

Relazione del progetto di Informatica III



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Laurea Magistrale in Ingegneria Informatica
I Anno

A.A. 2021/2022

Modulo di Progettazione e
Algoritmi

Studenti:

Wasim Essbai - 1060652

Matteo Locatelli - 1059210

Nicola Zambelli - 1053015

Sommario

Introduzione.....	1
Obiettivo	2
W3C.....	3
LoRaWan.....	4
MAPE-K loop.....	6
MQTT.....	7
ChirpStack	9
Code topiche.....	10
Api di integrazione	12
Iterazione 0	13
Analisi dei requisiti	14
R0 – Azioni di riconfigurazione per allungamento del ciclo di vita delle sentinelle:.....	14
R1 – Azioni di auto-diagnosi manutentiva di risoluzione autonoma dei guasti:	14
User-Stories	16
Lato edge device:.....	16
Lato application server:	17
Lato watchdog:	18
Deployment Diagram	19
Deployment Diagram Informale	19
Deployment Diagram UML	20
Iterazione 1	21
Goal di iterazione	22
Component diagram	22
Class diagram	22
Test con simulatore	24
LWN-Simulator.....	24
Simulazione	24
Iterazione 2	26
Goal di iterazione	27

Component diagram	27
Class diagram	28
Test comunicazione	30
Iterazione 3	32
Goal di iterazione	33
Component diagram	34
Class diagram	36
Il modulo <i>coder.py</i>	37
State machine watchdog	40
Analisi dinamica.....	41
Unit test	41
Integration test.....	43
Iterazione 4	44
Goal di iterazione	45
Component diagram	45
Class diagram	48
Analisi e verifica del funzionamento	49
Assenza di controllo	49
Introduzione del controllo	50
Iterazione 5	53
Goal di iterazione	54
Component diagram	54
Class diagram	56
Il <i>package</i> Payloads.....	57
L'algoritmo <i>check_nodes</i>	59
Pseudocodice:	60
Analisi complessità-tempo	61
Implementazione:	62
Analisi statica codice.....	63
Simulazione guasto.....	65
Scenario di guasto <i>edge-node</i>	65

Scenario di guasto <i>watchdog</i>	67
Risultati simulazione	67
Conclusioni.....	69
Sviluppi futuri.....	70

Indice delle figure

<i>Figura 1 Schema del pattern Digital Twin</i>	4
<i>Figura 2 Confronto delle principali tipologie di rete</i>	4
<i>Figura 3 Layer del protocollo LoRaWan</i>	5
<i>Figura 4 Schema del modello MAPE-K loop</i>	6
<i>Figura 5 Architettura dello stack ChirpStack</i>	9
<i>Figura 6 Scenario di auto-adattamento per allungare il ciclo di vita di un watchdog</i>	14
<i>Figura 7 Scenari di auto-rilevazione dei guasti</i>	15
<i>Figura 8 Deployment diagram informale</i>	19
<i>Figura 9 Deployment diagram in UML</i>	20
<i>Figura 10 Component diagram iterazione 1</i>	22
<i>Figura 11 Class diagram iterazione 1</i>	23
<i>Figura 12 Gateway dashboard ChirpStack</i>	25
<i>Figura 13 Esecuzione LWN-Simulator</i>	25
<i>Figura 14 Interfaccia avviamento simulazione con LWN-Simulator</i>	25
<i>Figura 15 Component diagram iterazione 2</i>	28
<i>Figura 16 Class diagram iterazione 2</i>	29
<i>Figura 17 Interfaccia grafica principale</i>	30
<i>Figura 18 Interfaccia di log edge-node</i>	30
<i>Figura 19 Chirpstack gateway dashboard</i>	31
<i>Figura 20 Component diagram iterazione 3</i>	35
<i>Figura 21 Class Diagram Iterazione 3</i>	36
<i>Figura 22 Class diagram modulo Coder</i>	37
<i>Figura 23 Rappresentazione dei messaggi</i>	39
<i>Figura 24 Macchina a stati finiti del funzionamento di un watchdog</i>	40
<i>Figura 25 Interfaccia grafica watchdog</i>	43
<i>Figura 26 Dashboard di ChirpStack per i device</i>	43
<i>Figura 27 Component diagram iterazione 4</i>	47
<i>Figura 28 Class diagram iterazione 4</i>	48
<i>Figura 29 Autonomia senza controllo adattativo</i>	49
<i>Figura 30 Interfaccia log EdgeCloudSoftware</i>	50
<i>Figura 31 Configurazione watchdog</i>	51
<i>Figura 32 Autonomia controllo adattativo</i>	51
<i>Figura 33 Component diagram iterazione 5</i>	55
<i>Figura 34 Class diagram iterazione 5</i>	56
<i>Figura 35 Class diagram package payloads</i>	58
<i>Figura 36 implementazione Algoritmo check_nodes</i>	62
<i>Figura 37 Complessità di McCabe watchdog</i>	63
<i>Figura 38 Complessità di McCabe edgenode</i>	64
<i>Figura 39 Complessità di McCabe AppServer</i>	64
<i>Figura 40 Complessità di McCabe dell'intero progetto</i>	64
<i>Figura 41 Schermata di log AppServer</i>	65
<i>Figura 42 Schermata di log simulazione guasto nodi edge</i>	66
<i>Figura 43 Rilevazione guasti</i>	66
<i>Figura 44 Rilevazione guasto watchdog</i>	67

Introduzione

Obiettivo

Il progetto ha come finalità la creazione di una piattaforma *edge-cloud* con la funzione di monitoraggio ambientale. Parte integrante del progetto sono i dispositivi **IoT** (*Internet of Things*), capaci di raccogliere dati con uno scarso consumo di risorse.

