.3	Rilevazione del livello di batteria	44
5.5.4	Diagramma del prototipo	45

6	Implementazione	46
6.1	Irrigazione	46
6.1.1	IoT Device	46
6.1.2	Gateway	53
6.1.3	AWS Lambda	56
6.2	Fertilizzazione con Drone	58
6.3	Videosorveglianza:	61
6.4	Telegram Client	64
6.5	Alexa Skill Client	66
7	Testing e performance	70
7.1	Volume Dati	71
7.2	Comunicazione	72
8	Analisi di deployment su larga scala	74
9	Piano di lavoro	77
10	Conclusioni	81
	Riferimenti bibliografici	82

1 Introduzione

Negli ultimi anni il settore agricolo fronteggia numerose sfide legate alla rapida crescita della popolazione mondiale e alle esigenze del nuovo consumatore. Tale crescita, da un miliardo nel 1800 a 7 miliardi nel 2017 con una stima di 9,6 miliardi nel 2050, innescherà una carenza di cibo evitabile solo con l'apporto di innovazioni significative nel settore agricolo. I principali problemi sono l'aumento dei prezzi dell'energia, il calo dei livelli delle acque sotterranee a causa dell'estrazione e dell'irrigazione insostenibili, la scarsità di terreni coltivabili e la crescente regolamentazione da parte delle autorità. L'Internet of Things, in sinergia con altre tecnologie, potrebbe essere utilizzato per massimizzare i rendimenti e minimizzare i costi, garantire una qualità costante del prodotto, ottimizzare il consumo, e minimizzando lo spreco, di acqua e ridurre al minimo l'impronta ambientale di un'azienda agricola e anche dell'intera filiera.

L'agricoltura, pur con alti e bassi, è da sempre uno dei settori primari che guida lo sviluppo economico mondiale e si è evoluta di pari passo con le diverse rivoluzioni industriali. Difatti, oggigiorno, si parla di Agricoltura 4.0 e Agricoltura Sostenibile. Con Agricoltura 4.0 si intende un nuovo modo di pensare le attività di coltivazioni, guidate dai dati raccolti sul campo grazie all'installazione di dispositivi e sensori connessi tra loro e con la rete. Questi dispositivi inviano i dati nel cloud, dove sofisticati modelli analitici prendono decisioni, aiutando l'agricoltore ad ottimizzare la qualità del prodotto.

In realtà, si parla anche di una nuova rivoluzione che porterà all'Agricoltura 5.0 in cui sarà predominante l'uso dei robot e dei co-bot come aiuto all'agricoltore nelle attività di coltivazione, in quanto avranno la capacità di operare in autonomia o con una guida limitata. [12]

Con Agricoltura Sostenibile, invece, si pone l'accento sull'impatto ambientale delle attività di coltivazioni, e sull'uso delle tecnologie per salvaguardare l'ambiente. Il progetto presentato in questa relazione rientra nell'ambito dell'Agricoltura 4.0 e dell'Agricoltura Sostenibile, in quanto viene affrontato il problema della gestione delle colture con particolare attenzione alla problematica della scarsità di risorse idriche nel sottosuolo.

La gestione delle colture si basa sull'osservazione, la misurazione e il controllo delle colture e del terreno. In tal modo gli agricoltori possono prendere decisioni guidate

dai dati reali meteorologici, ottenuti dall’analisi dell’aria e del suolo. IBM stima che il 90% dei fallimenti delle colture siano dovuti alle condizioni meteorologiche e l’uso di opportune tecnologie potrebbe ridurre la percentuale fino al 25%. [7] Dal punto di vista tecnologico, la problematica affrontata riguarda la raccolta dei dati dai sensori posti direttamente sul campo, la loro condivisione ed elaborazione su vasta scala sfruttando il cloud. La soluzione proposta ha anche un impatto economico ridotto, grazie all’uso di device a basso costo e di applicazioni mobile liberamente scaricabili, permettendo agli agricoltori di innovare tecnologicamente la loro azienda agricola con bassi investimenti. Di fondamentale importanza si è ritenuto anche preservare le colture da eventuali atti dolosi, come il furto, realizzando un sistema di rilevamento delle intrusioni.

2 Stato dell'arte

La Smart Agriculture, in italiano spesso ribattezzata con il nome di “Agricoltura 4.0”, mira innanzitutto a rendere i processi agricoli più efficienti e sostenibili attraverso l’utilizzo di sistemi hardware e software interconnessi e in grado di fornire un supporto alle decisioni dell’esperto del settore o talvolta anche a prendere decisioni al posto dell’uomo stesso. Si è potuto constatare quindi come l’uso di tecnologie immerse nell’ambiente agricolo aiuti l’utente a minimizzare l’impatto energetico ed ambientale in termini di risorse idriche ed elettriche (oggi sempre più preziose e tendenti all’esaurimento) nonché a facilitare la gestione di campi agricoli di grandi dimensioni per un utilizzo industriale. Oltre ai benefici ambientali, alcuni ricercatori hanno potuto anche constatare evidenti miglioramenti anche in termini di qualità dei prodotti alimentari coltivati, qualità che corrisponde a benefici dal punto di vista della salute del consumatore finale.

Analizzando la letteratura e il mercato inerente la Smart Agriculture sono state individuate differenti soluzioni nell'affrontare il problema, sia dal punto di vista infrastrutturale, da semplici sistemi embedded a sistemi più complessi che fanno uso del cloud, che per la logica con cui si irriga, la quale influenza la scelta di opportuni sensori. Molto utilizzati sono i sensori di temperatura e di umidità, che non solo giocano un ruolo fondamentale nel monitoraggio delle condizioni ambientali, ma sono molto apprezzati in qualsiasi contesto applicativo per il minore/minimo impatto che hanno sull’ambiente. Non meno importanti sono le Weather Stations, le quali se impiantate nel terreno permettono di raccogliere dati per fare previsioni sulle condizioni meteo delle prossime ore. E’ possibile anche monitorare, indirettamente, i bisogni delle piante tramite sensori che rilevano la quantità di acqua presente nel terreno, meglio conosciuti come sensori di umidità del terreno, e sensori che rilevano l’intensità di luce. Se il terreno è umido, anche solo parzialmente, si è dimostrato che non è necessario irrigare in quanto il quantitativo di acqua fornito è maggiore di quello necessario per le piante. Mentre, nelle giornate molto calde e lunghe come quelle estive, irrigando nelle ore più luminose si avrebbe il fenomeno di evaporazione dell’acqua, per cui l’acqua emessa non verrebbe assorbita dalle piante, causando non solo lo spreco di acqua ma anche la disidratazione delle piante.

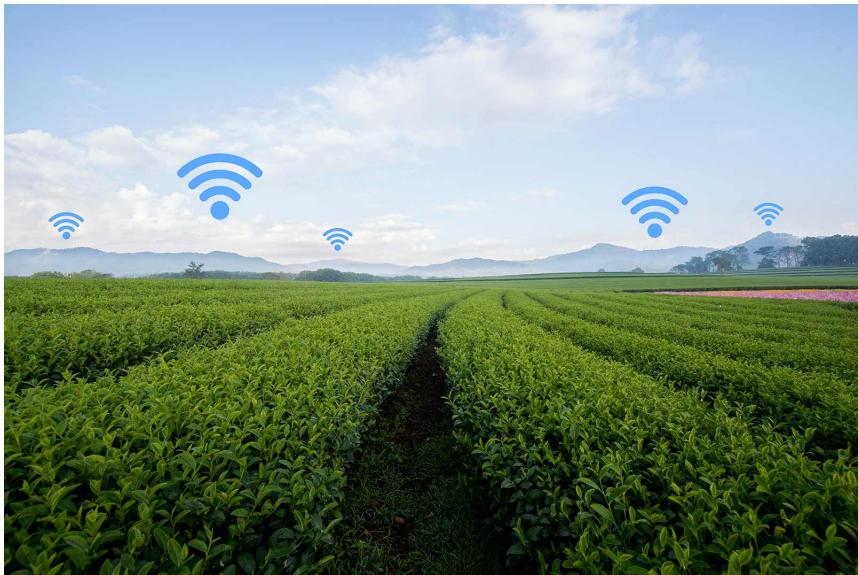


Figura 1: Smart Farming concept

Lo stato di salute delle piante è fondamentale, ma senza un buon terreno, fertile, non hanno modo di crescere. La fertilizzazione può essere applicata con diverse tecniche, dall’immissione di fertilizzante insieme all’acqua (fertilirrigazione) a sistemi di fertilizzazione a rateo variabile basati sull’utilizzo di droni. Quest’ultima soluzione è molto flessibile perché, nei casi in cui in un campo vi siano diverse piantagioni, permette di utilizzare un quantitativo di fertilizzante adatto alla particolare piantagione. [6] L’uso di un Drone facilita la vita dell’agricoltore il quale può risparmiare tempo nel trattamento, e permette anche una risposta più immediata alle minacce di insetti, erbe selvatiche e malattie. Dei droni vengono sfruttate anche le telecamere integrate per fini di sorveglianza sui campi, ma anche per stimare le caratteristiche di un campo e monitorare lo stato delle piante attraverso algoritmi di Image Processing.

Le soluzioni più complete adottano anche sistemi di rilevamento di intrusione attraverso apposite telecamere fissate su punti strategici del campo per proteggere le colture da mani indiscrete. Anche questo scenario è stato esplorato in modo variegato, dall’applicazione di sofisticati algoritmi per il rilevamento di una figura umana al semplice rilevamento della variazione della radiazione ad infrarossi irradiati dagli oggetti circostanti.

Le caratteristiche che accomunano i sensori per l’agricoltura presenti in commercio

sono senza dubbio il basso consumo energetico (batterie efficienti usate in locazioni geografiche ove risulta difficile portare la rete elettrica) e l'introduzione di architetture di Rete per la comunicazione ad ampio raggio, ottimizzate per il contesto IoT. La comunicazione tra gli edge devices si basa su tecnologie wireless a radiofrequenza, tra cui la più rinomata è LoRa, e una versione ottimizzata a basso consumo della rete cellulare 4G.

3 Analisi dei requisiti

Il progetto consiste nella realizzazione di un sistema di Smart Irrigation con l’obiettivo di ottimizzare il consumo di acqua, minimizzando gli sprechi. Il tutto preservando lo stato di salute delle colture. Un altro importante scopo del progetto è quello di facilitare la vita all’agricoltore rendendo automatizzabili i processi di irrigazione (e quindi la scelta del quando irrigare) e di fertilizzazione. Più precisamente, il sistema deve irrigare solo qualora vi siano determinate condizioni ambientali. Ovvero, il terreno deve essere secco e la temperatura e la luminosità non devono assumere valori elevati per evitare il fenomeno dell’evaporazione dell’acqua che porterebbe alla disidratazione delle piante. E’ fondamentale, anche, non irrigare se la giornata è piovosa in modo da ridurre ulteriormente lo spreco di acqua e danneggiare le piante fornendo molta acqua.

Inoltre, i parametri ambientali, rilevati durante l’intera giornata, devono essere resi disponibili all’utente per effettuare analisi più dettagliate al fine di individuare il momento giusto in cui è opportuno irrigare e per valutare lo stato delle colture. Il sistema di Irrigazione intelligente deve interfacciarsi anche con un aeromobile a pilotaggio remoto, utile per fertilizzare le colture tramite emissione di goccioline di fertilizzante da spargere direttamente sopra al terreno. Il Drone deve, quindi, sorvolare tutto il campo ed essere in grado di ritornare alla base, sia dopo aver terminato la propria attività sia nel caso in cui il livello di batteria non è sufficiente. Infine, è necessario realizzare anche un piccolo sistema di allarme che rileva la presenza di eventuali intrusioni non autorizzate all’interno del campo al fine di evitare o ridurre il furto delle colture, evitando così perdite economiche ingenti all’utente finale. L’agricoltore dovrà poter monitorare i dati raccolti e controllare lo stato di batteria sia del Drone che dei sensori immersi nel terreno.

3.1 Requisiti Funzionali

1. Il sistema deve monitorare periodicamente i parametri ambientali quali: umidità del terreno, umidità e temperatura dell’aria, precipitazioni, luminosità solare.
2. Il sistema deve irrigare, per un periodo di tempo prefissato, se e solo se:
 - il terreno è secco (umidità <40%)

- la temperatura dell'aria non è elevata ($<27^{\circ}\text{ C}$)
 - la luminosità solare non è elevata ($<30\%$)
 - non viene rilevata alcuna precipitazione in atto
 - le previsioni meteo non prevedono precipitazioni nell'arco delle prossime X ore.
3. Il sistema deve fertilizzare le colture non più di una volta al giorno.
 4. Il sistema deve gestire anche più di un campo di coltura con un proprio sistema di irrigazione.
 5. Il sistema deve rilevare eventuali intrusioni umane e inviare una notifica di allarme sul device dell'utente. L'allarme non deve essere azionato in caso di rilevazione di oggetti diversi da quello di forma umana.
 6. Il sistema deve monitorare lo stato della batteria dei sensori e del Drone.
 7. I sensori dovranno essere immersi nel terreno in punti strategici del campo agricolo in cui potrebbe non essere presente la corrente elettrica
 8. Si dovrà gestire la persistenza dei dati acquisiti dai vari sensori al fine di renderli facilmente fruibili all'utente per eventuali analisi personali
 9. L'utente deve poter lanciare il comando di irrigazione (tramite pompa d'acqua) e fertilizzazione (tramite Drone) da remoto in modo da garantire una gestione anche manuale. L'intervento manuale non deve inibire il funzionamento automatico del sistema.
 10. L'utente deve poter inviare i comandi manuali senza l'utilizzo di un device ma bensì tramite un sistema di riconoscimento vocale per permettere un'interazione *hands-free*.
 11. La pompa si blocca dopo un periodo ben definito in modo autonomo. Quallora la pompa ricevesse il comando di irrigazione, e la pompa è già attiva, allora si riavvia il timer.

3.2 Requisiti Non Funzionali

1. Il sistema di Smart Irrigation deve fornire i dati in real-time in modo che l'utente possa effettuare le proprie analisi con dati costantemente aggiornati. Essendo la mole dei dati sensoriali molto elevata, il sistema deve essere scalabile e fault-tolerance.
2. E' da considerare anche eventuali disconnessioni dalla rete internet dei componenti, in modo da garantire la disponibilità dei dati. Se il sistema rimane isolato per molto tempo, i dati rilevati dai sensori devono comunque essere resi persistenti e accessibili successivamente quando il sistema ritorna online. Questo requisito garantisce all'utente di effettuare le proprie analisi in modo accurato su un set di dati reali.
3. A fronte di cadute di rete internet i dati devono continuare ad essere resi persistenti per mantenerne la consistenza e la freschezza. In questo modo i dati potranno tornare accessibili anche una volta che la rete torna a funzionare.
4. Nella stessa zona del campo agricolo devono essere presenti più sensori con le stesse caratteristiche in modo da garantire la fault-tolerance anche a livello di sensori e quindi migliorare l'accuratezza dei dati.
5. Si deve realizzare il sistema in modo che sia modulare, ovvero deve essere possibile aggiungere in futuro ulteriori Edge devices (aventi gli stessi sensori) che in modalità *plug-and-play* si agganciano alla stessa unità di controllo presente sul campo agricolo.
6. Nel caso di campi estesi con più sistemi di irrigazione deve essere possibile la suddivisione in più zone, ciascuna dotata di una propria unità di controllo.
7. I sensori nel terreno devono essere pensati per minimizzare il consumo energetico, devono quindi implementare un metodo di comunicazione efficiente. Il sistema deve, quindi, essere facilmente installabile in campi estesi e deve essere garantita l'autonomia (tramite l'uso di batterie a litio) anche in punti in cui la rete elettrica non è sempre presente, in modo da evitare manutenzione continua e costosa.

- Il sistema deve essere scalabile e sicuro. In particolare è necessario gestire in real-time una mole di dati sensoriali elevata e deve essere possibile aggiungere ulteriori sensori e dividere il campo in sotto-campi, ognuno con il proprio sotto-sistema che agisce in modo autonomo.

4 Analisi di Mercato

In questi paragrafi si vuole dare una descrizione più dettagliata dei dispositivi fisici o più in generale delle entità scelte per la realizzazione del progetto.

4.1 Drone

L’idea di utilizzare un drone per simulare la fertilizzazione proviene dai contesti applicativi realmente esistenti in commercio: al di sotto del veivolo viene di solito applicata una tanica dal carico massimo consentito contenente il fertilizzante che verrà rilasciato sulle piante al passaggio.

Una famosa azienda produttrice di droni per la Smart Agriculture è DJI, azienda leader del settore dell’aviazione a pilotaggio remoto. Col desiderio di voler prototipare il più possibile un contesto reale, si è optato per la famiglia *Tello*, drone della stessa marca sopra citata, ma al contempo economicamente più accessibile e programmabile in qualsiasi linguaggio.

Tello permette di effettuare un percorso preimpostato dai comandi impartiti via codice, nonché di spedire lui stesso delle informazioni telemetriche sul suo stato corrente. La comunicazione tra il controller e il drone verrà spiegata più in dettaglio nei prossimi paragrafi.

Per “programmabile” si intende quindi, in soldoni, la possibilità di sostituire il joystick o l’app ufficiale rilasciati in dotazione all’acquisto con un client qualsiasi che funge da controller e interagisce spedendo dei comandi presi dal set di operazioni rese disponibili dalla casa produttrice.

4.2 Gateway

Si tratta del dispositivo che può essere collocato nel Fog Layer e fa da tramite tra i sensori presenti sul campo agricolo e il Cloud. Questo meccanismo ha portato a ribattezzare questo componente col nome di “Gateway”, proprio per dare l’idea di un’entità che fa da tramite tra un layer e l’altro.

Le sue funzioni principali che è chiamato a svolgere questo importante componente sono:

1. Spedire all’Edge il comando di irrigazione a fronte di una richiesta manuale da parte dell’utente o richiesta automatica da parte di AWS Lambda.

2. Acquisire informazioni dall'ESP32, immagazzinare in un database relazionale locale i dati salvati e spedire in Cloud una versione semi aggregata di questi nel momento opportuno (ad intervalli regolari).

A quel punto AWS si occuperà di renderli persistenti e di gestirli successivamente tramite AWS Lambda.

3. Gestire la comunicazione da e verso il drone e quindi implementare tutta la logica di comando del veivolo a fronte della richiesta di irrigazione

Come gateway abbiamo immaginato un dispositivo connesso alla rete elettrica e alla rete internet, posizionato in una posizione strategica del campo agricolo per consentire anche un'adeguata comunicazione con i vari Edge device.

E' venuta naturale la scelta di un Raspberry Pi, realizzato in UK dalla Raspberry Pi Foundation. Si tratta di un single-board computer delle dimensioni ridotte e dal prezzo contenuto. Nel corso degli anni sono state rilasciate diverse revisioni fino ad arrivare all'attuale versione 4. Per il nostro progetto abbiamo avuto la necessità di una potenza di calcolo discreta e della presenza nativa sia del modulo Bluetooth che del modulo WiFi, per consentire una comunicazione bidirezionale con Tello Drone e ESP32. La scelta non poteva che ricadere quindi sull'ultima versione: ver. 4 model B a 4 GB di RAM / 16 GB memoria.

4.3 IoT Devices

L'IoT Device è il dispositivo che più si interfaccia con il mondo fisico, in quanto pensato per essere installato direttamente nel terreno. È equipaggiato di sensori ed attuatori utili per rilevare i parametri ambientali e per irrigare.

L'IoT Device si basa su una scheda potente e generica ESP32-WROOM32, dotata di Wi-Fi, BLE MCU e Classic Bluetooth. Grazie a questa scheda è possibile realizzare una grande varietà di applicazioni, dalle reti di sensori a bassa potenza allo streaming musicale.

Lo scelta di utilizzare ESP32 piuttosto che Arduino o altre schede di prototipazione, è dovuto all'ottimo trade-off caratteristiche tecniche/prezzo. Infatti, con soli una decina di euro si ha una scheda che incorpora un chip progettato per essere scalabile e adattivo. È dotato di due core ed un co-processore a bassa potenza

che può essere utilizzato al posto della CPU per risparmiare energia quando non è richiesta molta potenza di calcolo.

Lato software, ESP32 sfrutta il sistema operativo FreeRTOS che permette di realizzare applicazioni multi-tasking e multi-core, sfruttando le potenzialità dell'hardware sottostante. A tutto ciò si affianca la possibilità di utilizzare il framework Arduino ed il linguaggio C++ semplificando l'accesso all'hardware ed utilizzando astrazioni di alto livello, partendo dal paradigma ad oggetti.

4.4 AWS IoT Core e NoSQL database

AWS IoT Core è una piattaforma cloud gestita, in grado di connettere bilioni di device in modo sicuro ed affidabile. L'uso di AWS IoT Core rappresenta una scelta strategica in quanto permette di avere tutti i vantaggi offerti da un MOM (*Message-Oriented Middleware*) quali comunicazione asincrona degli endpoint con differenti livelli di qualità del servizio (QoS) grazie all'utilizzo delle *Message Queue*. In aggiunta, offre tutti i benefici del cloud, infatti AWS IoT Core non è un semplice MOM, ma bensì può essere visto, esso stesso, come un servizio che abilita la comunicazione con differenti protocolli: MQTT, MQTT over WebSocket e HTTPS. **MQTT** rappresenta ormai lo standard de facto per la comunicazione nell'ambito IoT perchè è progettato per garantire il trasporto di messaggi estremamente leggero e basato sull'architettura *Publish&Subscribe*, con un consumo minimo della larghezza di banda della rete.

Nel progetto in esame si devono connettere il gateway con il Telegram Bot e Alexa in modo bidirezionale, per cui si è scelto l'uso di AWS IoT Core con protocollo MQTT per la sua leggerezza, e perché permette l'integrazione con Alexa, attraverso l'uso delle AWS Lambda.