I vantaggi dell'utilizzo di una rete di dispositivi *IoT* per la raccolta di dati ambientali sono molteplici, come la vasta scalabilità della rete ed il basso costo di manutenzione, rispetto agli approcci tradizionali.

I protagonisti della rete di acquisizione sono i *watchdog*, gli *edge-node* ed il *cloud server*:

- I **watchdog** sono i nodi sentinella. Hanno il compito di campionare i dati e di trasmetterli ai nodi *edge*. Tali dispositivi, tipicamente, sono mobili e alimentati a batteria; l'obiettivo, quindi, risulta essere quello di massimizzare il loro ciclo di vita, minimizzando il consumo di risorse. Un protocollo di comunicazione *wireless* a basso profilo è il *LoRaWan* che sarà presentato in seguito. Tale tecnologia permette di connettere i *watchdog* ai nodi *edge* con un modesto impiego di energia, garantendo al contempo una comunicazione efficace.
- Gli **edge-node** hanno un ruolo centrale nel progetto. Il loro compito è quello di fare da *bridge* tra i nodi *watchdog* ed il *cloud server*. Essi si interfacciano con i *watchdog* attraverso il protocollo *LoRaWan* e con il *cloud server* per mezzo del protocollo **MQTT**, anch'esso uno standard *ISO* per la messaggistica leggera, che si appoggia al livello *TCP/IP* dell'internet e ampiamente impiegato in applicazioni *IoT*.
- Il **cloud server** ha la funzione di acquisire e immagazzinare le informazioni che arrivano dagli *edge-node*. Inoltre, sarà capace di analizzare e interpretare i dati, pianificando delle operazioni *self-adaptive* in modo da garantire un determinato livello di *dependability* del sistema, al fine di minimizzare ed ottimizzare l'intervento umano sui dispositivi fisici. Questo tipo di approccio permette di ottenere una maggiore qualità delle acquisizioni dei dati e di risparmiare sui costi di manutenzione e riparazione. Il grado di *dependability* garantito dipende in modo particolare da questa componente, che quindi va progettata con cura al fine di determinare azioni correttive mirate e precise, rimanendo allo stesso tempo generali in modo da adattarsi a più scenari possibili.

W3C

Il **World Wide Web Consortium**, anche conosciuto come **W3C**, è un'organizzazione non governativa internazionale che ha come scopo quello di favorire lo sviluppo di tutte le potenzialità del *World Wide Web* e diffondere la cultura dell'accessibilità della rete. La principale attività svolta dal W3C consiste nello stabilire standard tecnici per il *World Wide Web* inerenti sia ai linguaggi di markup che ai protocolli di comunicazione.

Nel nostro caso siamo interessati al paragrafo 4.1.10 della guida, relativo monitoraggio ambientale:

“Il monitoraggio dell’ambiente si basa in genere su molti sensori distribuiti che inviano i propri dati di misurazione a gateway comuni, dispositivi perimetrali e servizi cloud.

Il monitoraggio dell’inquinamento atmosferico, dell’inquinamento idrico e di altri fattori di rischio ambientale come polveri sottili, ozono, composti organici volatili, radioattività, temperatura, umidità per rilevare condizioni ambientali critiche può prevenire danni irreparabili alla salute o all’ambiente.”

Un altro paragrafo d’interesse è il 4.2.6, quello sui *Digital Twins*, utile per simulare le componenti.

“Un gemello digitale è una rappresentazione virtuale, ovvero un modello di un dispositivo o un gruppo di dispositivi che risiede su un server cloud o su un dispositivo edge. Può essere utilizzato per rappresentare dispositivi del mondo reale che potrebbero non essere continuamente online o per eseguire simulazioni di nuove applicazioni e servizi, prima che vengano distribuiti ai dispositivi reali.”

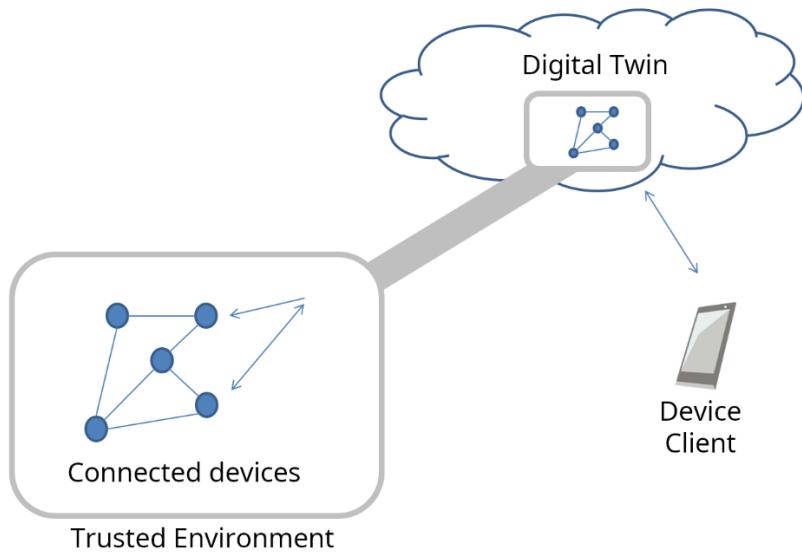


Figura 1 Schema del pattern Digital Twin

LoRaWan

La specifica *LoRaWan* è un protocollo di rete *LPWA* (*Low Power, Wide Area*) progettato per connettere in modalità *wireless "things"* alimentati a batteria in reti regionali, nazionali o globali e si rivolge ai requisiti chiave dell'*Internet of Things*, come bi-servizi di comunicazione direzionale, sicurezza end-to-end, mobilità e localizzazione.