Infine, ma non per ultimo, AWS IoT Core, ed in generale l'intera piattaforma cloud Amazon garantisce scalabilità on demand, evitando l'acquisto di server che potrebbero rilevarsi non necessari, e al contrario permette di incrementare le risorse in uso senza la complessità di gestione.

La scelta della piattaforma Amazon è accentuata dalla necessità di avere un database online in cui memorizzare i dati proveniente dai device IoT, in modo da poterli aggregare e analizzare, al fine di prendere la decisione di irrigare. Questo è reso semplice da AWS perchè fornisce l'utilizzo di DynamoDB come servizio.

DynamoDB è un database NoSQL che, in quanto tale, è progettato per fornire molti schemi di accesso ai dati che includono applicazioni a bassa latenza. In particolare, i database NoSQL offrono una varietà di modelli di dati come chiave-valore, documento e grafo ottimizzati per prestazioni e scalabilità. Infatti sono molto utilizzati in scenari in cui vi è un elevato throughput e bassa latenza con requisiti di scalabilità orizzontale (più di un'istanza). DynamoDB è un database chiave-valore e basato su documento, in grado di gestire più di 10 trilioni di richieste al giorno e può supportare più di 20 milioni di richieste per secondo.

5 Progettazione

5.1 Architettura del Sistema

Per l'analisi dell'architettura del sistema sono stati prodotti il diagramma dei componenti, utile a definire i sistemi in cui è suddiviso il progetto, il diagramma di sequenza che definisce la comunicazione tra i vari sistemi e il diagramma dei casi d'uso per evidenziare le funzionalità principali di ogni sistema.

Dall'analisi dei requisiti è emersa la necessità di avere:

- un IoT Device per la rilevazione dei parametri ambientali e per l'avvio dell'irrigazione, situato direttamente sul campo;
- un Gateway necessario per connettere in rete gli IoT Device, svincolandoli dall'onere di dover utilizzare una tecnologia heavyweight (come il WiFi) per ottimizzare l'uso della batteria, ma che può essere visto anche come un aggregatore di dati da inviare in cloud in quantità ridotta col fine di limitarne i costi di gestione;
- Un sistema di pilotaggio automatico da parte del Drone per fertilizzare;
- Un sistema di videosorveglianza direttamente installato sul campo agricolo;
- Message-Oriented Middleware per integrare il Gateway con il servizi in cloud;
- Il cloud per fornire servizi all'utente quali persistenza dei dati e irrigazione automatica.

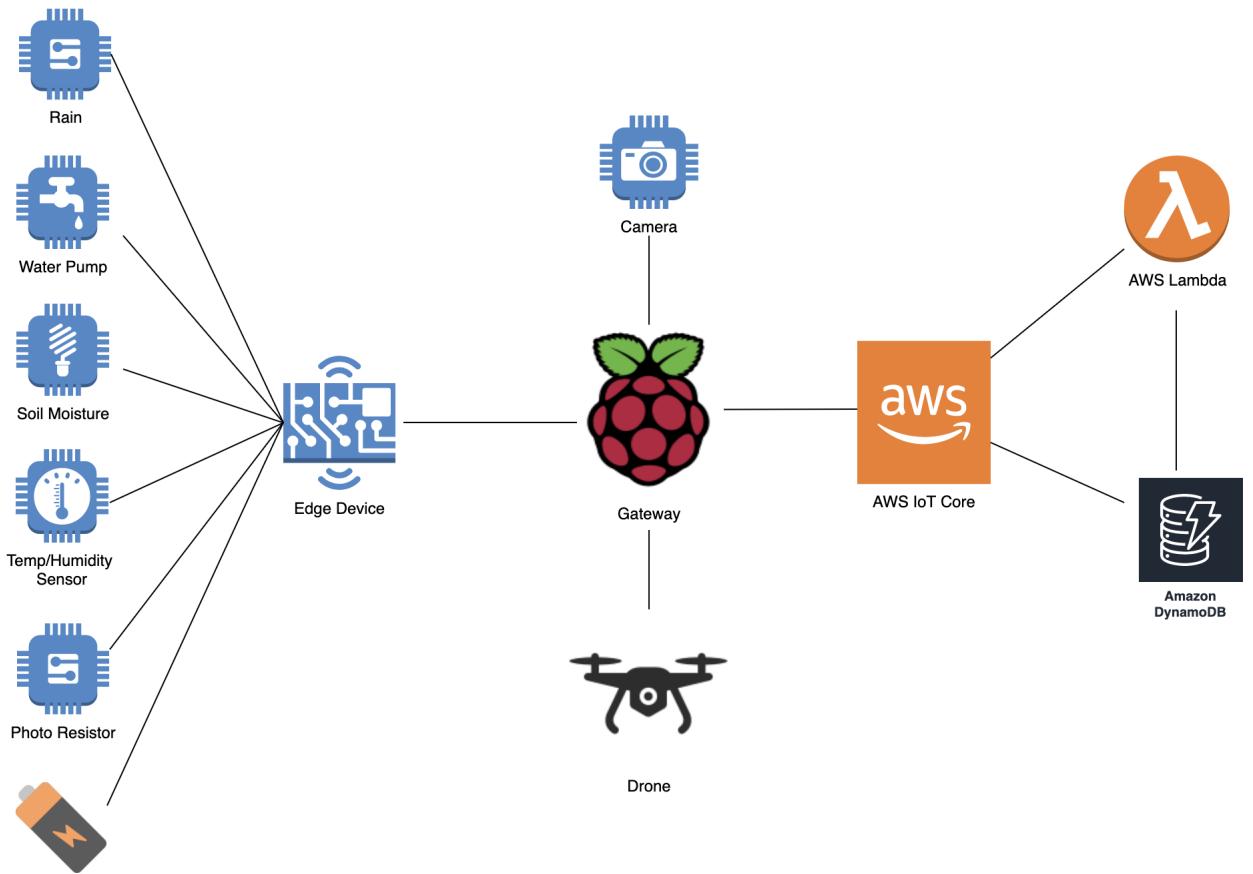


Figura 2: Diagramma delle componenti

5.1.1 Diagrammi dei Casi d’Uso

In questa sezione sono stati prodotti i diagrammi dei casi d’uso al fine di evidenziare e chiarire le necessità del settore della Smart Agriculture, che hanno motivato la realizzazione del progetto.

In Figura 3 è possibile individuare due casi d’uso principali, il *Gathering environment data* e *Start Irrigation*. Per il primo use case, i valori ambientali vengono rilevati periodicamente ed inviati al Gateway. L’Edge ha la responsabilità di ricevere i comandi di irrigazione dal Gateway ed avviare la pompa per un certo intervallo temporale. La pompa d’acqua verrà spenta automaticamente qualora, dopo un certo intervallo temporale, l’Edge non riceve un nuovo comando, sia esso di avvio o di spegnimento.

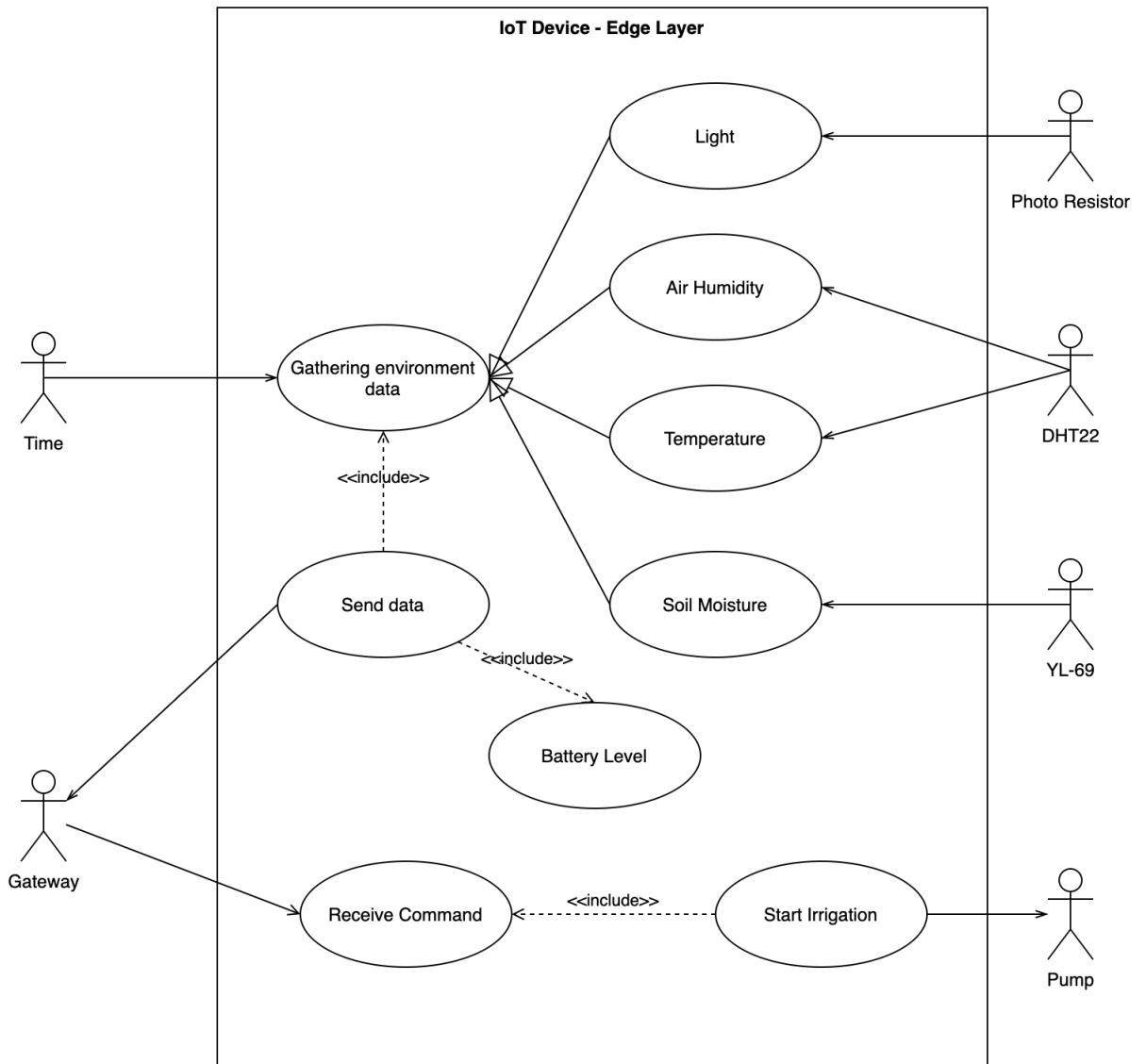


Figura 3: Diagramma dei casi d'uso per l'Edge Layer

Nel Fog Layer sono stati individuati quattro use case principali, *Aggregate data*, *Forward Irrigation Command*, *Soil Feeding* e *Human Detection*. Il primo use case viene avviato periodicamente ed ha l'onere di aggregare le misurazioni dei parametri ambientali precedentemente ricevute e memorizzate nel database. L'aggregazione include anche il loro invio in cloud.

Dall'architettura del sistema si evince che il *Gateway* rappresenta anche un “proxy”

che inoltra i messaggi scambiati tra il cloud e l'edge, per questo motivo è stato individuato lo use case *Forward Irrigation Command*.

Il *Soil Feeding* consiste nel pilotaggio del drone al fine di fertilizzare le colture. Tale use case è avviato dall'utente ed eseguito solo nel caso in cui le condizioni del drone sono favorevoli.

Inoltre, il Fog Layer deve rilevare la presenza umana all'interno del perimetro agricolo ed inviare una notifica all'utente qualora vi sia intrusione. In aggiunta, all'utente è data la possibilità di scattare manualmente una foto ed ottenere un breve video del campo.

In Figura 4 sono state modellate i servizi messi a disposizione dal cloud. In questo contesto, lo use case principale è *Irrigation Decision* che periodicamente determina se irrigare o meno in base alle misurazioni ambientali memorizzate nel database durante la giornata e alle previsioni meteo giornaliere. Qualora si decida di irrigare, si deve inviare il comando di avvio al *Gateway*.

Un'altra funzionalità del cloud, come visto in precedenza, è l'integrazione dei vari sistemi, tramite un MoM che inoltrerà i messaggi tra utente e *Gateway*.

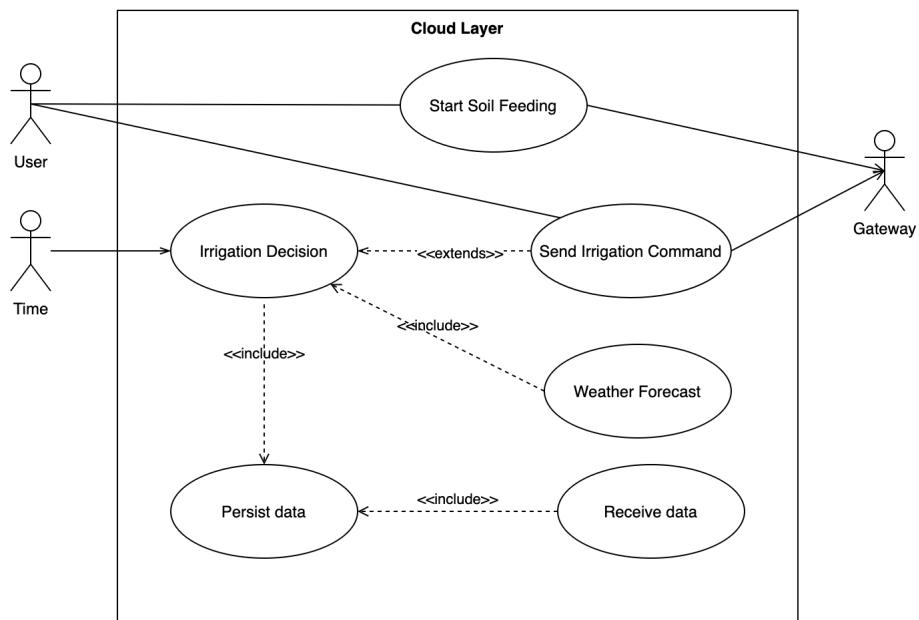


Figura 4: Diagramma dei casi d'uso per il cloud

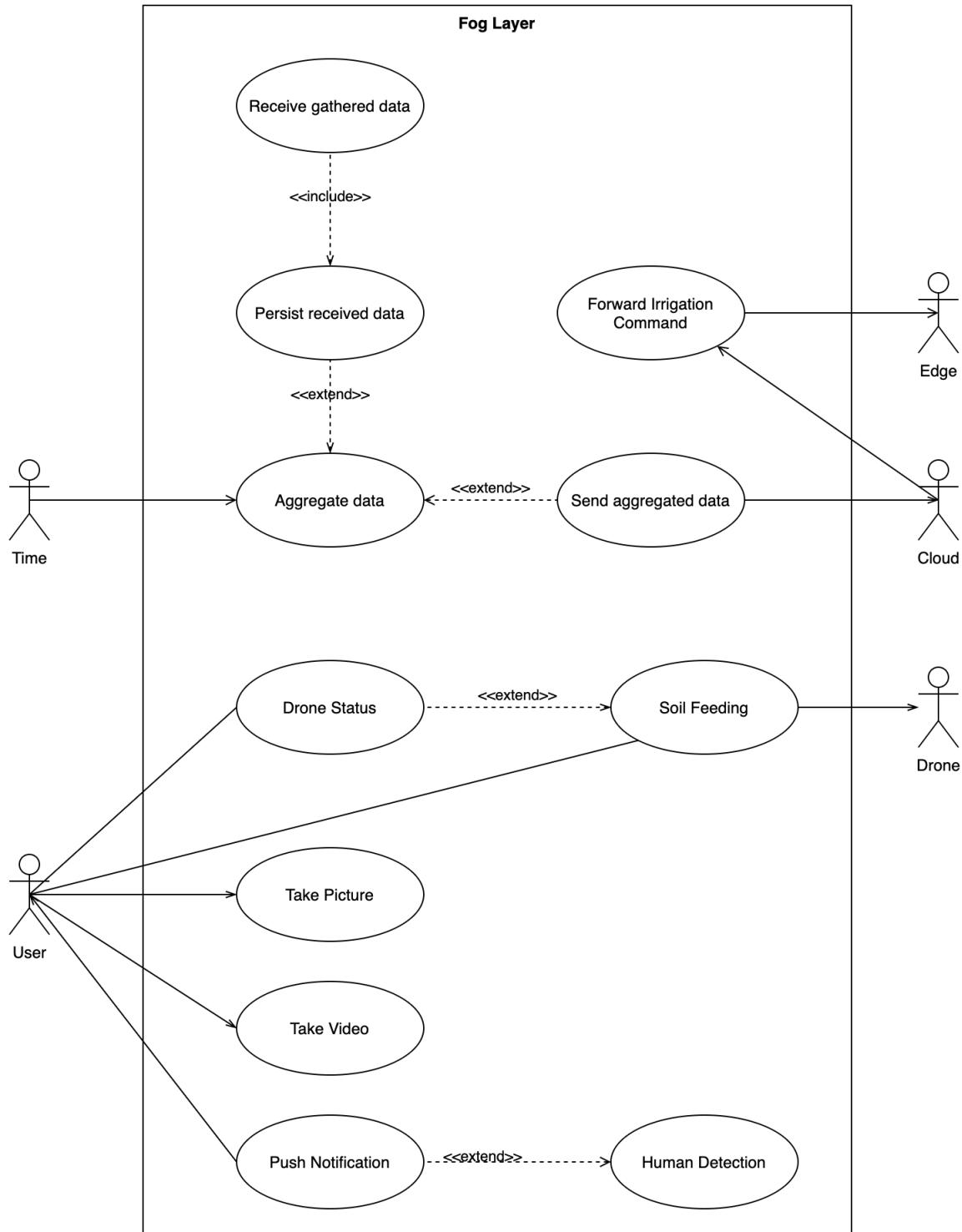


Figura 5: Diagramma dei casi d'uso per il Fog Layer
20

5.1.2 Interazione dei Componenti

Per quanto concerne l'interazione dei vari componenti, si è deciso innanzi tutto di porre l'accento sul flow dedicato all'acquisizione dei dati, trattandosi di una delle core feature del sistema sviluppato.

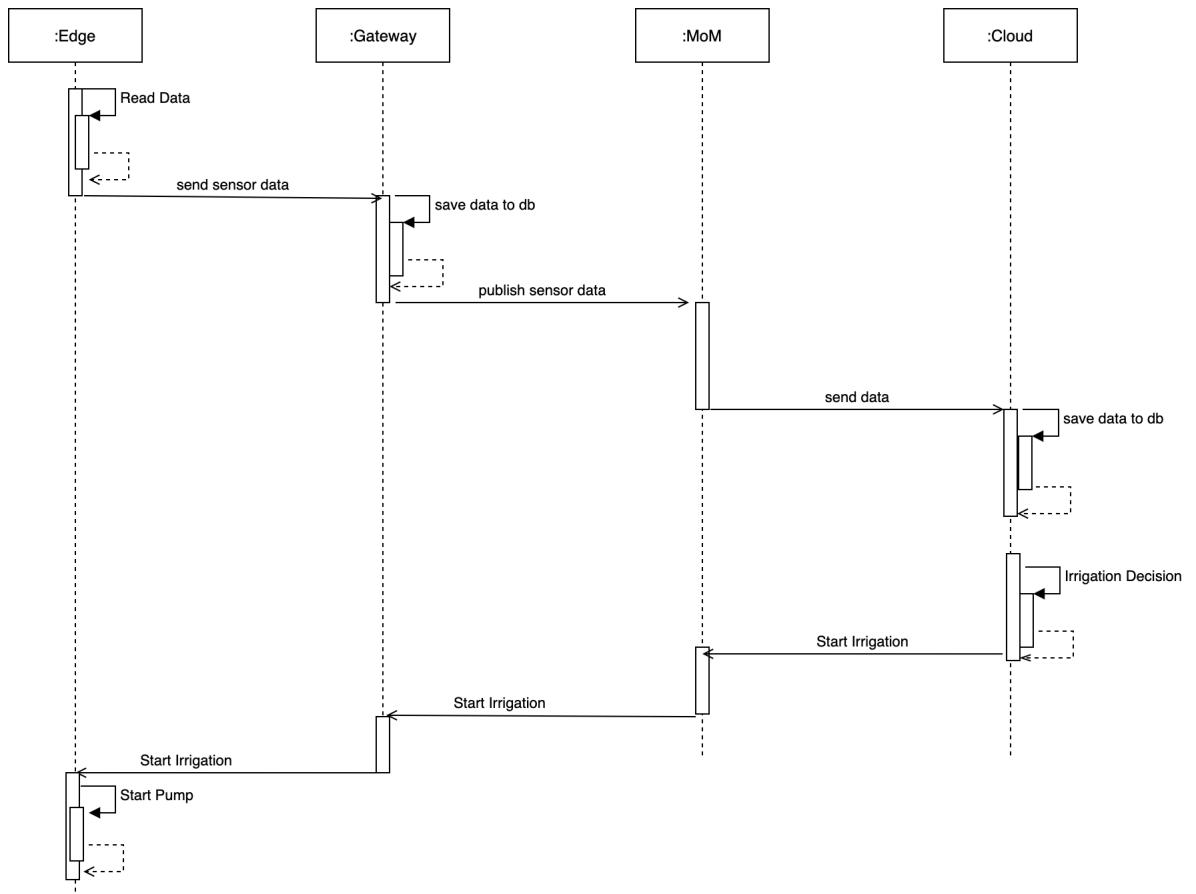


Figura 6: Diagramma di sequenza per il rilevamento dei dati sensoriali

Tale funzionalità, come anche le altre, fa uso di tutti i layer a disposizione: a partire dall'Edge i parametri ambientali vengono inviati al Gateway, situato nel Fog Layer, il quale, a sua volta, li rende temporaneamente persistenti e periodicamente li invia al cloud tramite l'uso di un MoM. Il cloud mantiene lo storico delle misurazioni e regolarmente invierà un comando di avvio irrigazione qualora saranno soddisfatti i vincoli stabiliti nei Requirements.