Perché proprio LoRaWan?

LoRaWan offre una durata della batteria pluriennale ed è progettato per sensori e applicazioni che richiedono di inviare piccole quantità di dati su lunghe distanze. Di seguito si può osservare un confronto tra *LoRaWan* e le comunicazioni wireless maggiormente adottate, come le reti *LAN*(*local Area Network*) e le reti *Cellular*.

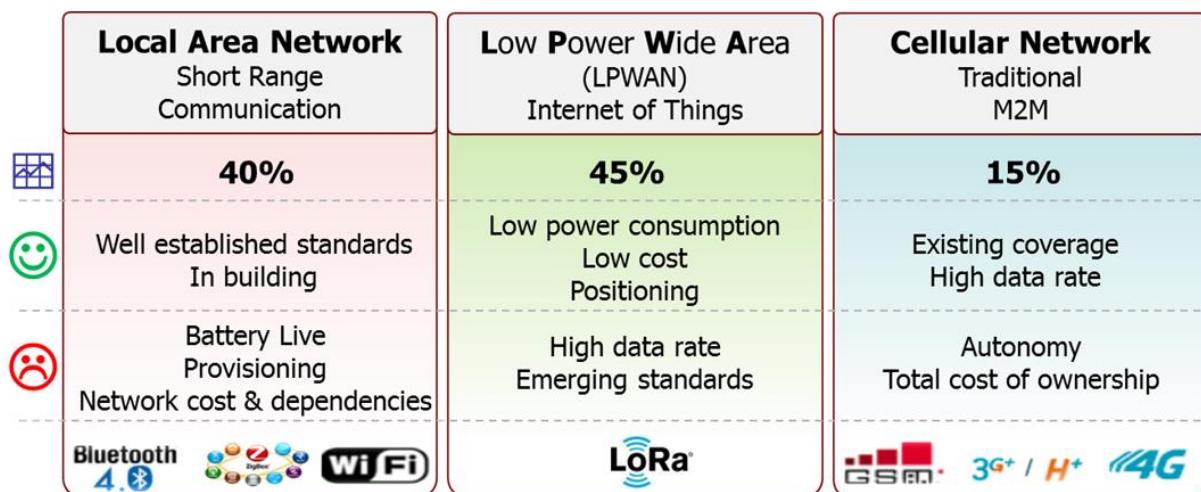


Figura 2 Confronto delle principali tipologie di rete

WiFi e BTLE sono standard ampiamente adottati, ma servono alle applicazioni che comunicano con dispositivi personali su piccole distanze.

La tecnologia cellulare è invece perfetta per applicazioni che richiedono un elevato throughput di dati, ma devono disporre di una notevole fonte di alimentazione.

LoRaWan definisce il protocollo di comunicazione e l'architettura di sistema (livello applicativo del modello *ISO/OSI*) ,mentre *LoRa* è lo strato fisico della telecomunicazione (livello1 del modello *OSI*), che consente il collegamento a lungo raggio.

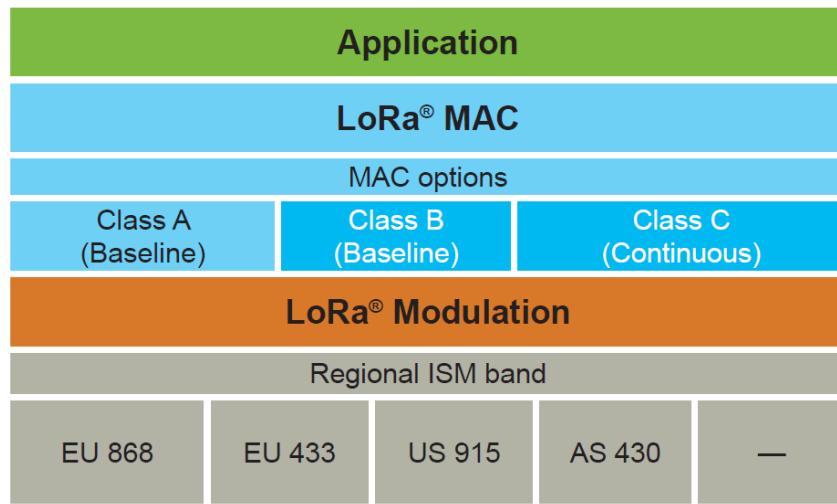


Figura 3 Layer del protocollo LoRaWan

LoRaWan per l'Europa

LoRaWan definisce dieci canali, suddivisi in

- Otto sono multi-data da 250 bps a 5,5 kbps
- Un singolo canale LoRa ad alta velocità di trasmissione dati, a 11 kbps
- Un singolo canale FSK a 50 kbps.

La potenza di uscita massima consentita da *ETSI (European Telecommunications Standards Institute)* in Europa è +14dBm. Ci sono restrizioni sul ciclo di lavoro, ma senza limiti di tempo massimo di trasmissione o di permanenza del canale.

MAPE-K loop

Il cosiddetto modello *MAPE-K control loop* è uno stile architetturale introdotto da IBM nel loro *white paper*: “*An architectural blueprint for autonomic computing*”. L’intento è quello di creare un ambiente informatico con le capacità di autogestione e auto-adattamento dinamico a seconda delle *business policies* implementate. Il ciclo è suddiviso in quattro funzioni principali:

- Monitor: Raccoglie i dati delle risorse gestite
- Analyze: Esegue complesse analisi dei dati e valuta i segnali dalla funzione monitor.
- Plan: Struttura le azioni necessarie per raggiungere gli obiettivi richiesti e crea o seleziona una procedura da attuare per la configurazione desiderata nella risorsa gestita.
- Execute: Modifica il comportamento della risorsa gestita utilizzando gli effettori, sulla base delle azioni consigliate dalla funzione di piano.

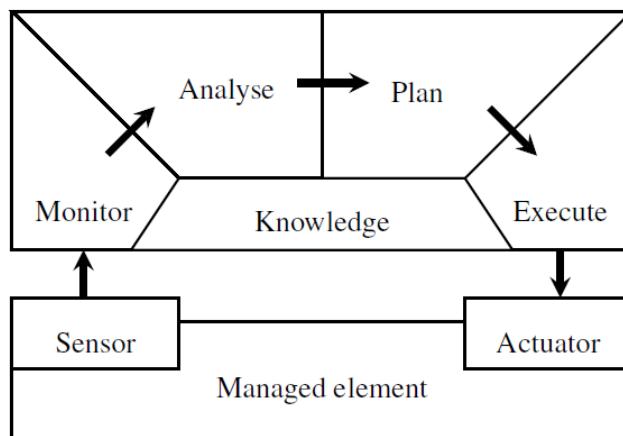


Figura 4 Schema del modello MAPE-K loop

L’obiettivo è quello di utilizzare questo modello concettuale inserendo un ciclo MAPE-K per ognuno degli *adaptation-goals*. Inoltre, verrà valutata la distribuzione delle funzioni MAPE, identificando, per ogni ciclo, l’attore del sistema che svolge una determinata funzionalità.

MQTT

MQTT (*Message Queue Telemetry Transport*) è un protocollo di messaggistica standard di *OASIS*, impiegato ampiamente in applicazioni *IoT*.

MQTT è un protocollo di tipo *publish/subscribe*, estremamente leggero, ideale per la connessione di dispositivi remoti con un *footprint* di codice ridotto ed una larghezza di banda di rete minima.

A differenza del paradigma di *request/response* di *http*, *MQTT* è basato su eventi e consente di inviare messaggi ad uno o più *client* che esprimono interesse per un certo *topic*. Un *topic* è una classe di messaggi che hanno caratteristiche comuni. Un nodo (detto *subscriber*) può chiedere di ricevere i messaggi pubblicati dai produttori di dati (detti *publisher*) relativi ad un certo *topic*, secondo dei parametri di qualità, che vengono indicati tipicamente con *Quality Of Service (QoS)*.

Questo tipo di comunicazione ha la caratteristica di essere asincrona dal momento che tra la pubblicazione di un messaggio e la sua lettura può intercorrere una certo intervallo di tempo, più o meno grande, che dipende dalla particolare configurazione in atto.

Questo tipo di architettura, inoltre, disaccoppia tra di loro i *client* per consentire una soluzione altamente scalabile senza dipendenze tra produttori di dati e consumatori di dati.

I principali vantaggi di *MQTT* sono:

- Leggerezza ed efficienza per ridurre al minimo le risorse richieste per il *client* e la larghezza di banda della rete.
- La possibilità di gestire la comunicazione bidirezionale.
- La possibilità di trasmettere messaggi di broadcast a gruppi di *client*.
- La scalabilità.
- Le specifiche dei livelli di *Quality Of Service (QoS)*, ad esempio per supportare l'affidabilità della consegna dei messaggi.
- Il supporto di sessioni persistenti tra dispositivo e server che riduce il tempo richiesto per riconnettere il *client* al *broker* su reti inaffidabili
- La possibilità di crittografare i messaggi con *TLS (Transport Layer Security)*, un protocollo crittografico usato nell'ambito delle telecomunicazioni e dell'informatica, che permette una comunicazione sicura *end-to-end* tra sorgente e destinatario su reti *TCP/IP*, fornendo autenticità ed integrità dei dati e operando sopra il livello di trasporto. I messaggi possono anche supportare protocolli di autenticazione *client*.

Gli attori principali nel protocollo *MQTT* sono il *broker* ed i *client*.

Il *broker* è responsabile della pubblicazione dei messaggi da parte dei *publisher* sulle code *topiche*, ovvero le code di messaggi relativi ad un certo topic. Inoltre, si occupa anche della notifica e della consegna dei messaggi pubblicati ai *subscriber* iscritti alle rispettive code topiche. Un *subscriber* può essere iscritto anche a più *topic* e, quindi, può ricevere diverse notifiche relative alla pubblicazione di messaggi su diverse code topiche.