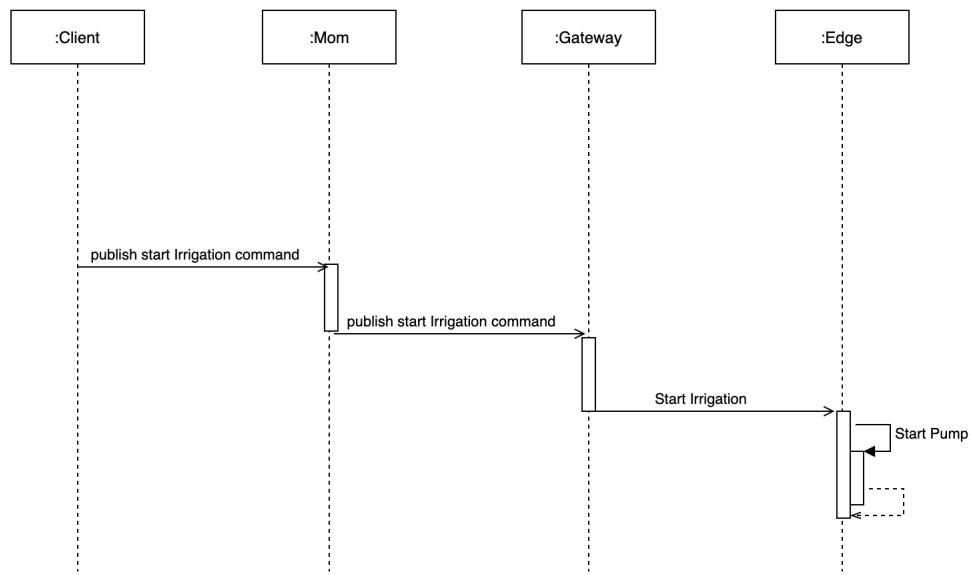


Figura 7: Diagramma di sequenza per l'irrigazione

Nel diagramma in Figura 7 è stato evidenziato il flusso per quanto riguarda l'avvio manuale dell'irrigazione a partire da un client gestito dall'utente finale.

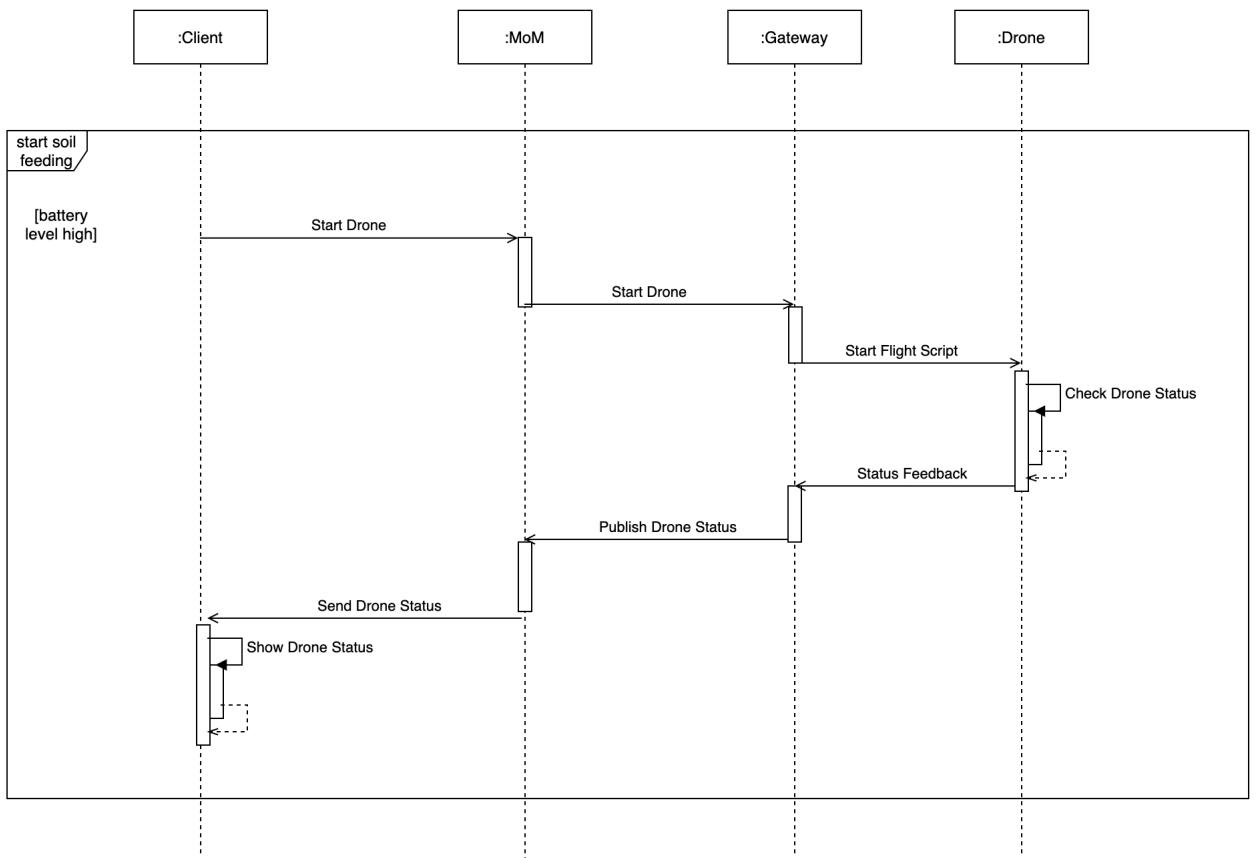


Figura 8: Diagramma di sequenza per la fertilizzazione

In figura 8 è rappresentato l'avvio manuale del drone. Anche in questo caso l'interazione coinvolge tutti i componenti del sistema, dal Client che invia il comando di fertilizzazione al gateway tramite MoM. Sarà il Gateway a pilotare il Drone ed inviare un feedback all'utente sullo stato del veivolo.

5.2 Clients

Per motivi di tempo e per rimanere ancorati il più possibile al contesto dell'esame si è scelto già in fase di design di non implementare un'interfaccia grafica per l'interazione utente, ma bensì di utilizzare l'app di messaggistica Telegram e l'interazione vocale di Amazon Alexa.

- **Telegram:**

La decisione di usare Telegram rispetto ad altre app di messaggistica è legata al fatto che viene offerta la possibilità di creare un Bot per interagire direttamente con le API del servizio di messaggistica e per compiere operazioni a fronte dell'invio di un comando speciale, che sintatticamente deve partire con il carattere “/”. In altre parole, dalla documentazione ufficiale si evince che i Bot sono dei semplici account di Telegram ma dove a rispondere non è una persona fisica ma un software che può svolgere qualsivoglia operazione, dal mostrare immagini e video o testo all'interagire con altri servizi o passare comandi all'Internet of Things. [11] Nel nostro caso, col fine di rendere il più possibile modulare il sistema, verranno creati due Bot:

1. *SmartCityBot_Manager*: usando questo Bot l'utente può innanzitutto inviare il comando di irrigazione e ricevere un feedback sull'avvenuta partenza della pompa d'acqua.

Inoltre, tramite questo Bot è possibile interagire direttamente con il Drone, quindi chiedere di farlo decollare, ricevere in tempo reale lo stato dell'avvenuto decollo o messaggi di warning in caso di errori riscontrati nel quadricottero, nonché un messaggio finale di fertilizzazione avvenuta con successo. I comandi da inviare si riassumono quindi in questo modo:

`/start_feeding` - Attiva il Drone e inizia il processo di fertilizzazione
`/start_irrigation` - Attiva la pompa e inizia il processo di irrigazione

2. *SmartCityBot_Surveillance*: Bot dedicato alla ricezione di immagini in caso di intrusione umana non autorizzata nel campo agricolo. I comandi da inviare si riassumono in questo modo:

`/get_photo` - Scatta una foto dalla telecamera e inviala all'utente
`/get_video` - Registra un video di qualche secondo e invialo all'utente

`/surveillance_start` - Fai partire modalità sorveglianza. Verranno inviate delle foto all'utente non appena la telecamera avverte un movimento di un essere umano

`/surveillance_stop` - Smetti di ricevere foto all'avvenuto rilevamento del movimento

`/surveillance_status` - Restituisce lo stato attuale per capire se la modalità sorveglianza è attiva o disattivata.

- **Alexa:** Anche Alexa è programmabile: Amazon mette a disposizione delle API per la creazione di una Skill che potenzialmente potrebbe essere inserita nello Skill Store di Amazon una volta terminata. Nell'ecosistema Alexa, per "Skill" si intende un'abilità in più che viene aggiunta all'assistente vocale. Ogni sviluppatore può creare una Skill scegliendo il linguaggio di proprio gradimento. Solitamente come backend per codificare la logica della Skill si utilizzano le Lambda, servizio offerto da Amazon Web Services.

Dato il tempo richiesto per lo studio e l'implementazione della Skill si è scelto di creare una sola abilità, che è quella di lanciare il comando di irrigazione. La Lambda dovrà dunque interagire con il Gateway per mezzo del Broker di AWS IoT Core, facendo quindi uso di MQTT. Il Gateway dovrà quindi contattare a sua volta l'Edge device per avviare la pompa. Quest'ultimo passaggio avviene anche nel caso si utilizzi Telegram per irrigare.

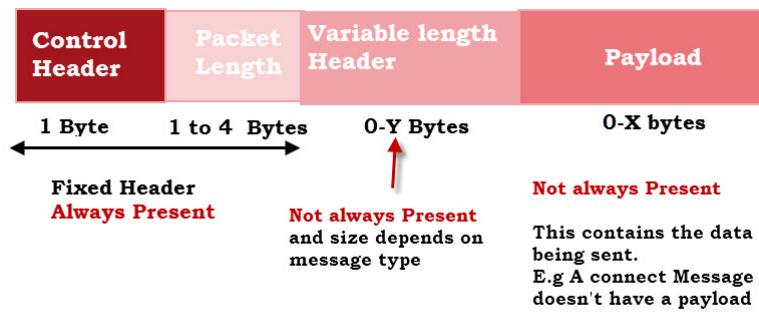
5.3 Comunicazione

La comunicazione è una delle caratteristiche fondamentali di un sistema distribuito, in quanto permette l'interazione tra i vari sotto-sistemi. In questo progetto vi è una forte comunicazione tra vari componenti distribuiti nell'ambiente, lo scambio di messaggi diventa quindi la parte più importante dell'intero sistema. Per questo motivo è stata dedicata una sezione apposita.

Prima di scendere in dettaglio, si vuole spendere qualche parola in generale sui concetti chiave di MQTT e BLE.

5.3.1 Introduzione a MQTT

Il Message Queuing Telemetry Transport è un protocollo di rete molto leggero, ideale per l'IoT in cui dispositivi distribuiti nell'ambiente devono comunicare. MQTT è basato sul pattern Publish-Subscribe, in cui vi è un Broker che inoltra i messaggi inviati dai Publisher ai Subscriber per un particolare “*Topic*”. Ad ogni “*Topic*” è associata una coda di messaggi, a cui i Subscriber devono sottoscriversi per poter essere notificati. MQTT, quindi, garantisce una comunicazione asincrona disaccoppiando le entità comunicanti. Ne consegue che, qualora una delle entità non dovesse essere attiva, l'altra entità non dovrà occuparsi di reinistaurare la connessione. Questo forte disaccoppiamento delle parti, che rimangono svincolate tra di loro e non hanno bisogno di mantenere uno stato, porta MQTT ad essere tra le prime scelte architettoniche da prendere in considerazione quando la mole di dati da scambiare è ingente e quando il sistema da sviluppare è squisitamente distribuito. A differenza di quanto avviene per HTTP, un pacchetto MQTT presenta una struttura molto snella e con un header contenente le informazioni essenziali per garantire la comunicazione.



MQTT Standard Packet Structure

Figura 9

In particolare, il formato è costituito da uno header di 2 byte fissi (sempre presenti), da un secondo header di lunghezza variabile (potrebbe anche non essere presente) e da un payload contenente i dati da scambiare. Anche quest'ultimo potrebbe essere vuoto, come nel caso del messaggio scambiato per instaurare la connessione, o per effettuare una disconnessione. L'unico collo di bottiglia potrebbe

be essere costituito dal Broker, che si deve occupare di smistare molti messaggi in tempo reale, ma nulla vieta di fare uso di una soluzione scalabile in Cloud dove il carico viene automaticamente gestito lato backend per mezzo di copie ridondanti e geograficamente dislocate. Per far fronte a questo problema, come già dichiarato nel paragrafo precedente, verrà scelta la soluzione AWS IoT Core, appositamente creato per la gestione e la comunicazione di molti device IoT e l'interfacciamento immediato con gli altri servizi AWS messi a disposizione. Per la nostra soluzione si è scelto di utilizzare MQTT, ma va puntualizzato che IoT Core offre una scelta più variegata di metodologie di comunicazione, ognuna con le proprie caratteristiche uniche ed utili a seconda delle diverse esigenze. Tra le altre alternative troviamo MQTT over WSS (Websockets Secure), HTTPS (Hypertext Transfer Protocol - Secure) e LoRaWAN (Long Range Wide Area Network).

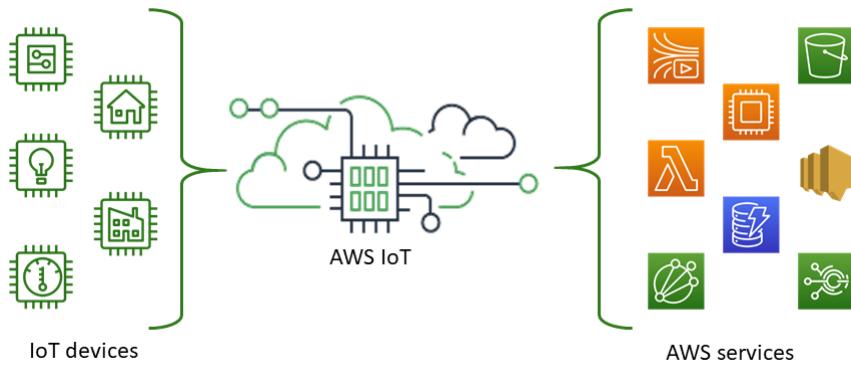


Figura 10: AWS IoT Core infrastructure

Tutti gli SDK messi a disposizione da Amazon per sviluppare un client che possa comunicare con l'infrastruttura AWS, hanno anche la possibilità di far scegliere allo sviluppatore il livello di qualità di servizio (QoS) per lo scambio dei messaggi. Il protocollo MQTT mette a disposizione 3 livelli di Quality of Service, ma IoT Core al momento si avvale soltanto dei primi due:

- *QoS Level 0*: Spedire 0 o più volte. Generalmente utilizzato per messaggi spediti su un tunnel di comunicazione piuttosto affidabile oppure per scenari in cui non è poi così importante che tutti i messaggi inviati vengano recapitati. Nel caso dei contesti IoT ad esempio, dove si parla di Big Data per

via della mole di dati emessi da tutti i sensori connessi, “ci si può permettere” una perdita di qualche messaggio, in quanto non andrà ad inficiare sulla conoscenza dello stato attuale del sensore e dell’informazione che si vuole ottenere.

- *QoS Level 1*: Spedire almeno 1 volta e rispedire finché non si riceve un messaggio di risposta di avvenuta ricezione (PUBACK). Questo livello viene utilizzato in contesti più critici dove ogni messaggio pubblicato è importante che venga recapitato correttamente al destinatario.

5.3.2 Comunicazione MQTT

Verrà ora riportata una panoramica sulla semantica dei topic e sulle Characteristics GATT per ogni entità in gioco.

- **Drone Topics:** Il sottosistema dedicato al Drone potrà avvalersi della sottoscrizione al topic `drone/command` per far decollare il drone. Su questo topic Alexa e il Telegram Bot pubblicheranno un comando richiesta di decollo. L’operazione di decollo potrebbe andare a buon fine, o potrebbe verificarsi un problema tecnico (ad esempio batteria scarica, problema al motore, assenza di segnale WiFi ecc). L’esito positivo o negativo viene spedito come messaggio di callback su Telegram utilizzando il topic `drone/fly`. Infine, se il percorso prestabilito viene eseguito correttamente e la fertilizzazione del campo agricolo va a buon fine, il drone atterra e spedisce un comando di avvenuta fertilizzazione nel topic dedicato `drone/status`.

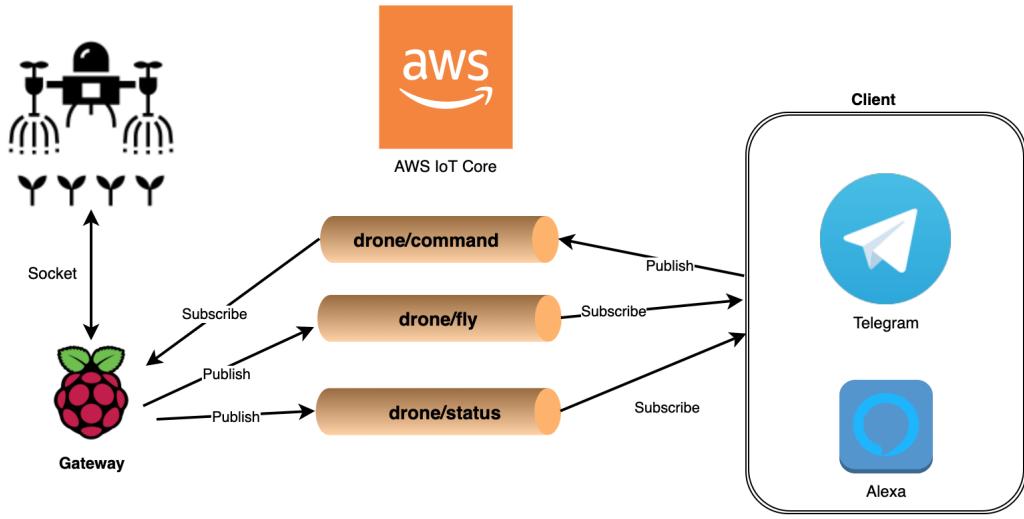


Figura 11: Drone publish-subscribe topics

- **Irrigation Topics:** Il *Gateway* avendo la responsabilità di inoltrare i messaggi di irrigazione tra IoT device e cloud, si sottoscrive al topic **irrigation/command** per ricevere i comandi per avviare o terminare l’irrigazione dai publisher, quali Alexa e Telegram Bot.

Pubblicherà invece lo stato dell’irrigazione, ricevuto tramite notifica BLE dall’IoT device, sul topic **irrigation/status** in modo che i subscriber (Alexa e Telegram Bot) possano riceverlo ed elaborarlo.

I messaggi che le entità si scambiano sono:

- sul topic **irrigation/command**:
 - * "on": per avviare l’irrigazione;
 - * "off": per terminare l’irrigazione.
- sul topic **irrigation/status**:
 - * "on": per indicare che l’irrigazione è stata avviata;
 - * "off": per indicare che l’irrigazione è terminata;
 - * "already-started": per indicare che l’irrigazione è già stata avviata ed è ancora attiva.

In questo modo, viene data all'utente la possibilità di ricevere gli aggiornamenti sull'effettivo stato dell'irrigazione, così da avere un feedback in tempo reale dell'effettivo funzionamento, e quindi intervenire prontamente sul sistema di irrigazione qualora occorra eseguire manutenzione.

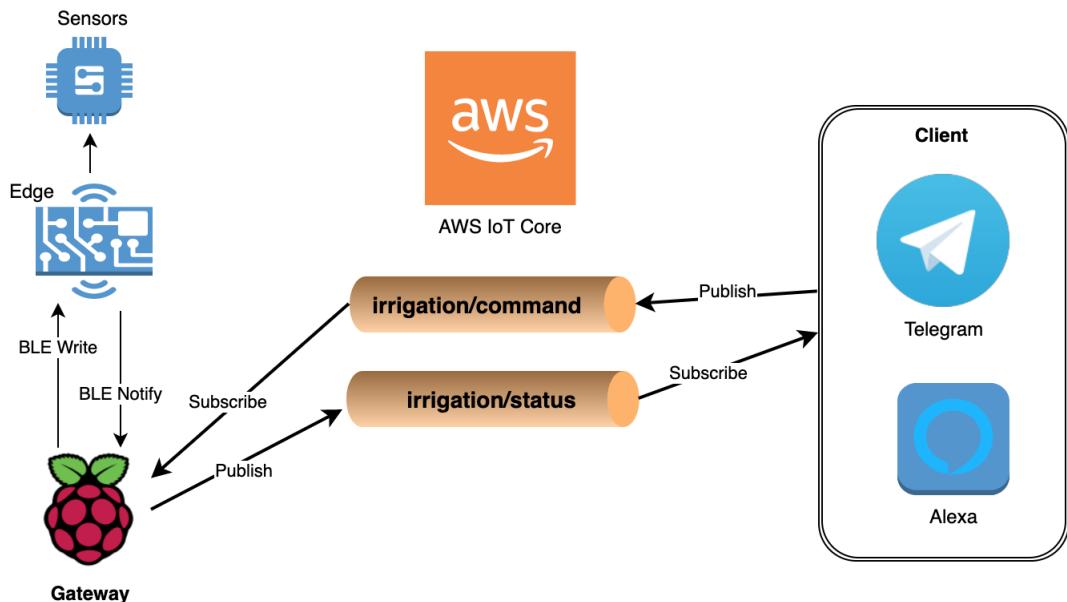


Figura 12: Irrigation publish-subscribe topics

- **Telemetry Topics:** Il *Gateway* oltre a comunicare con i client, Alexa e Telegram Bot, comunica anche con il cloud per memorizzare i dati telemetrici, ovvero le misurazioni relative alla temperatura, umidità relativa, umidità del terreno, intensità luminosa, la quantità di precipitazioni, e il livello della batteria utili per prendere la decisione di irrigare. In questo scenario, il microcontrollore immerso nell'ambiente invia ad intervalli regolari i dati telemetrici al *Gateway*, che a sua volta li pubblicherà sul topic `irrigation/telemetry`.

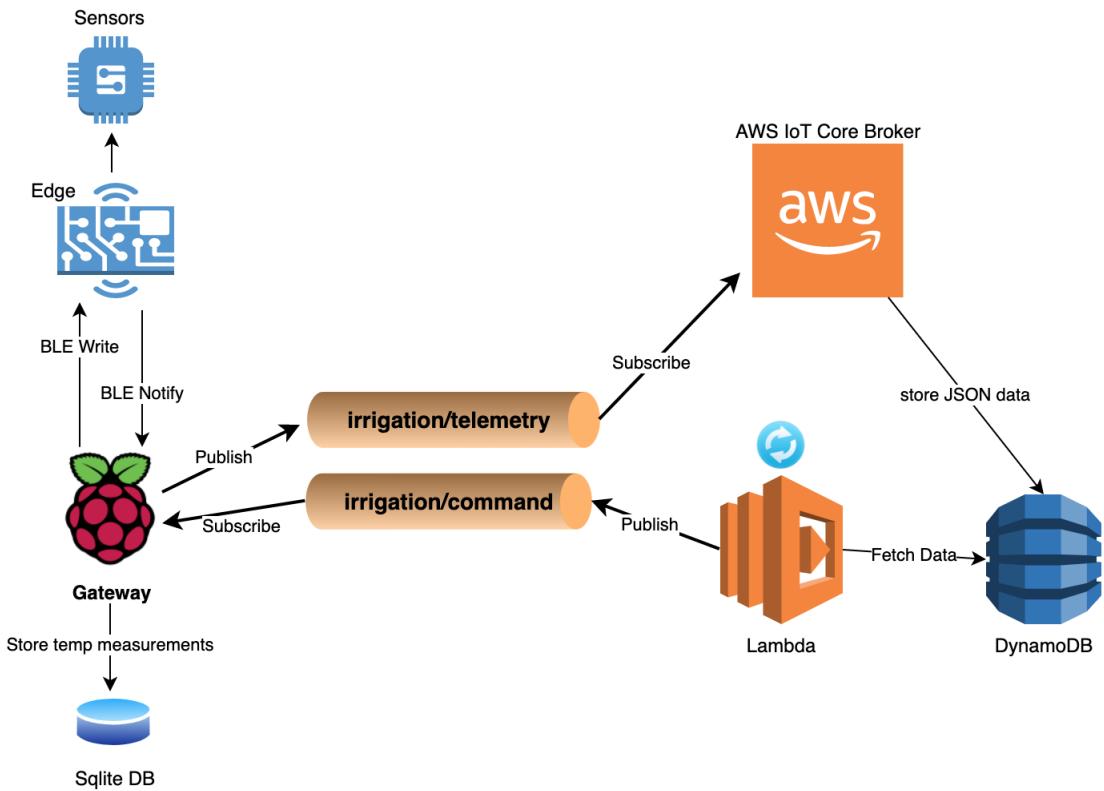


Figura 13: Irrigation publish-subscribe topics

In questo caso vi è un solo subscriber al topic, ovvero la AWS IoT Rule che alla ricezione dei dati li memorizza in modo permanente sul database. In un’ottica IoT, dove non è tanto importante la singola row nel db ma bensì un bulk di dati da processare ad ogni iterazione, si è optato per una soluzione NoSQL, e quindi DynamoDB, un document-oriented store database completamente proprietario e gestito da Amazon, ma per questo motivo facilmente integrabile con gli altri servizi AWS utilizzati.

5.3.3 Introduzione a BLE

Bluetooth Low Energy è una tecnologia progettata per ridurre il consumo energetico dei device, utile per contesti IoT. BLE utilizza il framework GATT

(Generic Attribute Profile) che stabilisce come le informazioni vengano scambiate in una connessione tra device BLE utilizzando concetti quali “*Services*” e “*Characteristics*”. Ogni service è identificato da un UUID univoco ed è un contenitore di characteristics, identificate anch’esse da un UUID. [8] A loro volta, le Characteristics sono contenitori di dati utente e di informazioni aggiuntive come le *properties*, che definiscono le azioni consentite sui dati. Un universally unique identifier (UUID) è un numero di 128-bit (16-bytes) avente la garanzia di essere unico a livello globale, ed oltre che nello standard Bluetooth è utilizzato anche in altri protocolli e contesti applicativi. GATT mette a disposizione degli UUID di base creati esclusivamente per gli usi standard e più comuni nei contesti applicativi. Alcuni esempi di semantiche associate ai codici standard possono essere il Device Name, il Battery Level, il livello di pressione, il TimeZone, il DateTime, il livello di glucosio nel sangue, la temperatura, l’umidità, ecc, ecc. Col fine di rimanere il più possibile fedeli agli standard proposti GATT, si vuole lasciare la generazione casuale degli UUIDs per le Characteristics con semantica specifica nel contesto di progetto (ad esempio Irrigation Characteristic), utilizzando invece gli UUIDs di base per gli usi più comuni. E’ buona norma non utilizzare mai i Base-UUID per scopi custom, ma anzi utilizzare tali codici pre-esistenti soltanto per identificare i servizi e le caratteristiche per cui sono stati creati.

5.3.4 Comunicazione BLE:

Di seguito vengono elencati i codici standard scelti per le varie Characteristics; tutti i rimanenti invece saranno generati casualmente da un generatore casuale di UUIDs:

- Battery Level: 00002a19-0000-1000-8000-00805f9b34fb
- Temperature: 00002a6e-0000-1000-8000-00805f9b34fb
- Humidity: 00002a6f-0000-1000-8000-00805f9b34fb

Per quanto concerne le properties, tra le diverse esistenti, nel nostro progetto sono state prese in considerazione le seguenti:

- Read: utile al client per chiedere i dati al server in modo sincrono;

- Notify: indica che il server invierà i dati al client non appena disponibili, secondo il principio Hollywood;
- Write: utile al client per cambiare il valore dei dati memorizzati dal server.

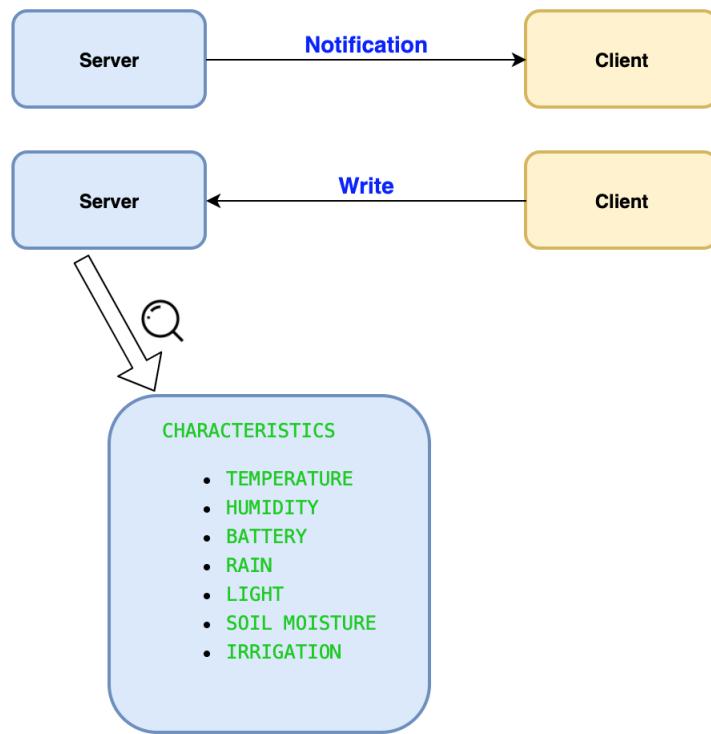


Figura 14: BLE communication between Edge device and Gateway

Come si evince già dagli schemi precedenti, il Gateway, che nel contesto BLE funge da client, dovrà avvalersi principalmente di due delle funzionalità proposte dall'architettura, ovvero lo scambio di messaggi usando il meccanismo di Write e di Notify.

Il server (IoT device) dunque si occuperà di leggere i valori dei sensori e serializzarli in formato binario per inviarli al client. Il meccanismo usato per la comunicazione verso il client sarà dunque quello della Notification (Notify), per evitare che il client richieda costantemente i dati aggiornati in modalità polling. In tal modo, si sfrutta il principio Hollywood ed il client

sarà notificato solo quando i nuovi dati saranno disponibili. L'applicazione di tale pattern di comunicazione, vede il client sottoscriversi alle Notify di tutti Service e tutte le Characteristics di suo interesse, che corrispondono esattamente a quelle dedicate ad ogni sensore.

Diversamente, la proprietà Write dovrà essere utilizzata dal client per trasmettere il nuovo valore che i dati del server dovranno avere. L'unico componente hardware che avrà bisogno di un aggiornamento manuale del proprio stato sarà il relè per azionare la pompa, quindi l'unico sensore a fare uso del comando impacchettato nel messaggio di Write da parte del client. Tutti gli altri sensori invece spediranno delle notify per aggiornare il server (Gateway).

Per quanto concerne il meccanismo di Read, seppur inizialmente preso in considerazione, sarà se possibile evitato per fare spazio alla Notify, rendendo l'intera comunicazione asincrona da e verso il Gateway. Come rafforzativo, va detto che la logica del Gateway non possiede nessun evento scatenante che richieda di sapere immediatamente (cioè in tempo reale) lo stato di un certo sensore presente nell'Edge, ma può anzi attendere che il dato venga spedito dall'Edge e recepito ad intervalli regolari. Tale scelta architetturale è stata spinta anche da un desiderio di lasciare i due layer il più possibile separati e indipendenti, rimanendo operativi in caso di cambio di hardware per sviluppi futuri.

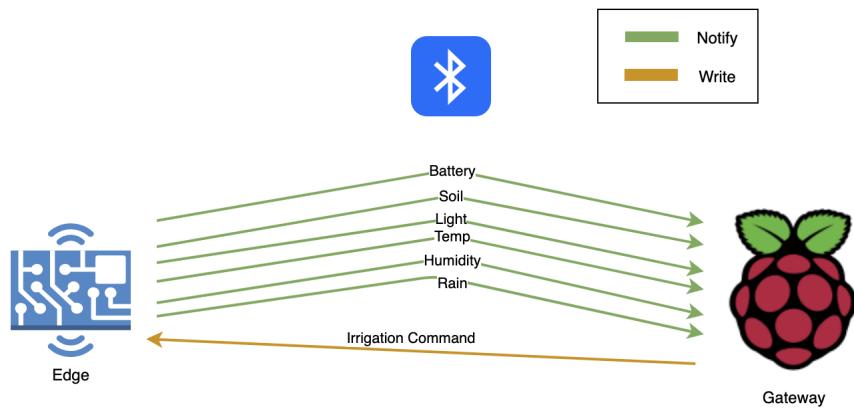


Figura 15: BLE Communication between Edge and Gateway devices

5.4 Micro-irrigation

La progettazione dell'IoT device consiste nel formalizzare le tre componenti tipiche di un sistema software: comportamento, struttura, interazione.

Nella Figura 17 è rappresentato il diagramma delle classi per i concetti fondamentali della modellazione dell'IoT device.

Nelle seguenti sezioni si analizzeranno le scelte progettuali ed i design pattern utilizzati, tipici dei sistemi embedded.

5.4.1 Comportamento

Il comportamento dell'IoT device è stato modellato tramite la macchina di Mealy, permettendo al sistema di reagire, quindi di eseguire un insieme di azioni, in base all'input ricevuto e allo stato attuale. I comportamenti individuati dall'analisi dei requisiti sono la rilevazione dei parametri ambientali e la gestione dell'irrigazione, che verranno trattati nei paragrafi seguenti.

Rilevazione dei parametri ambientali. La rilevazione dei parametri ambientali consiste nella lettura dei valori ambientali tramite opportuni sensori e nella loro trasmissione al Gateway. In Figura 16 è rappresentata la relativa macchina di Mealy, avente un solo stato con una sola auto-transizione temporizzata, ovvero che scatta periodicamente. La necessità di una transizione temporizzata è dovuta alla natura analogica dei sensori, che richiede una continua interrogazione ai sensori per ottenere i nuovi valori.

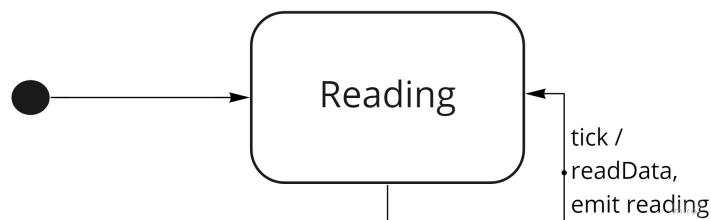


Figura 16: Macchina di Mealy per la rilevazione dei parametri ambientali

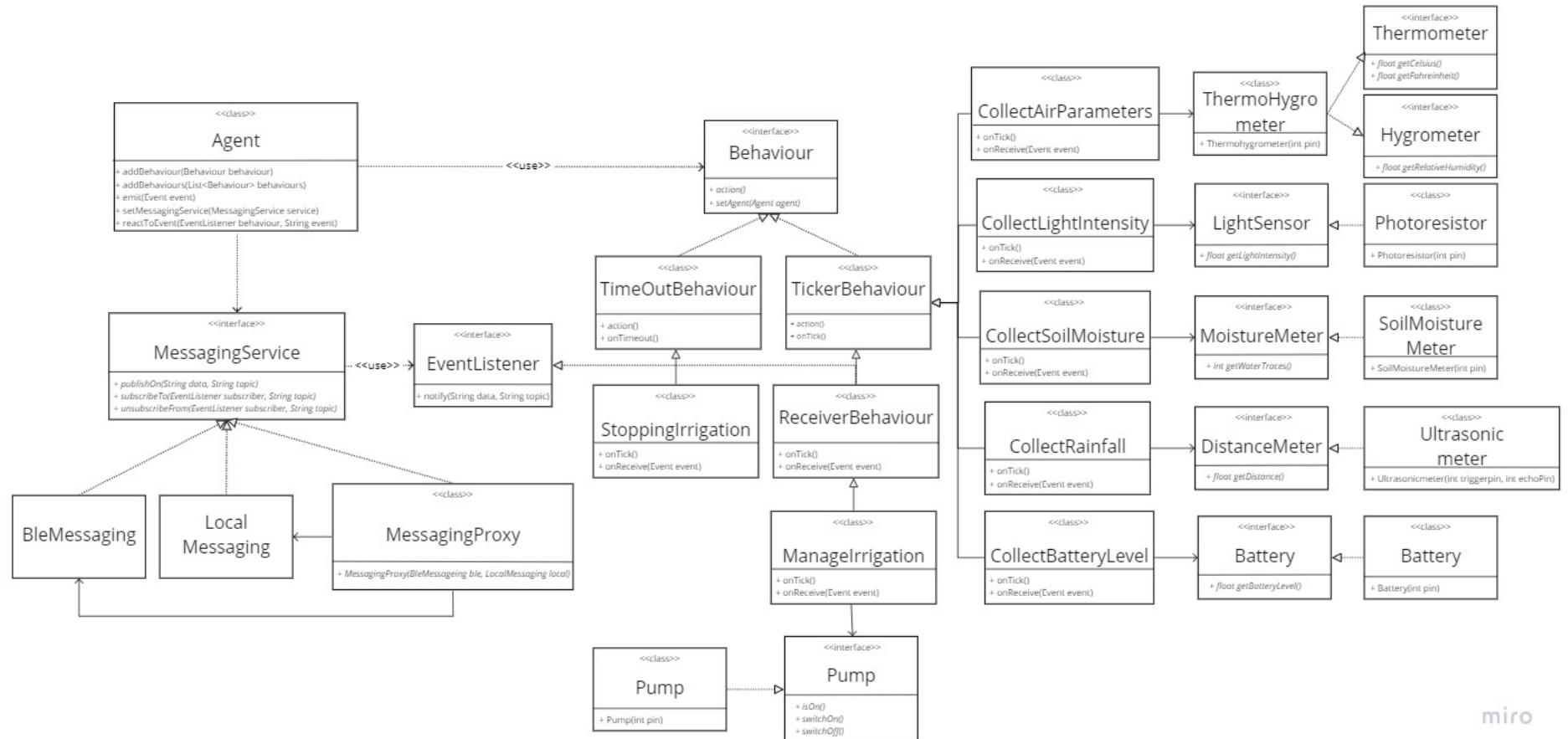


Figura 17: Diagramma delle classi per la modellazione del device IoT

Gestione dell’irrigazione. Questo comportamento è più complesso, in quanto il sistema deve reagire alla ricezione dei comandi inviati dal Gateway per accendere/spegnere la pompa di irrigazione. Lo scenario comune è quello in cui il sistema riceve il comando di accensione della pompa e successivamente il comando di spegnimento. Però, per evitare lo spreco di acqua e la sovra-irrigazione, è necessario garantire lo spegnimento della pompa anche quando la comunicazione con il *Gateway* potrebbe, per qualche motivo, interrompersi. A tale scopo, avviata la pompa deve essere avviato anche un timer in modo tale che al suo scadere (evento timeout) viene spenta la pompa se non è stato ricevuto il comando di spegnimento. Un altro scenario emerso dall’analisi dei requisiti, è il caso in cui il sistema riceva il comando di avvio quando la pompa è già attiva. In tal caso si è ritenuto opportuno inviare una notifica con contenuto “already-started” senza effettuare alcuna azione. La macchina di Mealy risultante è rappresentata in Figura 18; ha due stati e quattro transizioni. In questo caso gli eventi che fanno scattare le transizioni sono relativi alla ricezione di un comando e al timeout.

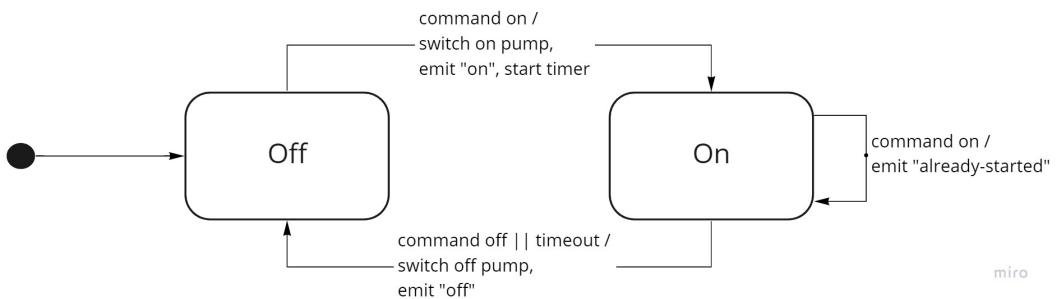


Figura 18: Macchina di Mealy per la gestione dell’irrigazione

5.4.2 Struttura

La modellazione strutturale del sistema consegue sia dall’analisi dei requisiti, ma anche dalle scelte effettuate nella modellazione del comportamento e dell’interazione. In particolare, il sistema interagisce con l’ambiente percepisce le modifiche tramite opportuni sensori ed attuatori che devono essere modellati in modo da disaccoppiare l’implementazione con il contratto di utilizzo, secondo il paradigma ad oggetti.

Tali sensori ed attuatori saranno utilizzati per realizzare il comportamento del sistema. Il sistema, come visto nella Sezione 5.4.1, ha diversi comportamenti che sfruttano diversi sensori ed attuatori.

Infine, è necessario modellare le entità in gioco per la comunicazione del sistema con l'esterno ma anche con l'interno, ovvero tra i vari comportamenti.

Di seguito si approfondirà la modellazione di questi concetti fondamentali, entrando nel merito delle scelte progettuali effettuate.

Modellazione dei Sensori ed Attuatori. Per la modellazione dei sensori sono stati adottati i pattern Hardware Proxy e Hardware Adapter[4] con l'obiettivo di nascondere tutti i dettagli di basso livello, inerenti l'accesso all'hardware, rispettando un determinato contratto. Questo permette di cambiare hardware e non influenzare il client, ma soprattutto di svincolare il client dalla gestione dell'hardware. Concretamente, per ogni sensore è stata definita un'interfaccia che rispecchia le proprie caratteristiche ed una classe che la implementa, come è possibile vedere in Figura 19.

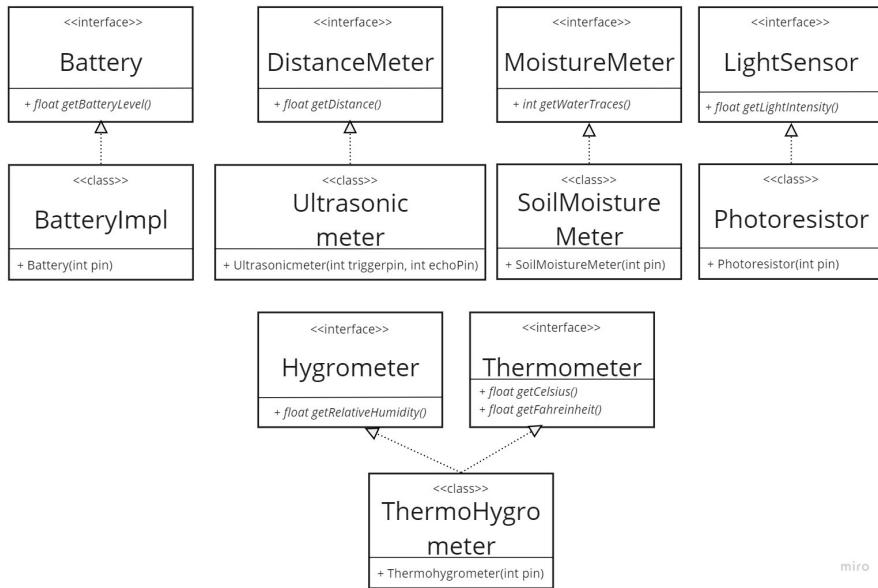


Figura 19: Diagramma delle classi dei sensori ed attuatori

Modellazione della comunicazione. La comunicazione, sia interna che esterna, è stata unificata mediante l'applicazione del pattern **Proxy** e dell'interfaccia

MessagingService. In tal modo il client non deve conoscere i dettagli del canale di comunicazione (né indirizzo, né porta o altro), ma solo indicare l'argomento di interesse simulando il pattern *publish & subscribe*. Sarà il Proxy a gestire il flusso dei messaggi verso l'opportuno canale di comunicazione, che nel nostro caso è *BLEMessaging* o *LocalMessaging*.