Un *client MQTT* pubblica un messaggio su un broker ed altri *client* possono iscriversi al *broker* per ricevere quei messaggi. Ogni messaggio *MQTT* include un argomento che è il **topic**. Il *broker*, poi, usa i *topic* e la lista dei *subscribers* a ciascun *topic* per inviare messaggi ai *client* appropriati.

Un *broker MQTT*, inoltre, è in grado di memorizzare, in un *buffer*, i messaggi che non possono essere inviati a *client* non connessi. Ciò è molto utile in situazioni in cui le connessioni di rete non sono affidabili. Per supportare la consegna affidabile dei messaggi, il protocollo supporta tre diversi tipi di messaggi di **QoS**: 0 (al più una volta), 1 (almeno una volta), 2 (esattamente una volta).

ChirpStack

Lo stack *LoRaWAN Network Server open-source ChirpStack* fornisce componenti *open-source* per reti *LoRaWAN*. Insieme forniscono sia un’interfaccia web per la gestione dei dispositivi che le *API* per l’integrazione.

Il vantaggio di *ChirpStack* sta nella sua architettura modulare, la quale consente l’integrazione dello *stack* all’interno di infrastrutture esistenti. Tutti i componenti, inoltre, sono autorizzati in base alla licenza *MIT*.

Lo *stack* fornisce i seguenti componenti:

- Un *ChirpStack Gateway Bridge* per gestire la comunicazione con i *gateway LoRaWAN*
- Un *ChirpStack Network Server* che rappresenta un’implementazione del server di rete *LoRaWAN*
- Un *ChirpStack Application Server* che rappresenta un’implementazione del *LoRaWAN Application Server*

In Figura 5 viene mostrato come sono collegati i componenti del *ChirpStack LoRaWAN Network Server*, ovvero la sua architettura.

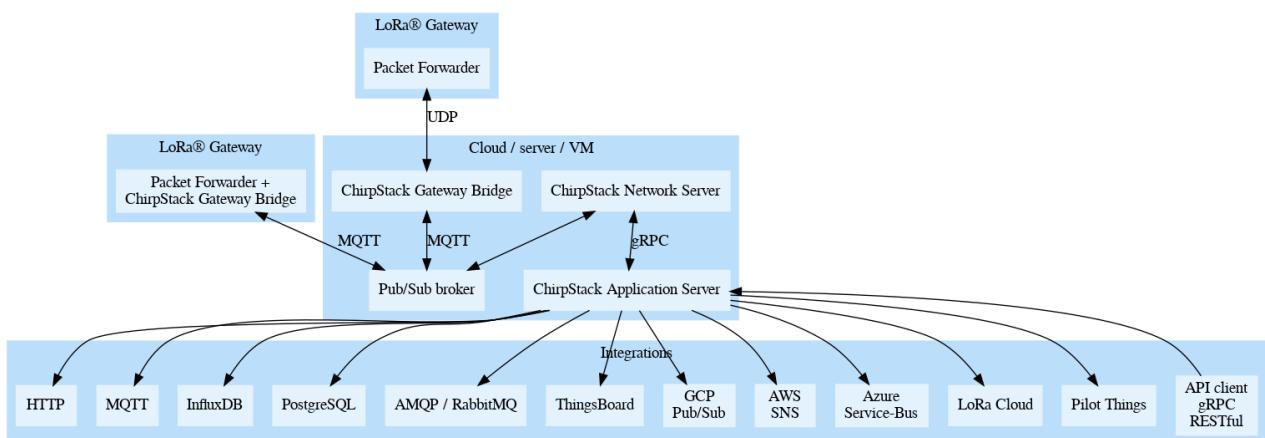


Figura 5 Architettura dello stack ChirpStack

I dispositivi *LoRaWAN* (non illustrati nel grafico sopra) sono i dispositivi che inviano dati al server di rete *ChirpStack* (attraverso uno o più *gateway LoRa*). Questi dispositivi potrebbero essere ad esempio sensori che misurano una serie di parametri ambientali (la qualità dell’aria, la temperatura, l’umidità, la posizione, ...).

Un *gateway LoRa* ascolta (di solito) 8 o più canali contemporaneamente e inoltra i dati ricevuti dai dispositivi a un server di rete *LoRaWAN* (in questo caso il server di rete *ChirpStack*). Il software in esecuzione sul *LoRa Gateway*, responsabile della ricezione e dell’invio dei dati, è chiamato *Packet Forwarder* (le implementazioni comuni sono *Semtech UDP Packet Forwarder* e *Semtech Basic Station Packet*

Forwarder). Si tratta di un programma in esecuzione sull'*host* di un *gateway Lora* che inoltra i pacchetti RF ricevuti dal concentratore a un server tramite un collegamento *IP/UDP* ed emette i pacchetti RF inviati dal *server*. Può anche emettere un segnale *beacon* sincrono GPS a livello di rete utilizzato per coordinare tutti i nodi della rete. La comunicazione è bidirezionale:

- *Uplink*: pacchetti radio ricevuti dal *gateway*, con metadati aggiunti dal *gateway*, inoltrati al server. Potrebbe anche includere lo stato del *gateway*.
- *Downlink*: pacchetti generati dal *server*, con metadati aggiuntivi, da trasmettere dal *gateway* sul canale radio. Potrebbe includere anche i dati di configurazione per il *gateway*.