In Figura 20 è rappresentata la modellazione minimale ma estendibile delle entità utili per la comunicazione.

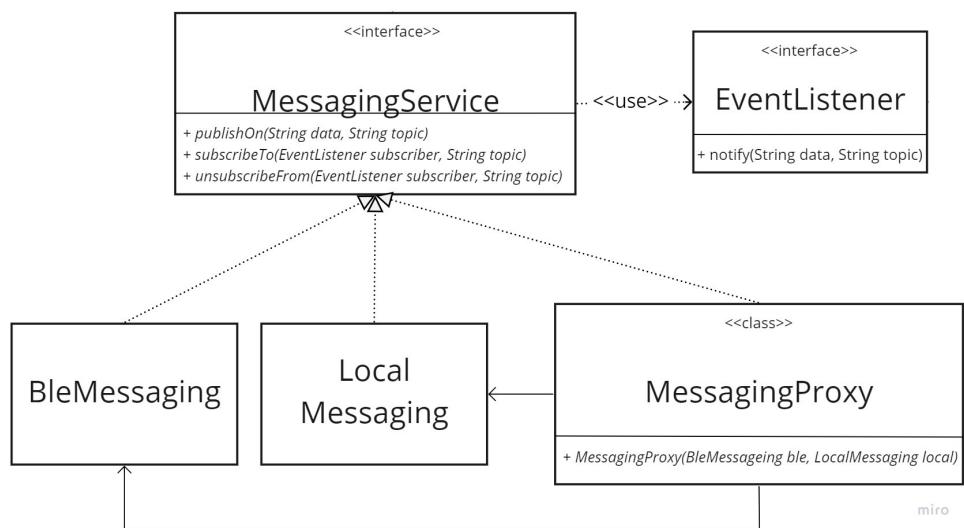


Figura 20: Diagramma delle classi del servizio di messaggistica

Modellazione del concetto di "Comportamento". Dopo avere modellato i sensori e gli attuatori, occorre modellare l'entità che coordina le azioni di un insieme eterogeneo di sensori/attuatori al fine di raggiungere l'obiettivo desiderato. Per fare ciò è stato utilizzato il design pattern Mediator [4]. Dal punto di vista semantico, invece, l'entità è stata nominata *Behaviour*. Ogni Behaviour è espressa con una macchina di Mealy implicita. Analizzando il problema, sono stati evidenziati diversi tipi di behaviour:

- periodico: eseguito ciclicamente con un certo periodo;
- ricevente: reattivo alla ricezione di un evento, sia esso un messaggio ricevuto o un evento interno come il timeout;

- timeout: eseguito solo una volta, al verificarsi dell'evento di timeout.

Infine, è stato modellata l'entità *Agent* che ha la responsabilità di eseguire i propri *Behaviour*, incapsulando la logica di basso livello legato al microcontrollore e alla piattaforma adottata.

La Figura 21 mostra il diagramma delle classi per le astrazioni *Agent* e *Behaviour*.

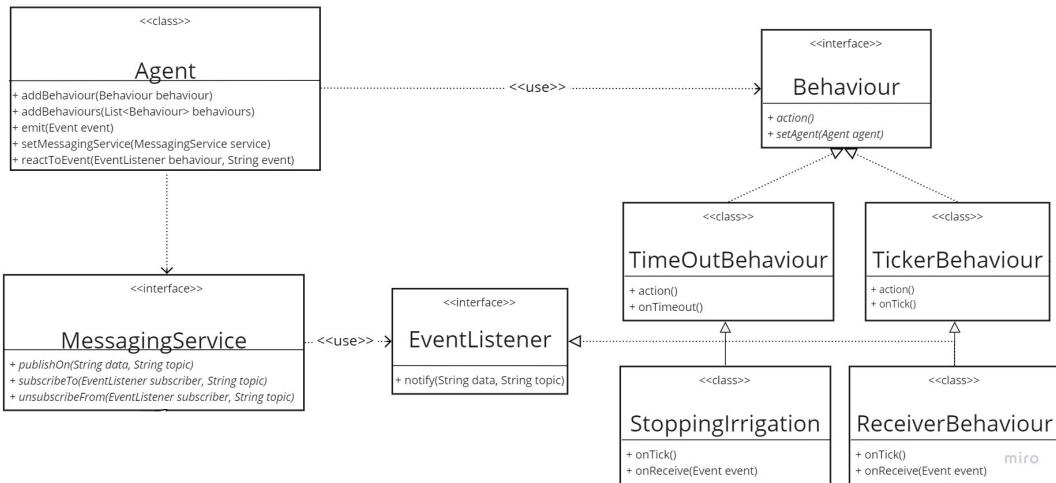


Figura 21: Diagramma delle classi: *Agent* e *Behaviour*

5.4.3 Interazione

Dall'analisi dei requisiti l'IoT device deve interagire con il *Gateway* al fine inviare i dati telemetrici al cloud e ricevere i comandi di irrigazione. Dalla modellazione della struttura, approfondita in Sezione 5.4.2, è necessario anche una forma di interazione interna tra gli *Agent*, per mezzo dei *Behaviours*.

Per entrambi i casi, l'interazione deve avvenire secondo il pattern **fire & forget**, noto anche come *One-way*, sia perché la gestione della perdita di un messaggio non è importante dato che il nuovo messaggio rimpiazzerà il vecchio, sia per non sovraccaricare il sistema embedded di una gestione più complessa della comunicazione.

Per ulteriori dettagli sulla comunicazione con il *Gateway* si rimanda alla Sezione 5.3.

5.5 Circuito elettronico

In questa sezione si vuole entrare nel merito della progettazione del circuito elettronico.

La scelta di effettuare una progettazione circuitale è dovuta alla mancanza di moduli pre-progettati, che ha reso necessari alcuni accorgimenti per poter sviluppare le feature.

Per quanto riguarda la rilevazione della temperatura, dell'umidità relativa, dell'umidità del terreno e della quantità di precipitazioni sono stati utilizzati moduli progettati appositamente per essere installati in modalità plug-and-play senza alcuna necessità di resistori, condensatori o altro materiale elettronico.

Diversamente, per la rilevazione dell'intensità luminosa e per il livello della batteria è stato necessario creare un proprio circuito.

Di conseguenza, nelle sezioni successive, verrà trattato il design circuitale per la rilevazione dell'intensità luminosa e per il livello della batteria. Occorre, però, effettuare una panoramica sul **partitore di tensione**, circuito essenziale per entrambe le feature.

Infine verrà presentato lo schema circuitale dell'intero sistema e il diagramma fritzing.

5.5.1 Partitore di Tensione

Il partitore di tensione, anche noto come **partitore resistivo**, è un circuito formato da due o più resistenze in serie, con l'obiettivo di ripartire la tensione applicata ottenendone una più piccola.

Si parla semplicemente di *partitore di tensione* quando si utilizzano resistori con valori fissi di resistenza, mentre di *partitore di tensione variabile* quando una delle resistenze è variabile.

La configurazione base di un partitore resistivo è formata da due resistori collegati in serie, attraversati dalla stessa corrente.

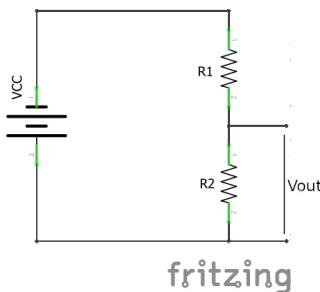


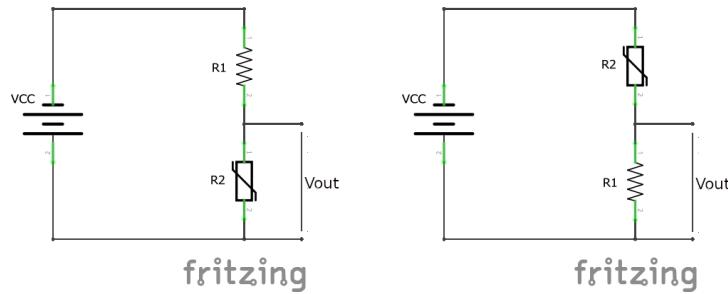
Figura 22: Circuito elettronico del partitore di tensione

Dalla legge di ohm, la tensione ai capi di ogni resistore è uguale al prodotto tra la corrente che attraversa quest'ultimo e la resistenza che essa presenta. Applicando le opportune sostituzioni si ha che la tensione ai capi della seconda resistenza è data dalla seguente formula.

$$V_{output} = V_s \times \frac{R2}{(R1 + R2)}$$

Nel caso del **partitore resistivo con resistenza variabile**, vi sono due modi equivalenti di progettare la rete di un partitore di tensione:

1. A valle dell'alimentazione viene collegata la resistenza con valore fisso ed in serie la resistenza variabile, come indicato in Figura 23(a). Il valore della tensione aumenta all'aumentare della resistenza.
2. A valle dell'alimentazione viene collegato la resistenza variabile ed in serie la resistenza con valore fisso, come indicato in Figura 23(b). Il valore della tensione aumenta al diminuire della resistenza.



(a) tensione max con valore max della resistenza (b) tensione max con valore min della resistenza

Figura 23: Partitore di tensione con resistenza variabile

5.5.2 Rilevazione dell'intensità luminosa

Per la rilevazione dell'intensità luminosa è stato utilizzato un fotoresistore LDR (Light Dependent Resistor). Un LDR è una cella foto-conduttiva realizzata con un materiale semi-conduttivo chiamato *solfato di cadmio*, o noto come *cadmium sulphide* (CdS). La resistenza di LDR cambia il proprio valore ohmico in funzione della quantità di luce che lo colpisce.

Al buio diventa un isolante, in quanto non ha abbastanza elettroni da condurre. In presenza di luce, invece, l'energia luminosa costituita da fotoni, lo colpisce rilasciando elettroni che permettono la conduzione. Più elettroni ha, più condurrà, diminuendo la propria resistenza.

Per misurare il valore resistivo di LDR occorre progettare un partitore di tensione con resistenza variabile. In questo caso si è scelto di realizzare il partitore in Figura 23(b), così all'aumentare della luminosità diminuisce la resistenza e aumenta la tensione.

L'accuratezza dell'intensità di luce calcolata dipende dal range dei valori di tensione agli estremi di LDR. Minore sarà il range, più sfumata sarà la differenza tra la condizione di buio e di luce. Per evitare ciò, è fondamentale la scelta del resistore da porre in serie a LDR. Dovrà avere un valore di resistenza calcolata in funzione della resistenza massima (R_{max}^*) e minima (R_{min}^*) generata da LDR.

Data la mancanza di un multmetro per compiere queste misurazioni sperimentali, sono stati utilizzati valori standard, quali $1M\Omega$ (R_{max}^*) e $5.4K\Omega$ (R_{min}^*).

Di conseguenza, avendo una tensione di alimentazione di $3.3V$, applicando la legge di Ohm vista nella Sezione 5.5.1 abbiamo che la resistenza deve essere di $10K\Omega$ per ottenere una $V_{out_{max}} = 2.14V$ e $V_{out_{min}} = 0.03V$.

Per il circuito prototipale con breadboard vedere la Figura 24.

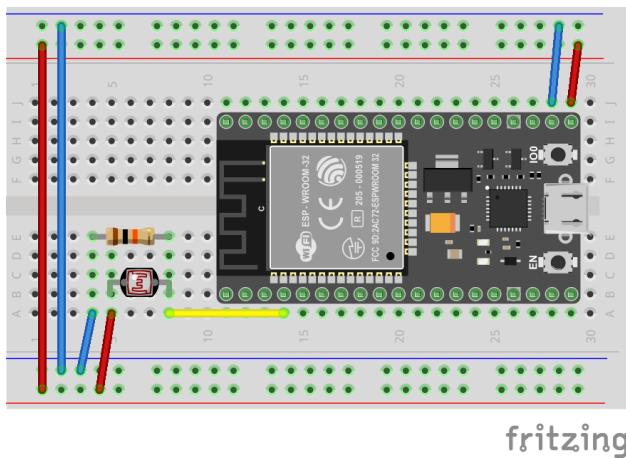


Figura 24: Diagramma fritzing per la rilevazione dell'intensità luminosa con LDR

5.5.3 Rilevazione del livello di batteria

Alimentando il microcontrollore con una batteria a 9V non è possibile leggere direttamente la tensione senza danneggiare la scheda. Per cui occorre utilizzare un partitore di tensione per leggere una tensione massima di 3.3V.

Secondo la legge di Ohm, vista nella Sezione 5.5.1, si ottiene: $R_1 = 2K\Omega$, $R_2 = 1.22K\Omega$ e $R_3 = 1K\Omega$.

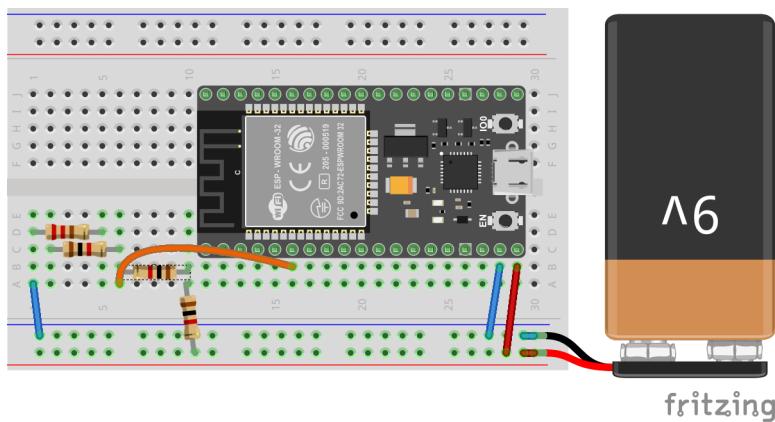


Figura 25: Diagramma fritzing per l'alimentazione del microcontrollore e la rilevazione dello stato della batteria

5.5.4 Diagramma del prototipo

Lo schema prototipale del circuito elettronico è rappresentato in Figura 26.

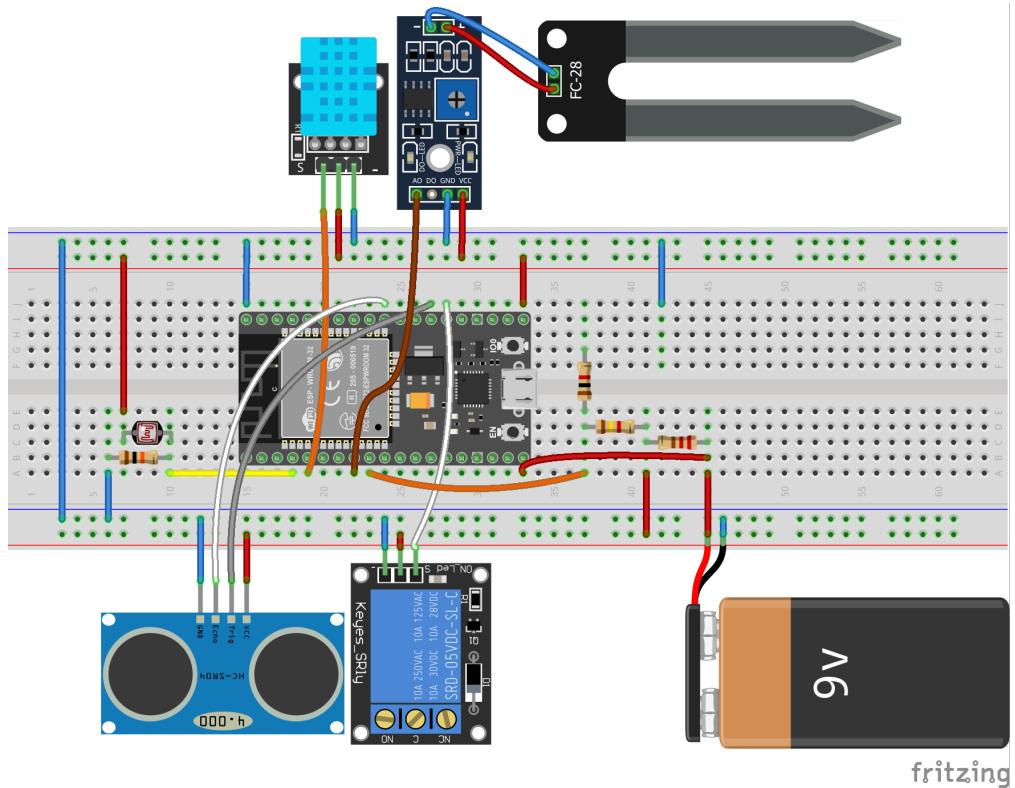


Figura 26: Diagramma complessivo del prototipo

6 Implementazione

In questo capitolo verranno descritte le scelte implementative, derivate da quelle progettuali. La descrizione di questa parte verrà suddivisa per quelle che sono le funzionalità core del sistema, evidenziando le parti salienti ed eventuali problemi riscontrati durante la stesura del codice.

6.1 Irrigazione

L’irrigazione, core feature del progetto, vede l’implementazione di più sistemi tra loro comunicanti al fine di prendere una decisione (se irrigare o meno) in base ai dati rilevati sul campo. I sistemi coinvolti sono tre, ovvero l’IoT device (ESP32), il Gateway (Raspberry Pi 4) posto nel Fog Layer e il cloud (AWS).

6.1.1 IoT Device

L’IoT device ha lo scopo di rilevare i parametri ambientali dai sensori ed essere reattivo ai comandi ricevuti tramite BLE al fine di avviare/spegnere la pompa istantaneamente.

L’implementazione delle componenti core del sistema derivano direttamente dalla modellazione prodotta nella Sezione 5.4, con ulteriori dettagli legati alla board, alla piattaforma, al linguaggio e alle librerie utilizzate.

Sensori ed Attuatori. Per la rilevazione della temperatura ed umidità relativa, dell’umidità del terreno, dell’intensità luminosa sono state utilizzate librerie implementate da terzi adattando l’interfaccia fornita con quella di dominio.

Per la rilevazione del livello della batteria, invece, è stata implementata la seguente formula per ottenere il voltaggio dai valori discreti letti dal pin analogico.

$$Voltage = AnalogueValue \times \frac{MicroOperatingVoltage}{ADCreadings} \times \frac{BatteryVoltage}{MicroVoltage}$$

Dal voltaggio è possibile ottenere anche la percentuale, come mostrato nel Listato 1.

Listing 1: Logica per rilevare il voltaggio e la percentuale di batteria

```
1 #define MAX_ADC_READING 1024.0
2 #define ESP32_VOLTAGE 3.3
3
4 class BatteryImpl : public Battery {
5 public:
6     explicit BatteryImpl(int pin, float maxVoltage):
7         pin(pin), maxVoltage(maxVoltage),
8         divisionFactor(maxVoltage/ESP32_VOLTAGE),
9         conversionFactor((ESP32_VOLTAGE / MAX_ADC_READING) * divisionFactor) {}
10
11    float getVoltage() {
12        int rawData = analogRead(pin);
13        float voltage = rawData * conversionFactor;
14        return voltage;
15    }
16
17    int getPercentage() {
18        float batteryVoltage = getVoltage();
19        return mapFloat(batteryVoltage, 0, maxVoltage, 0, 100.0);
20    }
21
22    ...
23 }
```

Infine, l'unico attuatore in uso è la pompa ad immersione alimentata tramite la rete elettrica e connessa al circuito tramite un relay. Per questo motivo la classe *Pump* contiene la logica per aprire e chiudere il circuito del relay affinché la corrente alternata possa circolare verso la pompa.

La pompa è collegata al pin NO (*Normally Open*) del relay, bloccando di default il passaggio della corrente. Per chiudere il circuito occorre applicare la logica inversa, ovvero fornire in uscita il valore HIGH come mostrato nel Listato 2.

Listing 2: Logica di funzionamento della pompa

```

1 class ImmersiblePump : public Pump {
2 public:
3     explicit ImmersiblePump(int pin) : pin(pin) {
4         pinMode(pin, OUTPUT);
5         this->switchOff();
6     }
7
8     void switchOn() {
9         digitalWrite(pin, LOW);
10        this->status = true;
11    }
12
13    void switchOff() {
14        digitalWrite(pin, HIGH);
15        this->status = false;
16    }
17
18    bool isOn() {
19        return this->status;
20    }
21
22    ...
23 }
```

Architettura ad Agenti. Nella Sezione 5.4 l’IoT device è stato modellato come un sistema multi-agente con interazione verso l’esterno.

Essendo un **sistema embedded**, con restrizioni sulle risorse disponibili, l’astrazione multi-agente è stata implementata tramite un’astrazione di più basso livello fornita dal sistema operativo **FreeRTOS**, ovvero il multi-tasking.

In questo scenario, ogni *Behaviour* è stato mappato come un task mentre l’*Agent* contiene la logica di creazione ed esecuzione dei task.

Essendo il microcontrollore dotato di una cpu **dual-core**, è possibile parallelizzare due task alla volta. Gli altri task saranno eseguiti con la tecnica dell’**interleaving**, per cui non devono essere bloccanti, altrimenti il core sarà impegnato ad eseguire solo e sempre lo stesso task.

Come soluzione, è stato implementato il **TickerBehaviour**, in Listato 3, che viene mappato come un task periodico. Dopo ogni tick il task passa nello stato di **waiting**, grazie alla chiamata alla funzione `vTaskDelay(period)`, liberando il core per l’esecuzione di un altro task.

Listing 3: Logica del *Behaviour* periodico

```

1 void TickerBehaviour :: action () {
2     for (;;) {
3         onTick();
4         vTaskDelay( period / portTICK_PERIOD_MS );
5     }
6 }
```

Diversamente il **TimeOutBehaviour**, in Listato 4, è mappato come un task che viene eseguito solo una volta ma dopo un certo intervallo di tempo, definito sempre dalla chiamata alla funzione `vTaskDelay(period)` in modo che durante questo intervallo, altri task possano essere eseguiti.

Listing 4: Logica del *Behaviour* eseguito al timeout

```

1 void TimeOutBehaviour :: action () {
2     vTaskDelay( timeout / portTICK_PERIOD_MS );
3     onTimeout();
4 }
```

Nel dominio del progetto, i *Behaviour* implementati sono due: *Reading* e *Watering*. I *Behaviour* che si occupano di leggere i parametri ambientali, ereditano dalla classe astratta *TickerBehaviour*. Periodicamente interrogano i sensori e trasmettono i valori al *Gateway*, come mostrato in forma di pseudocodice nel Listato 5.

Listing 5: Lettura dei parametri ambientali

```

1 void onTick() override {
2     float environmental_value = sensor->read();
3     SimpleBehaviour :: emit(environment_parameter, environmental_value);
4 }
```

Il **Watering**, in Listato 6, eredità dalla classe astratta *ReceiverBehaviour* ed implementa, in modo implicito, la macchina di Mealy descritta nella Sezione 5.4.1.

Listing 6: Logica di irrigazione

```

1 void handle(Event& event) override {
2     if (!this->pump->isOn() && isCommandOn(event)) {
3         this->pump->switchOn();
4         this->getAgent()->addBehaviour(timeout);
5         SimpleBehaviour :: emit(IRRIGATION_STATUS, ON);
6     } else if (this->pump->isOn()) {
7         if (isCommandOff(event) || isTimeout(event)) {
8             this->pump->switchOff();
9             SimpleBehaviour :: emit(IRRIGATION_STATUS, OFF);
10        } else if (isCommandOn(event))
11            SimpleBehaviour :: emit(IRRIGATION_STATUS, ALREADY_STARTED);
```

Il **ReceiverBehaviour** è una classe astratta che incapsula la logica di ricezione dei messaggi, come mostrato nel Listato 7. La ricezione è non bloccante in quanto questa classe eredità, a sua volta, da *TickerBehaviour*.

Nello specifico, i messaggi possono essere ricevuti sia dall'interno, da altri *Behaviour*, sia dall'esterno, tramite BLE. Per i dettagli vedere il paragrafo seguente.

Listing 7: Logica del *Behaviour* con capacità di ricevere messaggi

```

1 void ReceiverBehaviour::onTick() {
2     struct Event event;
3
4     while (queue->pop(event)) {
5         String type = String(event.type);
6         String description = String(event.description);
7         ...
8
9         if (!type.isEmpty() && !description.isEmpty()) {
10             handle(event);
11         }
12     }
13 }
```

Interazione. L'interazione è totalmente basata sullo scambio di messaggi, riducendosi a pura comunicazione. In questo scenario il client, ovvero l'*Agent* utilizza l'interfaccia **MessagingService** che viene implementata da:

- **BLEMessaging:** incapsula i dettagli implementativi della comunicazione BLE, come la creazione dei *Services* e delle *Characteristics* visti nella Sezione 5.3, del server in ascolto di richieste di connessione, con i metodi per notificare i valori al *Gateway* e per ricevere i comandi di irrigazione.

Listing 8: Servizio di Messaggistica via BLE

```

1 class BleMessaging : public MessagingService, public
2     BLECharacteristicCallbacks {
3
4     public:
5         explicit BleMessaging() {
6             BLEDevice::init("Micro-irrigation");
7             this->server = BLEDevice::createServer();
8
9             Service::createService(server, EnvironmentalService)
10            ->addCharacteristic(HumidityCharacteristic, "Humidity 0 to 100%");
11            ->addCharacteristic(TemperatureCharacteristic, "Temperature 0-60 C");
12            ->start();
```

```

12     Service :: createService(server , BatteryService)
13     ->addCharacteristic(BatteryCharacteristic , "Battery Level 0 to 100%")
14     ->start();
15
16     Service :: createService(server , SoilService)
17     ->addCharacteristic(SoilMoistureCharacteristic , "Soil Moisture 0 to 100%
18         ")
19     ->start();
20
21     Service :: createService(server , WeatherService)
22     ->addCharacteristic(RainCharacteristic , "Amount of precipitations in
23         millimeters")
24     ->addCharacteristic(LightCharacteristic , "Light intensity in lux")
25     ->start();
26
27     Service :: createService(server , IrrigationService)
28     ->addCharacteristic(IrrigationCharacteristic , "Sprinkler status: On or
29         Off" , this)
30     ->start();
31
32     server->getAdvertising()->start();
33 }
34
35 void notifyCharacteristic(const char* serviceUUID , const char*
36     characteristicUUID , int measurement) {
37     BLECharacteristic* characteristic = getCharacteristic(serviceUUID ,
38         characteristicUUID);
39     characteristic->setValue(measurement);
40     characteristic->notify();
41 }
42
43 void onWrite(BLECharacteristic* characteristic) {
44     String s = String(characteristic->getValue().c_str());
45     s.trim();
46     this->eventListener->notify("in:irrigation/command" , s);
47 }
48 }
```

- **LocalMessaging:** incapsula la logica di comunicazione inter-task, avvalendosi dell'uso di una **message queue**, fornita da *FreeRTOS*.

Listing 9: Servizio di Messaggistica inter-task

```

1 class LocalMessaging : public MessagingService {
2 public:
3     void publishOn(String data , String topic) {
4         int size = listenerOnTopic->size();
5         for (int i = 0; i < size; i++) {
6             EventListener* l = listenerOnTopic->get(i);
```

```

7         l->notify(topic, data);
8     }
9 }
10
11 void subscribeTo(EventListener* subscriber, String topic) {
12     int element = 0;
13     int size = this->listenerOnTopic->size();
14     bool isAlreadySubscribed = false;
15
16     while (!isAlreadySubscribed && element < size) {
17         if (this->listenerOnTopic->get(element)->getUUID() == subscriber->
18             getUUID()) {
19             isAlreadySubscribed = true;
20         }
21         element++;
22     }
23
24     if (!isAlreadySubscribed) {
25         this->listenerOnTopic->add(subscriber);
26     }
27
28 void unsubscribeFrom(EventListener* subscriber, String topic) {
29     this->listenerOnTopic->remove(subscriber);
30 }
31 };

```

- **ProxyMessaging:** incapsula la logica di inoltro dei messaggi verso uno o più sistemi di messaggistica. Come mostrato nel Listato 10 i messaggi provenienti dall'esterno vengono indirizzati agli *EventListener* implementati dai *ReceiverBehaviour*, mentre i messaggi provenienti dall'*Agent* vengono indirizzati verso l'esterno ma anche verso gli altri *ReceiverBehaviour*. Quest'ultimo punto è stato necessario per evitare l'utilizzo di strutture dati complesse come la *Map* sia perché nel framework Arduino manca sia per non crearne una ad-hoc ma non ottimizzata. Sarà compito di ogni *ReceiverBehaviour* valutare se un messaggio è di suo interesse o meno. Da questo consegue anche che è stato necessario rendere il *ProxyMessaging* un *ReceiverBehaviour* per evitare di memorizzare il riferimento degli *EventListener* sia nella classe *BLEMessaging* sia in *LocalMessaging*.

Listing 10: Proxy di comunicazione

```

1 class MessagingProxy : public MessagingService, public ReceiverBehaviour {
2     public:
3         void publishOn(String data, String topic) {

```

```

4     String outTopic = String("out:") + topic;
5     ReceiverBehaviour::notify(outTopic, data);
6 }
7
8 void subscribeTo(EventListener* subscriber, String topic) {
9     localMessaging->subscribeTo(subscriber, topic);
10 }
11
12 void unsubscribeFrom(EventListener* subscriber, String topic) {
13     localMessaging
14         ->unsubscribeFrom(subscriber, topic);
15 }
16
17 void handle(Event& event) override {
18     String eventType = String(event.type);
19     String data = String(event.description);
20
21     if (eventType.startsWith("in:")) {
22         eventType.remove(0, 3);
23         localMessaging->publishOn(data, eventType);
24     } else if (eventType.startsWith("out:")) {
25         eventType.remove(0, 4);
26         localMessaging->publishOn(data, eventType);
27
28         if (eventType != "internal") {
29             bleMessaging->publishOn(data, eventType);
30         }
31     }
32 }
33 }
```

6.1.2 Gateway

Tale device ha un ruolo fondamentale nella raccolta e gestione dei messaggi in arrivo da ESP32 verso AWS, e viceversa. Inoltre è lo stesso device su cui è fisicamente installata la telecamera oltre agli applicativi per la sorveglianza e per il pilotaggio del Drone.

Nel sottomodulo “control-unit-irrigation”, dedicato alla gestione dell’irrigazione, il Gateway resta periodicamente in ascolto dei dati inviati dall’IoT device, e li memorizza temporaneamente in SQLite. Ogni 10 minuti, vengono prelevati i dati memorizzati nel db e aggregati applicando l’operatore di media ed inviati al cloud. Di seguito viene presentato un diagramma delle Classi relativo a tale sottomodulo.

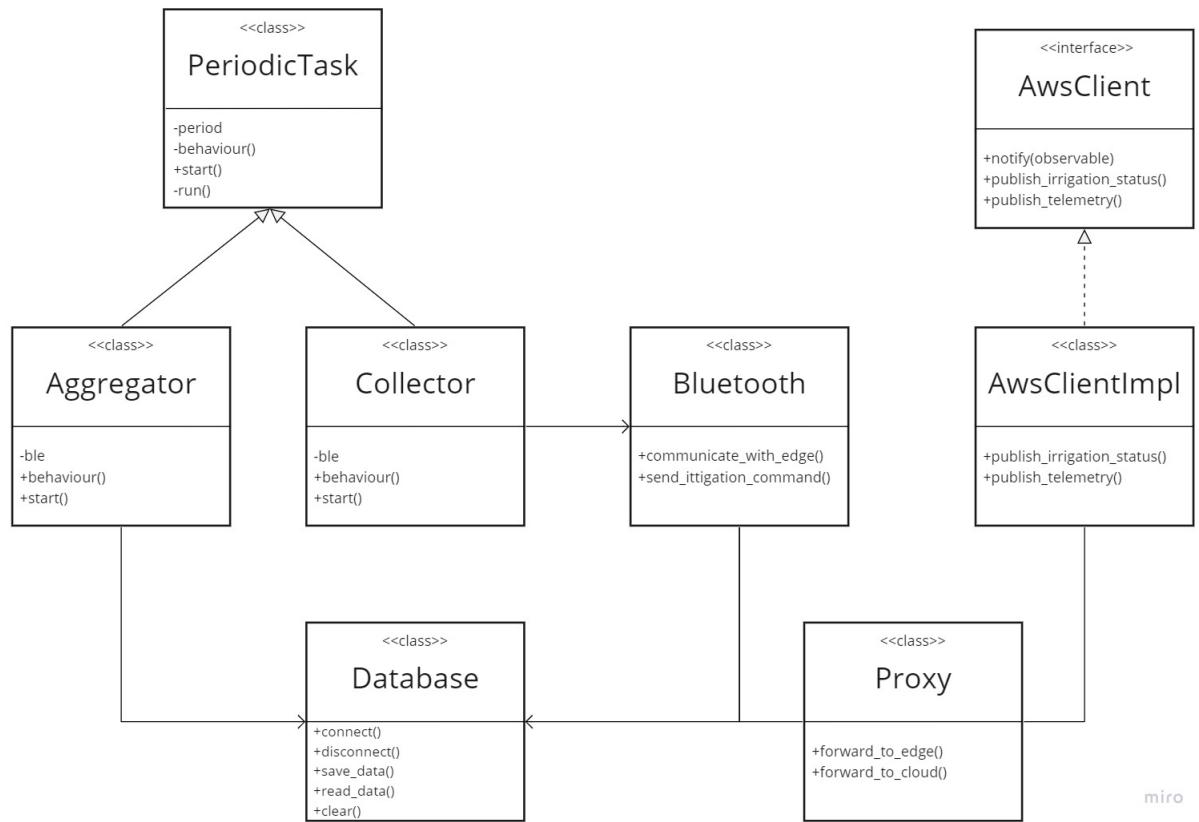


Figura 27: Control Unit Irrigation Class Diagram

L’aggregazione è parte fondamentale per due motivi:

1. permette al cloud di decidere se irrigazione sulla base della media giornaliera piuttosto che sui dati attuali;
2. permette di ridurre i dati inviati al cloud, riducendo i costi legati ad AWS IoT Core.

L’aggregazione e l’invio periodico dei dati, però, può essere fatto solo se c’è connessione Internet. Dopo l’aggregazione, i dati presenti sul db vengono cancellati, in quanto temporanei. Tra le componenti del Gateway, la più importante per la feature dell’irrigazione è la classe **Aggregator**, il cui codice è mostrato in parte nel Listato 11.

Listing 11: Aggregatore dei parametri ambientali

```

1 def aggregate_environmental_measurements(self, data):
2     environmental_parameters =
3         ["temperature", "humidity", "rain", "soil-moisture", "light-intensity"]
4
5     measurements = [self.get_values_of_type(data, environmental_parameter) for
6                      environmental_parameter in environmental_parameters]
7
8     average_measurements = [
9         round(statistics.mean(valuesOfMeasurement), 2)
10        if valuesOfMeasurement != [] else 0
11        for valuesOfMeasurement in measurements
12    ]
13
14     return list(zip(environmental_parameters, average_measurements))

```

Per la ricezione dei parametri ambientali (che avviene nella classe `Bluetooth`), il Gateway si avvale della libreria “`Bleak`”, un progetto open-source dedicato all’implementazione in Python di BLE lato client e che fornisce le API necessarie per effettuare la connessione e per scambiare i dati in formato binario. Per poter implementare meccanismi come quello della *Notification*, Bleak internamente fa largo uso di `asyncio`, un’altra libreria che apre le porte alla programmazione asincrona e che rende molto più elegante e affidabile la gestione dei vari scambi di messaggi tra client e server nell’ambito Bluetooth.

Listing 12: BLE Write example with asyncio and Bleak

```

1 async def write_irrigation_command(self, command):
2     try:
3         bytes_to_send = bytearray(map(ord, command))
4         await self.client.write_gatt_char(IRRIGATION_CHARACTERISTIC_UUID,
5                                         bytes_to_send)
6     except Exception as e:
7         print("Write Exception: ", e)

```

Nel codice fornito viene riportato il metodo dedicato alla Write, che corrisponde all’invio del comando di irrigazione da parte del Gateway all’Edge device. La stringa “`command`” conterrà il valore “`on`” proveniente direttamente dal MoM e quindi da uno dei due client.

Si noti la presenza di `async` e `await` in puro stile Event Driven, un po’ rimandante ai contesti asincroni di NodeJS e quindi Javascript.

Supporto seriale. Infine, per supportare anche device non dotati di BLE, è stato implementato anche un modulo per la comunicazione BLE tramite ESP32

che, a sua volta, si interfaccia con il *Gateway* via seriale. In tal modo il *Gateway* è svincolato dalla gestione del BLE.

Questa feature è stata implementata attraverso il **pattern Adapter**. La classe *BleAdapter* incapsula la comunicazione seriale, rendendola trasparente. L’interfaccia è la medesima della classe *Bluetooth* in modo da preservare il polimorfismo e, in generale, la modularità.

Su ESP32 è stata implementata la logica per creare un BLE central, o anche noto come BLE client, che effettua una scansione dei device BLE nei dintorni e si collega periodicamente all’IoT Device.

6.1.3 AWS Lambda

La logica di Decision Making, ovvero la decisione di irrigare, è presa sui dati giornalieri dei sensori. In particolare, viene effettuata una seconda aggregazione dei dati pre-aggregati lato *Gateway*, in modo far pesare ancor più lo storico giornaliero. Le regole per irrigare sono le seguenti:

- la temperatura non deve essere superiore a 25°C;
- l’umidità relativa non deve essere superiore a 40%;
- l’umidità del terreno deve essere inferiore a 30%;
- la luminosità solare deve essere inferiore a 66000 lux;
- le precipitazioni devono essere inferiori allo 0.2 mm.

L’implementazione di tale logica è stata incapsulata in una AWS Lambda, come mostrato nel Listato 13 ovvero una funzione distribuita e modulare che interagisce con AWS IoT Core e con altri servizi Amazon come DynamoDB.

Le AWS Lambda sono componenti molto più fini dei microservizi, in quanto sono funzioni fornite come servizio serverless e basate su un’architettura event-driven.

Listing 13: AWS Lambda che implementa la logica di irrigazione

```
1  async function irrigation_decision(measurement) {  
2      if (measurement.soil_moisture < 30 && measurement.rain < 3 &&  
3          measurement.temperature < 25 && measurement.humidity < 40 &&  
4          measurement.light_intensity < 66000) {  
5              return "on";  
6          }  
7      }  
8  
```

```
6      }
7      return "off";
8  }
9
10 exports.handler = async () => {
11   try {
12     let data = await readData();
13     let m = aggregate_measurements(data);
14     let irrigation_command = await irrigation_decision(m);
15
16     if (irrigation_command == "on") {
17       let command = JSON.stringify({
18         irrigation: irrigation_command
19       });
20       await publishMessage(brokerConfig, IRRIGATION_COMMAND_TOPIC, command);
21     }
22   } catch(e) {
23     console.log("Connection closed");
24   }
25};
```

6.2 Fertilizzazione con Drone

Non esiste un vero e proprio SDK per DJI Tello rilasciato dall'azienda produttrice, ma dalla documentazione ufficiale si evince che è possibile connettersi ad una Socket esposta dal Drone stesso utile per la ricezione di semplici comandi di pilotaggio o per spedire un feedback sullo stato di vari valori esposti dal Drone, come ad esempio, la potenza del segnale WiFi, l'altezza in cui si trova, la velocità, lo stato della batteria, ecc. L'indirizzo IP e la porta UDP di default attivi sul Drone Tello sono 192.168.10.1:8889. A questa Socket ci si deve connettere per mezzo di un thread dedicato, per consentire di non bloccare il main thread e permettergli di continuare a svolgere le operazioni principali. Una volta connessi, si inviano dei messaggi di testo predefiniti e aventi una certa semantica che il Drone riuscirà a riconoscere. Alcuni comandi presi dalla documentazione ufficiale [3] ed utilizzati nel progetto sono “`forward xx`” (procedi in avanti di xx centimetri), “`cw xx`” (gira il Drone sull'asse orizzontale in senso orario di xx gradi, senza modificare l'altitudine), “`Battery?`” (restituisci lo stato attuale della batteria, richiesta effettuabile anche mentre il Drone è in volo), “`Time?`” (restituisci il tempo di volo attuale).

Vengono riportati due snippet di codice posizionati in due metodi diversi della classe `tello.py`, che raffigurano l'apertura della socket per l'invio del comando richiesto e la ricezione del messaggio di callback da parte del Drone. Il messaggio di risposta verrà salvato in un array e opportunamente letto dal controller per intercettare eventuali errori fisici del veivolo, e quindi per dare un feedback realistico all'utente tramite Telegram.

Listing 14: Drone commands list for ground feeding

```
1 # send command to Tello
2 command = "forward 500"
3 client_socket.sendto(command.encode('utf-8'), self.address)
4
5 [...]
6
7
8 # listens for responses of Tello
9 while True:
10     try:
11         data, address = client_socket.recvfrom(1024)
12         Tello.LOGGER.debug('Data received from {} at client_socket'.format(address))
13         drones[address]['responses'].append(data)
14     except Exception as e:
```

```
15     Tello.LOGGER.error(e)
16     break
```

La scelta del linguaggio Python per questo e altri sottomoduli è motivata da una sintassi di facile lettura e comprensione, nonché da un netto snellimento del codice che ha contribuito ad un risparmio di tempo di realizzazione. Avendo provato più soluzioni, è stato notato che l'SDK per il client MQTT di AWS scritto in Python risulta più facile da utilizzare e la libreria sembra di qualità migliore rispetto, ad esempio, alla versione in NodeJS. Anche per quanto concerne la comunicazione con il Drone, come si è potuto constatare dallo snippet di cui sopra, sono venute in aiuto delle eleganti librerie Python per la comunicazione tramite Socket.

Sarà quindi il Gateway (Raspberry Pi 4) a svolgere il ruolo di pilota del Drone o se vogliamo di joystick per impartire i comandi in volo. Tuttavia, per svolgere questo compito dovrà rimanere costantemente connesso a Tello tramite WiFi. Si è ritenuto quindi necessario l'acquisto di un dongle WiFi da apportare al Raspberry Pi, per far sì che vi siano due schede di rete WiFi attive su indirizzi IP diversi, una per la comunicazione WAN verso l'esterno, e quindi con AWS, l'altra per la comunicazione esclusiva con il Drone.

Una breve nota sulla logica interna: a seguito dell'invio della richiesta di fertilizzare, si prevede di notificare l'utente che il Drone ha preso il volo correttamente e che sta per iniziare la fertilizzazione. Le condizioni scelte per prevedere un decollo andato a buon fine sono un buono stato di batteria, un tempo di volo maggiore di zero e un buon segnale WiFi.