Il *ChirpStack Gateway Bridge* si trova tra il *Packet Forwarder* e il *broker MQTT*. Trasforma il formato *Packet Forwarder* in un formato dati utilizzato dai componenti *ChirpStack*. Fornisce inoltre integrazioni con varie piattaforme *cloud* come *GCP Cloud IoT Core* e *Azure IoT Hub*.

Il *ChirpStack Network Server* è un server di rete *LoRaWAN*, responsabile della gestione dello stato della rete. È a conoscenza delle attivazioni dei dispositivi sulla rete ed è in grado di gestire funzioni di *join-requests* quando i dispositivi desiderano unirsi alla rete. Quando i dati vengono ricevuti da più *gateway*, il *ChirpStack Network Server* deduplica questi dati e li inoltra come un carico utile al *ChirpStack Application Server*. Quando *Application Server* deve inviare i dati a un dispositivo, il *ChirpStack Network Server* manterrà questi elementi in coda, finché non sarà in grado di inviare dati a uno dei *gateway*.

Il *ChirpStack Application Server* è un *LoRaWAN Application Server*, compatibile con il *ChirpStack Network Server*. Fornisce un'interfaccia web e *API* per la gestione di utenti, organizzazioni, applicazioni, *gateway* e dispositivi. I dati di *uplink* ricevuti vengono inoltrati a una o più integrazioni configurate.

L'applicazione finale riceve i dati del dispositivo tramite una delle integrazioni configurate. Può utilizzare l'*API ChirpStack Application Server* per programmare un *payload* di *downlink* sui dispositivi. Lo scopo di un'applicazione finale potrebbe essere analisi, avvisi, visualizzazione dei dati, attivazione di azioni, ecc...

Code topiche

Essendo MQTT un protocollo di tipo *publish/subscribe*, i messaggi vengono pubblicati su delle code topiche per consentire la consegna di messaggi ad uno o più *device* sottoscritti ad una particolare coda.

Lo stack *ChirpStack* definisce delle code topiche che vengono utilizzate sia per veicolare il flusso dati della rete, che per l'integrazione con applicazioni *client*.

Tutte le componenti di *ChirpStack* (*gateway bridge*, *network server* e *application server*) hanno dei topic di riferimento, ognuno con il propria struttura, per gestire l'invio e la ricezione di messaggi su queste code topiche. Per ogni componente ci sono un topic in ***lettura*** e uno in ***scrittura***.

Nell'ambito *MQTT*, il carattere “+” rappresenta un valore *jolly*, ovvero un valore in generale.

gateway bridge:

I topic in ***lettura*** per *gateway bridge*, ovvero quelli a cui si iscrive, hanno la struttura:

`gateway/{gatewayID}/command/{commandType}`

Dove a *gatewayID* si sostituisce l'id del *gateway* di cui interessa ricevere messaggi mentre *commandType* indica il tipo di messaggio¹.

I topic in ***scrittura*** per *gateway bridge*, ovvero quelli dove si iscrive, sono di due tipologie:

- `gateway/{gatewayID}/state/conn` → Topic usati per pubblicare lo stato di operatività di un *gateway*².
- `gateway/{gatewayID}/event/{eventType}` → Topic per pubblicare i messaggi ricevuti dai *watchdog*³.

Network Server

Topic di ***scrittura*** con struttura `gateway/+event/+`

Topic di ***lettura*** con struttura `gateway/+command/+`

I *topic* di questa componente sono speculari a quelli del *gateway bridge*. Infatti, i dati scritti da quest'ultimo sono letti dal *Network Server*.

Application Server

Dal lato dell'*Application Server*, il protocollo *MQTT* viene utilizzato come una delle possibilità di integrazione con applicazioni consumatrici. A tal fine vengono definite delle code sulle quali la componente pubblica dei messaggi relativi ai *device* di una certa applicazione. La comunicazione è possibile anche nell'altra direzione, ovvero dalle applicazioni consumatrici all'*Application Server*, tramite delle code a cui

¹ Tipologie di commandType: <https://www.chirpstack.io/gateway-bridge/payloads/commands/>

² Topic state: <https://www.chirpstack.io/gateway-bridge/payloads/states/>

³ Tipologie di eventType: <https://www.chirpstack.io/gateway-bridge/payloads/events/>

quest'ultimo si iscrive, ed è possibile schedulare dei messaggi in *downlink* verso di *devices*.

I *topic* di *scrittura* hanno la struttura:

`application/{applicationID}/device/{devEUI}/event/{eventType}`

dove:

- *applicationID* è il codice identificativo univoco dell'applicazione.
- *devEUI* è il codice identificativo del dispositivo di cui viene pubblicato il messaggio.
- *eventType* rappresenta il tipo del messaggio⁴.

I *topic* di *lettura* hanno la struttura:

`application/{applicationID}//device/{devEUI}//command/down`

I topic in lettura sono quelli utilizzati per schedulare messaggi di *downlink*⁵ verso i dispositivi. Infatti, i messaggi pubblicati su queste code vengono letti dall'*Application Server* che poi li trasmette al *Network Server* per la codifica e invio attraverso la rete *LoRaWan*.