Di seguito viene riportato uno snippet di codice riguardante il percorso da effettuare. Come simulazione si è ipotizzato campo di viti, quindi un percorso in avanti di 2 metri (Tello SDK consente arrivare fino a 5 mt, oltre i quali non è consentito andare per non rischiare perdite di segnale WiFi), per poi girare di 180 gradi, tornare indietro e atterrare nella stessa posizione di partenza, pronto per il prossimo decollo. A percorso terminato, viene effettuata una *publish* sul topic “drone/status” per notificare l'utente che l'intera operazione è andata a buon fine.

Listing 15: Drone commands list for ground feeding

```
1 def start_feeding(self):
2     if(self.really_took_off()):
3         self.drone.forward(200)
4         self.drone.cw(180)
5         self.drone.forward(200)
6         self.drone.cw(180)
7         battery = self.drone.get_battery()
8         self.drone.land()
9         self.aws_client.publish_drone_status(battery, False)
10        self.aws_client.disconnect_broker()
```

Se da una parte abbiamo tutta la logica basata sull'SDK fornito da Tello e sulla gestione del decollo, dall'altra si utilizza la libreria AWS IoT SDK [1] (interamente scritta in Python) per interagire con AWS IoT Core. Il sottosistema potrà in questo modo avvalersi dell'uso di MQTT e del Broker di AWS per poter spedire comandi in tempo reale sullo stato del Drone e per ricevere la richiesta di decollo da parte di Telegram e Alexa. Il client dovrà quindi avere la possibilità sia di ricevere che spedire dati attraverso MQTT.

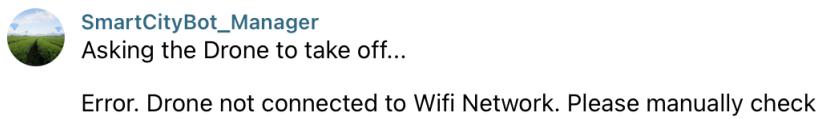


Figura 28: Drone status feedback after sending start command

Tale AWS-client è stato sviluppato in un'unica classe, posta nel file `aws.py`. Lo stesso identico codice, ma con certificati X.509 diversi (diversa Thing) e topic diversi, è stato utilizzato anche per il sottosistema dedicato al gathering dei dati da parte del Raspberry di cui si è discusso nel paragrafo “Gateway”, e quindi per spedire i valori dei sensori in Cloud col fine di renderli persistenti in DynamoDB (classe `AWSClient` dell'immagine in Figura 27).

Di seguito viene riportata una porzione di tale classe. E' doveroso sottolineare l'incredibile semplicità con cui è possibile interfacciarsi ad una Thing creata in IoT Core: bastano poche righe di codice e si ha un nuovo dispositivo IoT pronto ad emettere dati. Questo grazie alle librerie fornite da Amazon, che in questo caso sono `awscrt` e `awsiot`.

Listing 16: AWS Client - Topic subscriber and message sender

```

1 def connect(self):
2     self.mqtt_connection = mqtt_connection_builder.mtls_from_path(
3         endpoint=self.endpoint,
4         cert_filepath=PATH_TO_CERT,
5         pri_key_filepath=PATH_TO_KEY,
6         client_bootstrap=self.client_bootstrap,
7         ca_filepath=PATH_TO_ROOT,
8         client_id=self.client_id,
9         clean_session=False,
10        keep_alive_secs=180)
11    connect_future = self.mqtt_connection.connect()
12    connect_future.result()
13
14    def subscribe_topic(self, topic):
15        self.subscribe_future, self.packet_id = self.mqtt_connection.subscribe(
16            topic=topic,
17            qos=mqtt.QoS.AT_MOST_ONCE,
18            callback=self.on_message_received)
19
20    # Callback when the subscribed topic receives a message
21    def on_message_received(self, topic, payload, **kwargs):
22        subscribe_result = self.subscribe_future.result()
23        self.notify(json.loads(payload))
24
25    def publish(self, topic, message):
26        self.mqtt_connection.publish(topic=topic, payload=json.dumps(message), qos=
mqt

```

6.3 Videosorveglianza:

Come descritto in precedenza, il dispositivo Gateway è connesso alla rete elettrica e internet e posto in una posizione strategica del campo agricolo. Abbiamo immaginato un contesto simil-reale in cui a tale device è collegata anche una telecamera di sorveglianza che sia in grado di rilevare movimenti di soggetti non autorizzati. In particolare, in un campo agricolo, soprattutto di notte, potrebbe circolare qualche animale scatenando quindi falsi positivi. Per questo motivo si è cercato di fare ricorso a tecniche di Machine Learning per il riconoscimento esclusivo di un corpo dalle sembianze umane.

L'hardware utilizzato è l'ultima versione da 5 MP della famosa Pi Camera module [5]. La scelta di questo articolo, oltre che legata alla buona qualità/prezzo, è scaturita dall'eccellente integrazione con Raspberry Pi. La connessione tramite

seriale permette di avere buone prestazioni in scrittura e lettura, lasciando libere le porte USB della board, che possono essere utilizzate per altri scopi.

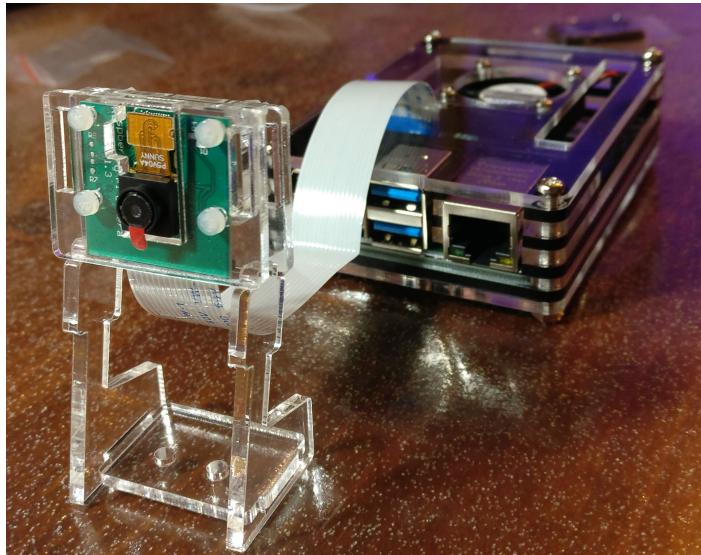


Figura 29: Raspberry Pi Camera and Raspberry Pi 4 - Gateway - Fog Computing

Si è voluto evitare di utilizzare un PIR sensor per rilevare i movimenti, a favore del pieno sfruttamento della telecamera. Per raggiungere questo obiettivo è stata utilizzata la libreria OpenCV, la quale si avvale di un Classificatore per il riconoscimento della persona. OpenCV è una libreria open-source molto apprezzata nell'ambito della Computer Vision e del Machine Learning. E' utilizzata dalle più importanti compagnie dell' ITC (Google, Yahoo, Microsoft, Intel, IBM, Sony ecc.) e vanta di più di 2500 algoritmi ottimizzati per risolvere i problemi di visione allo stato dell'arte [9].

Tra i più importanti algoritmi di object-detection in OpenCV troviamo il “cascode classifier”, algoritmo proposto nel 2001 da Paul Viola e Michael Jones. [10]. Ha bisogno di un numero elevato di immagini positive e negative per essere addestrato e non essendo tra i più recenti algoritmi esistenti oggi è anche a volte soggetto ad errori. Si è scelto in ogni caso di utilizzarlo perché direttamente integrato nella libreria di OpenCV, la quale fornisce anche dei modelli preaddestrati in formato XML pronti all'uso e caricabili in memoria attraverso il metodo `cv.CascadeClassifier.load`. Una volta caricati i modelli, la vera e propria de-

tention è possibile grazie al metodo `cv.CascadeClassifier.detectMultiScale`, il quale può anche mostrare dei rettangoli contenenti l'oggetto di interesse.

Implementare un intero progetto da zero per la gestione delle immagini catturate da OpenCV avrebbe richiesto uno sforzo non indifferente, sia in termini di studio che di tempo. Per questo motivo quindi il modulo software dedicato alla videosorveglianza è partito da un progetto preesistente con Licenza GNU v3 su GitHub [2] scritto in Python e che fa uso sia delle API di Telegram (le stesse usate per il Manager Bot) che di OpenCV per avere a disposizione un Bot software in grado di ricevere immagini in tempo reale su Telegram non appena si avverte un movimento, oltre ad avere la possibilità di scattare foto o girare video su richiesta. Parte della seconda Sprint è stata dedicata a ricerche e test su alcuni progetti realizzati dalla community, ma si sono rilevati non pienamente funzionanti o comunque poco adatti alle nostre necessità. Un altro importante motivo per cui è stato scelto il progetto sopra citato come base di partenza è la totale assenza dell'algoritmo di classificazione per il riconoscimento del corpo umano; questo ci ha permesso di modificarlo a nostro piacimento, di sperimentare la libreria OpenCV e di integrare il classificatore usando i metodi di cui sopra col fine di rispettare i Requisiti. Se usato *as-is*, il progetto preesistente spedisce una foto o un video al verificarsi di qualsiasi movimento, generando così molti falsi positivi. Pur essendoci molte altre modifiche apportate al software, di seguito viene riportata la porzione di codice dedicata all'attivazione dell'algoritmo di classificazione per selezionare i frame soltanto quando contengono una persona:

Listing 17: OpenCV Cascade Classifier

```

1 classifier = cv2.CascadeClassifier(model_filename)
2 # Upperbody recognition model
3 bodies = classifier.detectMultiScale(
4     gray,
5     scaleFactor=1.1,
6     minNeighbors=4,
7     minSize=(30, 30),
8     flags=cv2.CASCADE_SCALE_IMAGE
9 )
10
11 if len(bodies) > 0:
12     print("Human body detected !")
13     detected = True

```

Una volta rilevato il movimento, la Pi Camera viene attivata nuovamente per

scattare 5 foto nei prossimi 30 secondi, che verranno direttamente inviate nella chat Telegram che ha avviato.

6.4 Telegram Client

La Skill di Alexa e i due Bot di Telegram progettati nella fase di design sono i due client disponibili per interagire con il sistema.

Quando si interagisce con un Bot di Telegram, ogni comando spedito viene in realtà intercettato dai server del servizio, i quali fanno un redirect al Bot di interesse, che reagirà di conseguenza. L'unico modo per associare il proprio script all'infrastruttura Telegram è per mezzo di un Token generato durante la creazione del Bot. La creazione del Bot lato server è molto semplice: è sufficiente interagire con un Bot fornito da Telegram stesso che funge da creatore di altri Bot, il *Bot-Father*. Grazie a questo creatore si possono impostare le configurazioni iniziali (immagine che dovrà avere il nostro Bot, una descrizione, un nickname, ecc) e si può generare un Token che andrà poi ad essere utilizzato nel codice per poter instaurare la comunicazione e rendere il tutto funzionante.

Generalmente, una volta creato un Bot e messo in esecuzione, chiunque abbia accesso al suo nome identificativo univoco può interagirvi, e quindi avere accesso alle funzionalità sviluppate per la gestione smart del campo agricolo. Per questo motivo (e quindi per un motivo di sicurezza) si è voluto restringere il campo a due soli utenti (i membri del progetto), aventi i privilegi di interagire e impartire comandi.

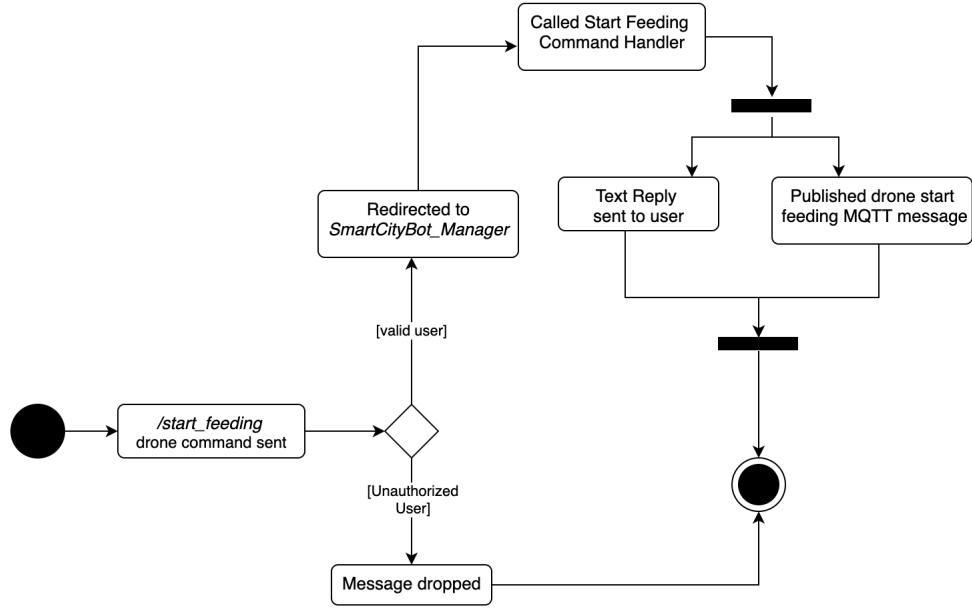


Figura 30: Telegram Manager state machine - Drone case

Nel caso di richiesta di avvio del Drone Tello ad esempio, i server Telegram ricevono il comando (che come già detto viene riconosciuto attraverso l'apposito “/” posto all'inizio della stringa) e lo inoltrano al Manager Bot. L'inoltro avviene solo nel caso in cui l'utente è autorizzato ad interagire con il Bot in questione. Tramite una funzione di callback il programma reagirà all'arrivo di tale messaggio con un messaggio di risposta (sulla stessa pagina di chat dalla quale è stato inviato) e azionerà il Drone pubblicando un messaggio al Broker e usando l'apposito topic corrispondente semanticamente alla richiesta iniziale (che nel caso del Drone sarà `drone/command`).

Di seguito viene riportato del codice del modulo `smartcitybot_manager` raffigurante lo scenario di cui si è parlato sopra:

Listing 18: Handler ricezione comando su Telegram Bot

```

1 def __init__(self):
2     self.updater = Updater(BOT_TOKEN, use_context=True)
3     dispatcher = self.updater.dispatcher
4
5     dispatcher.add_handler(CommandHandler("start", self.start,
6                                         Filters.user(username=[USER_MARCO,USER_FRANCESCO])))
7
8     dispatcher.add_handler(CommandHandler("start_feeding", self.start_feeding,
9                                         Filters.user(username=[USER_MARCO,USER_FRANCESCO])))
10
11    dispatcher.add_handler(CommandHandler("start_irrigation", self.start_irrigation,
12                                         Filters.user(username=[USER_MARCO,USER_FRANCESCO])))
13
14    dispatcher.add_handler(CommandHandler("help", self.help,
15                                         Filters.user(username=[USER_MARCO,USER_FRANCESCO])))
16
17 [...]
18 def start_feeding(self, update: Update, context: CallbackContext) -> None:
19     self.rep.message.reply_text('Asking the Drone to take off... ')
20     self.client.publish_start_drone()
21 [...]
22 def publish_start_drone(self):
23     message = {"drone": "start"}
24     self.client.publish(DRONE_COMMAND_TOPIC, message)

```

Il funzionamento del Bot relativo alla sorveglianza è del tutto similare, con l'unica differenza che nei metodi di callback, anziché interagire con un Broker esterno, viene avviata la telecamera di sorveglianza scattando foto, girando video, o facendo partire il classificatore per l'intrusion detection.

6.5 Alexa Skill Client

In questa sede vale la pena di spendere qualche parola su come si è potuto interagire con l'assistente vocale.

Quando si crea una Skill per Alexa, occorre accedere alla console di creazione del modello offerta da Amazon e descrivere attraverso un file JSON quali sono gli Intent (sia custom che predefiniti) che un utente può invocare all'assistente vocale. Gli Intent possono essere visti come un set di azioni, o affermazioni (scritte sotto forma di linguaggio naturale) invocabili dall'utente per ottenere una risposta.

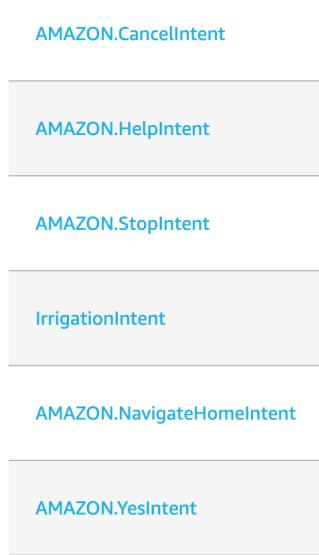


Figura 31: Alexa default and custom Intents for Agricoltore Smart Skill

Nel nostro caso si vuole chiedere ad Alexa di irrigare. Nella lingua italiana, vi sono molti modi per chiedere di effettuare questa operazione. Ogni possibilità linguistica è stata codificata nell'Intent, con frasi del tipo: “avvia pompa d'acqua”, “inizia irrigazione”, “irriga”, “comincia ad irrigare”, ecc. Tale modello creato sarà addestrato e precompilato così da rispondere all'Intent con un'azione programmata dallo sviluppatore. Il modello di interazione linguistica presente nella console sopra descritta può essere visto come il front-end dell'applicazione. Ogni front-end ha un suo back-end, che in questo caso è una Lambda collegata.

Nella Lambda quindi, nell'*handler* della richiesta di irrigare, si è voluta effettuare una *publish* al topic `irrigation/command`, topic utilizzato anche da Telegram e dalla Lambda dedicata all'irrigazione automatizzata. Per ragioni di tempo quella appena descritta è stata l'unica funzionalità creata. Nelle implementazioni future si pensa di estendere la Skill con altri Intent quali l'avvio del Drone per la fertilizzazione (“Alexa, avvia drone”) o un resoconto del momento migliore per irrigare basandosi sui dati raccolti dai sensori nel terreno (“Alexa, qual’è il momento giusto per irrigare?”, lavoro già svolto dalla Lambda dedicata all'irrigazione automatizzata).

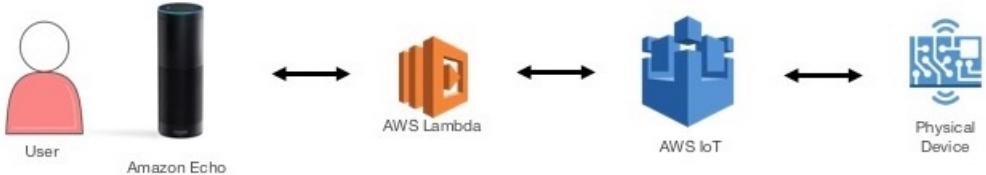


Figura 32: Alexa interactions from User to ESP32 with Pump relay

Di seguito viene riportato un frammento di codice della Lambda dove si può notare il metodo di callback chiamato a seguito dell’invocazione vocale dell’Intent.

Listing 19: Start irrigation command sent from Alexa Skill

```

1 def handle(self, handler_input):
2     speak_output = "Ok, avvio la pompa per l'irrigazione!"
3
4     # START IRRIGATION
5     payload = json.dumps(
6         { 'state':
7             {
8                 'desired': { 'irrigation': 'on' }
9             }
10        })
11    client = boto3.client('iot-data', region_name='us-west-2')
12    ret = client.publish(
13        topic='$aws/things/gateway/shadow/update',
14        qos=1,
15        payload=payload
16    )
17
18    return (
19        handler_input.response_builder
20            .speak(speak_output)
21            .response
22    )
23

```

Si noti, che per pubblicare il messaggio sul topic da AWS Lambda è stata utilizzata la libreria `boto3`. Boto3 è parte integrante di AWS SDK in Python e rende semplice l’integrazione della propria applicazione o script con i servizi AWS quali Amazon S3, Amazon EC2, Amazon DynamoDB ed altri. Permette quindi di creare, modificare cancellare e gestire risorse AWS attraverso un semplice script Python. Per quanto riguarda le Skill di Alexa, `boto3` funge da collante tra la

Skill creata e i servizi Amazon, dando libero spazio alla creazione di un servizio enterprise di alto livello per facilitare la vita dell'utente finale.

In questo caso Boto3 si interfaccia con AWS IoT Core, permettendo quindi alla Lambda dedicata alla logica dell'Alexa Skill di comunicare direttamente con il Broker interno all'ecosistema.

Il messaggio non viene pubblicato direttamente sul topic dedicato al comando di irrigazione, ma bensì al suo *Shadow*.

Come si evince dalla documentazione ufficiale, il servizio Device Shadow AWS IoT aggiunge copie shadow agli oggetti AWS IoT. Le copie shadow possono rendere disponibile lo stato di un dispositivo per app e altri servizi indipendentemente dal fatto che il dispositivo sia connesso a AWS IoT o meno. Gli oggetti AWS IoT possono avere più copie shadow con nome in modo che la soluzione IoT abbia più opzioni per connettere i dispositivi ad altre app e servizi.

7 Testing e performance

Il testing di ciascun sistema è stato effettuato manualmente, senza il supporto di strumenti di automazione. L'analisi delle performance, invece, è stata svolta solo per l'IoT Device per mancanza di tempo. L'IoT Device è stato sviluppato sul sistema operativo real-time free-rtos. Questo ha portato ad adottare un'architettura multi-tasking, in cui ogni Behaviour è eseguito come un task. Questo approccio ha permesso di sfruttare la caratteristica dell'ESP32 di essere dual-core. In tal modo è possibile la parallelizzazione di al più due task, mentre i task restanti vengono eseguiti con la tecnica dell'interleaving. Data l'architettura multi-tasking, l'analisi delle performance si focalizza sull'utilizzo della cpu, valutando se la parallelizzazione dei task consente di avere prestazioni migliori rispetto al solo interleaving.