Tutti i messaggi devono essere rispettare una determinata struttura. Questi messaggi possono essere rappresentati sia in formato *JSON* che in formato *Protobuf*.

Api di integrazione

Oltre all'integrazione *MQTT*, *ChirpStack* mette a disposizione numerose *API* di integrazione⁶, sia per la gestione delle sue componenti che per l'interazione con l'utente.

Come si può vedere anche da Figura 5, l'integrazione con *ChirpStack* da parte di sistemi esterni può essere fatta mediante API esposte sia tramite protocollo **gRPC** che **RESTful JSON**.

⁴ Tipologie di eventType per ChirpStack Application Server: <https://www.chirpstack.io/application-server/integrations/events/>

⁵ Integrazione MQTT: <https://www.chirpstack.io/application-server/integrations/mqtt/>

⁶ Api ChirpStack : <https://github.com/brocaar/chirpstack-api/tree/master/protobuf/as/external/api>

Iterazione 0

In questa prima iterazione l'attenzione viene posta sulla definizione dei requisiti utenti, per poi quindi stilare il documento delle specifiche software. La specifica viene fatta con la descrizione dei casi d'uso che vengono identificati in seguito all'analisi dei requisiti. Oltre al documento di specifica viene definita anche l'architettura (o topologia) dell'intero sistema.

Analisi dei requisiti

Di seguito vengono elencati i requisiti che si vogliono soddisfare.

R0 – Azioni di riconfigurazione per allungamento del ciclo di vita delle sentinelle:

Dal momento che i dispositivi *IoT*(le sentinelle) sono portatili, quindi alimentati da batteria, si vuole ottimizzare la durata della batteria intraprendendo delle azioni correttive sulla loro configurazione. Delle possibili azioni potrebbero intervenire sulla gestione della potenza e della frequenza di trasmissione.

Scenario	Goal	Fenomeno	Azione
S1	Massimizzare la durata della batteria dei <i>watchdog</i> .	La batteria dei <i>watchdog</i> scende al di sotto di determinate soglie. (50%, 30%, 15%, 10%)	<p>Al superamento di ogni soglia è necessario:</p> <ul style="list-style-type: none"> • Diminuire la frequenza di trasmissione dei dati da parte dei <i>watchdog</i>. • Diminuire la potenza di trasmissione dei <i>watchdog</i>.

Figura 6 Scenario di auto-adattamento per allungare il ciclo di vita di un *watchdog*

R1 – Azioni di auto-diagnosi manutentiva di risoluzione autonoma dei guasti:

Si vuole rendere possibile la rilevazione automatica di guasti sui *watchdog*, identificando la sentinella danneggiata con le sue caratteristiche tecniche e geografiche. A seguito di un guasto si possono intraprendere due diverse tipologie di azioni:

- Correzione: insieme di azioni, che il sistema compie autonomamente, volte a ripristinare il corretto funzionamento del dispositivo.
- Allerta: se le azioni correttive non sono efficaci, viene mandato un messaggio di allarme per richiedere un intervento manuale.

Oltre a rendere robusta la rete di sentinelle, si vuole tutelare il sistema da possibili malfunzionamenti dovuti a guasti degli *edge-node*. Questo può essere fatto

introducendo diversi nodi *edge* che comunicano tra loro e cooperano in modo che, a fronte di un guasto su uno di essi, intervenga un altro nodo *edge* funzionante per prendere in carico i compiti di quello guasto.

Scenario	Goal	Fenomeno	Azione
S3	Rendere il sistema in grado di gestire i guasti sui <i>watchdog</i> in maniera autonoma. <i>(self-recovery)</i>	<i>Watchdog</i> silente, cioè non trasmette dati per un certo intervallo di tempo. (Ad es. 60s)	L' <i>edge-node</i> tenta di ripristinare il corretto funzionamento del <i>watchdog</i> guasto tramite il suo riavvio.
S4	Allerta in caso di guasto ad un <i>watchdog</i> e identificazione esatta delle sue caratteristiche fisiche e geografiche.	Guasto ad un <i>watchdog</i> non recuperabile.	L' <i>edge-node</i> allerta un operatore per un intervento manuale sul nodo <i>watchdog</i> guasto indicando le sue caratteristiche.
S5	Rendere il sistema in grado di gestire guasti agli <i>edge-node</i> in maniera autonoma. <i>(self-recovery)</i>	Guasto di un <i>edge-node</i> : non riceve i dati inviati dai <i>watchdog</i> e, quindi, non può inoltrarli all' <i>application server</i> .	<ul style="list-style-type: none"> L'<i>application server</i> ridistribuisce il traffico di dati che arrivava all'<i>edge-node</i> guasto verso altri <i>edge-nodes</i>. L'<i>application server</i> notifica il personale del guasto avvenuto.
S6	Limitare la perdita di informazioni dovute a guasti sull' <i>application server</i> .	Malfunzionamenti dell' <i>application server</i> che, quindi, non è in grado di ricevere e/o elaborare i dati trasmessi dai <i>gateway</i> .	<ul style="list-style-type: none"> Gli <i>edge-node</i> devono mantenere i dati ricevuti dai <i>watchdog</i> in un <i>buffer</i> locale finché il server non riprende la sua operatività. Gli <i>edge-node</i> devono notificare il guasto ad un indirizzo di rete dedicato e ritenuto affidabile.