Occorre però fare una precisazione sull'architettura multi-tasking del sistema free-rtos. Il main loop del framework di Arduino è un task eseguito sempre sul core 0 e con priorità elevata rispetto ai task di dominio, ovvero ai Behaviour. Nel caso il main loop sia vuoto, potrebbe comunque essergli data la cpu in quanto si tratta di un task ciclico non periodizzato. Questo comporta un context switch non necessario.

Gli scenari che hanno guidato l'analisi delle performance sono:

1. Multi-tasking & single-core: il main loop e i task sono eseguiti su un unico core, ovvero il core 0;
2. Multi-tasking & narrow dual-core: il main loop eseguito sul core 0 mentre i task sul core 1;
3. Multi-tasking & dual-core: il main loop eseguito sul core 0 mentre i task possono essere eseguiti sia sul core 0 sia sul core 1 in base alla disponibilità.

I risultati ottenuti sono mostrati nelle Tabelle seguenti.

	Multi-tasking & single core (%)										Average (%)
Core 0	21.5	10	9.8	29.4	12	10	75.6	30	13.7	10.9	22.3
Core 1	0.9	0	0.5	1.6	0.4	0.6	6.5	0.8	0.1	0.4	1.2

Tabella 1: Analisi sull'uso dei core: task eseguiti sul core 0

	Multi-tasking & narrow dual core (%)										Average (%)
	0.1	0.3	0.6	0.4	0.7	0	0.4	0	0.3	0.4	
Core 0	0.1	0.3	0.6	0.4	0.7	0	0.4	0	0.3	0.4	0.3
Core 1	69.6	19.7	15	11.3	81.5	40.6	26.5	19.4	15.4	13.2	31.2

Tabella 2: Analisi sull’uso dei core: task eseguiti solo sul core 1

	Multi-tasking & dual core (%)										Average (%)
	69.1	25.8	15.9	11.5	9.5	85.5	45.5	24.5	15.9	13.1	
Core 0	69.1	25.8	15.9	11.5	9.5	85.5	45.5	24.5	15.9	13.1	30.5
Core 1	1.9	0	0.5	0	0.6	66.2	0.1	0	0	0.7	7

Tabella 3: Analisi sull’uso dei core: task eseguiti arbitrariamente da FreeRTOS

Dall’analisi dei dati mostrati si evince che la differenza tra l’esecuzione dei task in modalità single-core e dual-core è minima, con una differenza del 10% sull’utilizzo dei core. Questo prova che ogni task non è bloccante, per cui il core su cui è eseguito viene rilasciato quanto prima, dando la possibilità agli altri task di essere eseguiti non appena siano nello stato *ready*.

7.1 Volume Dati

Per completezza è stata effettuata una stima dei volumi, quindi della quantità di row sia nel database SQLite locale nel Raspberry Pi, sia di oggetti salvati nel database non relazionale DynamoDB.

In particolare, il Raspberry Pi ogni minuto si abilita alle Notify da parte dell’ESP ed effettua una rilevazione della durata di 5 secondi, a seguito dei quali avviene di nuovo uno stop della sottoscrizione della ricezione dati. Ogni minuto il *Gateway* riceve 5 messaggi contenenti i valori ambientali dall’ESP, in quanto quest’ultimo effettua le misurazioni ogni secondo.

Per il calcolo dei volumi per DynamoDB, consideriamo che ogni 10 minuti il *Gateway* aggrega i presenti presenti in SQLite e li invia in un unico json. Per cui in quell’arco di tempo viene memorizzato un singolo object in DynamoDB.

	N. object in SQLite
al minuto	5 messaggi x 6 valori = 30
all'ora	30 x 60 = 1800
al giorno	1800 x 24 = 43200
all'anno	43200 x 365 = 15768000

Tabella 4: Volume dei dati in SQLite

	N. object in DynamoDB
ogni 10 minuti	1
all'ora	1 x 6 = 6
al giorno	6 x 24 = 144
all'anno	144 x 365 = 52560

Tabella 5: Volume dei dati in DynamoDB

Dati 52560 object all'anno, è possibile calcolare la dimensione del data storage in DynamoDB considerando che ogni object pesa all'incirca 150 byte, ovvero 0.0078GB. Questa dimensione, insieme al numero di read e write, può essere utilizzata per la stima dei costi AWS per la gestione di DynamoDB.

7.2 Comunicazione

La reattività della comunicazione dipende per quanto riguarda l'IoT Device, dal periodo con cui viene eseguito il ReceiverBehaviour. In particolare, il device non perde alcun messaggio, perché viene memorizzato in una coda dei messaggi. Quando la classe concreta che estende il ReceiverBehaviour sarà pronta (avrà terminato le sue computazioni), gestirà il messaggio. L'unico task che ascolta i messaggi in entrata è il MangeIrrigation che estende da ReceiverBehaviour. Al task è stato assegnato come periodo un secondo, e testando manualmente la comunicazione con il gateway il task è abbastanza reattivo. Per quanto riguarda invece la comunicazione con il cloud, è stato utilizzato il MoM fornito da AWS IoT Core. In questo caso è stato scelto il livello QoS 0 in quanto non è necessario rispedire un messaggio perso in quanto ormai è vecchio. Anche in questo caso la comunicazione è abbastanza reattiva.

Per quanto concerne gli integration test manuali sugli altri componenti, sono stati provati tutti i flow per il passaggio dei dati tra i vari layer, e quindi opportunamente testate tutte le feature sviluppate. Perché un flow funzioni interamente, i sensori devono essere opportunamente collegati, la connessione Bluetooth deve instaurarsi correttamente, il client MQTT deve essere programmato opportunamente per pubblicare oggetti JSON semanticamente corretti sui giusti topic, e AWS deve essere in grado di salvare i dati sul proprio database interno senza errori. Il tutto deve avvenire in maniera automatica e ciclica, cioè ad intervalli regolari (come nel caso del controllo da parte della Lambda per la decisione di irrigare, o per l'invio dei valori ambientali da parte del Gateway, ecc).

Anche il test sul Drone e sulla telecamera sono stati effettuati in maniera manuale. Nel caso della telecamera, è stato posto svariate volte un ostacolo in movimento davanti all'area di motion, seguito dall'apparizione dell'umano. Questo processo è stato ripetuto cambiando gli iperparametri da settare nel classificatore finché il numero di errori nella detection non si è ridotto al minimo. Per quanto concerne il Drone, sono state riscontrate varie difficoltà nel fargli effettuare il percorso prestabilito a fronte dell'invio del comando, non tanto per problemi di programmazione sbagliata, che man mano sono stati risolti, ma piuttosto per via di problemi fisici legati al dispositivo, quindi al suo motore, alla posizione delle eliche, o la scarsa luminosità nell'ambiente, la batteria scarica, ecc.

Le svariate prove sono tuttavia servite per comprendere le varie eccezioni lanciate dall'SDK del quadricottero e quindi per spedire all'utente un feedback il più accurato possibile sulle condizioni del Drone.

Il testing della Skill di Alexa è avvenuto invece attraverso “**Alexa Skills Kit Command Line Interface (ASK CLI)**”, tool molto utilizzato dagli sviluppatori di Skill per effettuare il deploy in produzione, ma anche per interagire con le funzioni di AWS Lambda nonché di *Skill Management API*. Tra le funzioni di quest'ultimo troviamo la possibilità di “parlare” con l'assistente vocale direttamente attraverso la linea di comando (ottimo sostituto dei classici Echo Dot disponibili in commercio) avendo così un feedback immediato sull'affidabilità della risposta e sul corretto funzionamento della parte appena sviluppata.

8 Analisi di deployment su larga scala

Attualmente il sistema vede due tipologie di device da installare direttamente sul campo, ovvero l’IoT Device e il Gateway. Questa configurazione di deployment è dovuta al mancato possesso di risorse necessarie per fare il deployment in un contesto reale, su larga scala. Difatti la nostra soluzione, si propone essere solo un prototipo.

Per il deployment su larga scala si può agire su due fronti: scindere l’IoT Device in più device indipendenti che hanno diverse responsabilità ed aumentare il numero di device da installare sul campo; aumentare anche il numero di Gateway da installare e garantire la loro coordinazione.

Sul primo fronte siamo già in grado di agire, in quanto occorre solo compilare e fare l’upload del software sui vari microcontrollori, isolando i Behaviour opportuni. In questo scenario, è possibile scindere l’IoT device in due o tre device indipendenti. Uno per la gestione dell’irrigazione in quanto si presume che il numero di pompe idriche sul campo sia inferiore al numero necessario di sensori per la rilevazione dei parametri ambientali. Un altro device, opzionale, per la rilevazione delle precipitazioni. E infine un terzo device per la rilevazione degli altri parametri ambientali. Il secondo ed il terzo device possono essere anche accorpati in un unico device qualora sia necessario installarne la stessa quantità e collocarli nelle medesime posizioni, che siano favorevoli per avere misurazioni quanto più precise.

Sul secondo fronte, invece occorre sviluppare una logica di coordinazione dei vari Gateway dal momento che si deve garantire il corretto funzionamento del sistema anche al fronte di un guasto ad uno o più gateway. Quindi, nel caso di guasti ad un gateway, gli altri devono rilevare il suo non funzionamento ed individuare un altro gateway (quello più vicino) in grado di prendersi la responsabilità di processare i propri dati ma anche quelli del gateway guasto. La logica di coordinazione è stata deliberatamente rimandata ad uno sviluppo futuro.

Agire sul secondo fronte è molto più importante in quanto è l’unico collo di bottiglia, nonché problema di resilienza. Se il gateway, per qualche motivo, dovesse disconnettersi dalla rete elettrica o addirittura avere un guasto, il sistema è bloccato. Se invece viene a mancare la connessione ad Internet non vi è alcun problema, dato che questo caso è stato risolto ed affrontato nella Sezione X.X (implementazione aggregatore).

Un’ulteriore soluzione per evitare la coordinazione dei vari gateway e la loro totale rimozione. Sicuramente è la scelta più economica inizialmente, in quanto si riducono il numero di componenti da acquistare. Ma successivamente potrebbe portare all’aumento dei costi per la gestione del cloud, in quanto il numero di messaggi che AWS IoT Core dovrà gestire sarà sicuramente maggiore non avendo più la fase di aggregazione dei dati sensoriali. Inoltre, dovendo sostituire la comunicazione BLE con quella HTTPS, si avrebbe un consumo maggiore di larghezza di banda ma anche di energia, portando ad un ulteriore incremento dei costi di manutenzione dei device.

È fondamentale ricordare i motivi per cui il gateway è stato introdotto, e per cui è importante tenerlo:

- ridurre il numero di messaggi inviati al cloud effettuando una prima aggregazione;
- ridurre il numero di device connessi in rete;
 - per problemi di sicurezza (evitare di esporre i device IoT alla rete);
 - per problemi di connessione (il segnale WiFi potrebbe non raggiungere tutti i punti del campo in modo uniforme e potrebbe non fornire una larghezza di banda sufficiente);
- è indispensabile per pilotare il drone; rappresenta quindi, il brain del drone.

Per quanto riguarda invece il contesto Cloud, il deployment su larga scala è molto più semplice perché il nostro sistema di base su un’architettura cloud pubblica serverless. Ne consegue che è l’azienda fornitrice dei servizi, Amazon in tal caso, a gestire, manutenere e garantire le caratteristiche fondamentali enunciate dal Reactive Manifesto (scalabilità, resilienza, disponibilità, reattività, guidato dai messaggi) per un sistema distribuito. In questo caso, la scalabilità è ottenuta on-demand, ovvero su richiesta. Questo porta numerosi vantaggi, quali l’aumento delle risorse solo quando davvero necessario, con tempi azzerati pagando un corrispettivo in base alle condizioni richiesti.

Focalizzandosi sui due Bot messi a disposizione invece, occorrerà una soluzione IaaS con un ambiente virtuale dedicato e accessibile a basso livello, così da poter installare e configurare le dipendenze del Bot e metterlo in esecuzione.

Viene naturale sfruttare anche qui un servizio Amazon per poter meglio interagire con le altre soluzioni AWS. Per il *deploy* e la *messa in produzione* di **SmartCityBot_Manager** si utilizzerà quindi EC2, che tra le altre cose mette a disposizione dei “Virtual computing environments”, anche conosciuti come “instances”. Si prospetta un uso poco intensivo di tale Bot, ma se dovesse crescere il numero di utenti, risulterà a quel punto semplice anche ed estenderne le risorse hardware sottostanti all’occorrenza.

Oltre alla Scalabilità, disiegare il Bot in un server always-on avrà anche altri vantaggi, legati alla possibilità per l’utente di interagire ed acquisire importanti messaggi di errore in caso di malfunzionamenti del sistema sia nel Fog che nel Edge Layer. Ad esempio, se i gateway distribuiti nel proprio campo agricolo andranno tutti down per un calo di tensione della corrente, Telegram sarà comunque in grado di accorgersi del malfunzionamento grazie alla mancata risposta nei tentativi di comunicazione, e quindi di notificare l’utente di conseguenza.

Il Bot **SmartCityBot_Surveillance** invece dovrà essere per forza di cose installato ed eseguito sul Gateway stesso (quindi nell’ambiente Linux di Raspberry Pi 4). Questa scelta è forzata dal fatto che il Bot in questione è direttamente collegato ad OpenCV e quindi alla telecamera di sorveglianza. Questo limita anche l'affidabilità del sistema di sicurezza stesso, che potrebbe venire meno in caso di malfunzionamento dei vari gateway sparsi nell’ambiente. Per questo motivo, tra gli sviluppi futuri si prospetta un disaccoppiamento del mero Bot con un altro modulo applicativo indipendente che invece interagirà con la telecamera ed invierà (magari sfruttando HTTP) le immagini di interesse al Bot, che questa volta sarà presente in Cloud, alleggerendo l'utilizzo delle risorse hardware necessarie sul gateway.

EC2 potrà essere sfruttato anche per creare un applicativo Web per la visualizzazione in forma di chart e grafici a torta raffiguranti l’andamento continuo dei vari sensori, facendo uso dei dati resi persistenti in DynamoDB nel tempo. Anche qui, utilizzare una soluzione AWS anziché un’istanza VPS (Virtual Private Server) creata in un altro ambiente facilita di molto la gestione e l’interazione con gli altri servizi offerti dalla piattaforma.

9 Piano di lavoro

La divisione dei lavori è stata per lo più forzata dalla situazione pandemica attuale, quindi dalla distanza geografica tra i due membri del Team e dall'hardware a disposizione nelle case di entrambi. E' stato utilizzato un approccio Agile Scrum con l'aiuto dello strumento Trello per la gestione e l'assegnazione dei task da portare a termine.

Di seguito la lista delle parti svolte insieme, sia per mezzo di meeting verbali tramite Teams che di sessioni di pair programming.

- Decisioni sull'Architettura generale, stesura dei diagrammi e degli schemi descrittivi del sistema
- Decisioni sulla logica di funzionamento di ogni sottoparte del sistema
- Scelte implementative, linguaggi, studio delle librerie disponibili, studio di gruppo riguardo il background tecnologico e metodologico
- Stesura della relazione

In particolare:

- Setup iniziali dei device per lavoro da remoto (connessione `ssh` con Raspberry Pi)
- Studio architettura AWS IoT Core e creazione delle Thing e Policy
- Risoluzioni di problemi legati al funzionamento della connessione Bluetooth tra Raspberry Pi e ESP32
- Risoluzioni di problemi legati al funzionamento di AWS Lambda e di Alexa Skill
- Risoluzioni di problemi legati al decollo del Drone e alla realizzazione del percorso stabilito
- Implementazione client BLE su Raspberry Pi per scambio dati con Edge device ESP32
- Test di integrazione, scalabilità e performance

Insieme è stata svolta anche ogni sessione di stand-up meeting, di Backlog e di Planning delle nuove funzionalità, oltre all'assegnazione dei vari task basandosi sull'hardware a disposizione, sulle conoscenze personali pregresse e il tempo di sviluppo. Si è cercato di rendere il più possibile equa l'assegnazione di ogni task, cercando di livellarne l'effort e il tempo di implementazione.

Le scelte sull'assegnazione dei task sono state prese anche col fine di addentrarsi in contesti tecnologici nuovi e mai affrontati prima. In questo modo ogni membro del gruppo ha potuto apprendere nuovi concetti e tecniche.

Si noti inoltre che ogni feature è stata revisionata ed approvata dall'altro componente prima del merge sul branch `master`, così da consentire ad entrambi i membri del team di avere una visione il più dettagliata possibile di tutte le parti del sistema. Di seguito la lista delle feature svolte individualmente a seguito delle fasi di planning di gruppo. Per ogni feature sono da considerarsi momenti di pair programming dovuti ad eventuali dubbi o micro decisioni da prendere:

Marco Baldassarri

- Surveillance Telegram Bot, Manager Telegram Bot
- Videosorveglianza e sistema di rilevamento intrusione umana usando OpenCV e Telegram
- Fertilizzazione tramite Drone, creazione SDK e pilotaggio, integrazione con Telegram Bot, implementazione comunicazione e manual testing
- Creazione dei Readme file descrittivi per sotto progetti
- Creazione DynamoDB table e Rule per salvataggio dati telemetrici dei sensori
- Installazione e configurazione SQLite su Raspberry Pi e creazione di operazioni CRUD in Python per interazione con db
- Alexa Skill
- Ricerca migliore libreria per BLE Client e primi test di comunicazione con Server

Francesco Vignola

- Progettazione, implementazione e manual testing delle feature inerenti l'IoT Device:
 - Rilevazione dei parametri ambientali tramite sensori;
 - Azionamento della pompa con spegnimento automatico e manuale;
 - Implementazione della comunicazione BLE.
- Progettazione e realizzazione del circuito elettronico
- Aggregazione periodica dei dati salvati in SQLite, lato Gateway
- Logica di Decision Making sull'irrigazione, in AWS Lambda
- Creazione Repository su Bitbucket e setup dei progetti

Di seguito viene riportato un diagramma di Gantt che sintetizza la collocazione temporale della realizzazione delle varie feature e le risorse impiegate per svolgerle in termini di giorni/uomo.

Smart City Project

University of Bologna

10 Conclusioni

Si ritiene che il sistema realizzato nell’ambito di questo progetto sia riuscito nell’intento di offrire una soluzione smart ed eco-sostenibile al sempre più presente problema della siccità, quindi di risparmiare energia e acqua senza compromettere l’esperienza degli utenti del servizio. Tutti i requisiti inizialmente specificati sono stati soddisfatti. Il sistema è stato progettato in modo da essere altamente flessibile ed estendibile in base alle esigenze specifiche di ogni attività agricola.

Ogni componente è stato pensato per essere modulare: purché la logica e l’architettura di fondo venga rispettata è quindi possibile sostituire ogni parte con una versione più aggiornata (tecnologia più innovativa, un linguaggio nuovo ecc.).

Pur essendo un progetto di dimensioni notevoli, la maggior parte delle funzionalità proposte in fase di progettazione sono state rispettate. Come sviluppi futuri, è plausibile una naturale aggiunta di una GUI per fornire all’utente dei grafici sull’andamento dell’irrigazione e quindi sullo stato attuale delle piante.

Per motivi di tempo non sono state implementate le interrogazioni verso un servizio di Weather Forecast per contribuire nella scelta da parte della Lambda se irrigare o meno. Ci siamo quindi limitati ad interrogare il database di DynamoDB e interpretare i dati dei sensori a disposizione. Ciò è comunque risultato esaustivo per consentire una buona logica di irrigazione.

Non manca anche l’idea di un ampliamento della Skill per Alexa che porti ad un’interazione più forte con l’utente per consentire di acquisire più informazioni sulla condizione della piantagione e per impartire nuovi comandi, come quello della fertilizzazione tramite Drone.

Si pensa inoltre di voler rafforzare l’uso di algoritmi di Machine Learning per il riconoscimento, tramite telecamera in dotazione ai Droni moderni, di eventuali malattie parassitarie presenti nella coltura, oltre ad un rafforzamento della componente di videosorveglianza.

Riferimenti bibliografici

- [1] Amazon. *AWS IoT Core SDK Python*. URL: <https://github.com/aws/aws-iot-device-sdk-python>.
- [2] Pablo Chinea. *Telegram Surveillance Bot*. URL: <https://github.com/pchinea/telegram-surveillance-bot>.
- [3] DJI. *Tello SDK*. <https://dlcdn.ryzerobotics.com/downloads/tello/0228/Tello+SDK+Readme.pdf>.
- [4] Bruce Poel Douglass. *DESIGN PATTERNS FOR EMBEDDED SYSTEMS IN C. An Embedded Software Engineering Toolkit*. Newnes, 2011. ISBN: 978-1-85617-707-8.
- [5] Raspberry Foundation. *Raspberry Pi Camera*. URL: <https://www.raspberrypi.org/products/camera-module-v2/>.
- [6] Reggio Emilia LUCIANO RINALDI MAGDA C. SCHIFF Crpa Spa. *Droni in campo per fertilizzare il vigneto*. http://www.nutrivigna.it/media/documents/nutrivigna-www/articoli/RivAgr_7-8_2017_p50_Nutrivigna.pdf?v=20180418.
- [7] Mavoco. *The impact of IoT in agriculture – Smart Farming*. URL: <https://www.mavoco.com/iot-in-agriculture/>.
- [8] O'Reilly. *BLE GATT*. URL: <https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html>.
- [9] OpenCV. *About Open CV*. URL: <https://opencv.org/about/>.
- [10] OpenCV. *OpenCV Classifier*. https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html.
- [11] Telegram. *Bot Revolution*. URL: <https://telegram.org/blog/bot-revolution>.
- [12] Zambon-Cecchini-Egidi-Saporito-Colantoni. *Revolution 4.0: Industry vs. Agriculture in a Future Development for SMEs*. URL: https://www.researchgate.net/publication/330330957_Revolution_40_Industry_vs_Agriculture_in_a_Future_Development_for_SMEs.