Figura 7 Scenari di auto-rilevazione dei guasti

L'avvenimento di guasti viene fatto secondo la tattica ***ping-echo***. In particolare, le condizioni di verificabilità dei vari scenari sono:

- S3: *ping* mandato da un *edge-node*, *echo* non mandato dal *watchdog*
- S5: *ping* mandato dal server e/o da un *edge-node*, *echo* non mandato dall'*edge-node* destinatario
- S6: *ping* mandato dagli *edge-node*, *echo* non mandato dall'*application server*

User-Stories

Le *user stories* sono suddivise in tre raggruppamenti, uno per attore della rete. Questa soluzione ci permette di identificare i diversi *use-case* che saranno essenziali nello *use case diagram UML*.

Lato edge device:

✓ Requisito R0:

- Come nodo *edge*, voglio essere in grado di conoscere il livello della batteria delle sentinelle in modo da sapere quando il livello di batteria scende sotto determinate soglie (M)
- Come nodo *edge*, voglio poter diminuire la frequenza di trasmissione delle sentinelle in modo da allungare la durata della loro batteria. (E)
- Come nodo *edge*, voglio poter diminuire la potenza del segnale di trasmissione delle sentinelle in modo da allungare la durata della loro batteria. (E)

✓ Requisito R1:

- Come nodo *edge*, voglio essere in grado di sapere quando una sentinella è silente in modo da poter rilevare sui guasti. (M)
- Come nodo *edge*, voglio essere in grado di riavviare una sentinella guasta in modo da ripristinare il suo funzionamento. (E)
- Come nodo *edge*, voglio essere in grado di mandare messaggi a del personale in modo da avvisare che una sentinella è guasta e comunicare i suoi dati. (E)
- Come nodo *edge*, voglio poter mandare un messaggio di *ping* ad altri nodi *edge* in modo da verificare che siano in funzione. (M)
- Come nodo *edge*, voglio poter subentrare al posto di un altro nodo *edge* in modo da svolgere i suoi compiti. (E)
- Come nodo *edge*, voglio poter mandare dei messaggi di *ping* all'*application server* in modo da avere informazioni sul suo funzionamento. (M)
- Come nodo *edge*, voglio poter analizzare le risposte ai messaggi di *ping* mandati all'*application server* in modo da rilevare se sta male funzionando. (A)
- Come nodo *edge*, voglio poter mantenere i dati che ricevo in un *buffer* locale in modo da evitare perdite di dati quando l'*application server* non li può ricevere. (E)
- Come nodo *edge*, voglio essere in grado di mandare messaggi ad un indirizzo di rete assegnato in modo da avvisare che l'*application server* è malfunzionante. (E)

Lato application server:

✓ Requisito R0:

- Come *application server* voglio essere in grado di valutare lo stato di carica delle sentinelle in modo da sapere quando la loro autonomia oltrepassa una certa soglia. (A)
- Come *application server* voglio essere in grado di determinare la frequenza e potenza del segnale di trasmissione per le sentinelle in modo da allungare il loro ciclo di vita. (P)
- Come *application server* voglio essere in grado di analizzare il livello di traffico in ingresso ad un nodo *edge* in modo da rilevare situazioni di congestione del nodo. (A)
- Come *application server* voglio essere in grado di determinare la frequenza di trasmissione delle sentinelle in modo da diminuire il flusso di dati in arrivo ad un nodo *edge* congestionato. (P)
- Come *application server* voglio essere in grado di analizzare il livello di traffico di un nodo *edge* in modo da rilevare situazioni di sottoutilizzo della rete. (A)
- Come *application server* voglio essere in grado di determinare la frequenza di trasmissione delle sentinelle in modo da aumentare il flusso di dati in arrivo ad un nodo *edge*. (P)

✓ Requisito R1:

- Come *application server*, voglio poter fare delle analisi sui dati (le risposte ai *ping*) delle sentinelle ricevuti dal nodo *edge* in modo da sapere quando una sentinella è guasta. (A)
- Come *application server*, voglio poter fare delle analisi sui dati (le risposte ai *ping*) delle sentinelle ricevuti dal nodo *edge* in modo da sapere quando un sensore di una sentinella è guasto (A)
- Come *application server*, voglio poter comunicare ad un nodo *edge* di riavviare una sentinella in modo da ripristinare il suo funzionamento. (P)
- Come *application server*, voglio poter decidere a quale nodo *edge* assegnare ad un gruppo di sentinelle in modo da subentrare ai compiti di un nodo *edge* malfunzionante o guasto. (P)
- Come *application server*, voglio poter inviare un messaggio di allerta in modo da notificare il malfunzionamento di un nodo *edge* e comunicare i suoi dati. (E)

Lato watchdog:

- Come sentinella, voglio poter generare dei dati fintizi in modo da simulare una rilevazione di dati da parte di sensori.
- Come sentinella, voglio poter simulare diversi stati di funzionamento in modo da simulare un dispositivo elettronico fisico.

Deployment Diagram

Lo scopo di questa fase è quello di realizzare un *deployment diagram*, ossia un diagramma che ha il compito di rappresentare la vista statica delle componenti della rete. In altre parole, si vogliono descrivere le parti *hardware* della rete e le relazioni tra esse. In un primo momento si è optato per un diagramma informale, che non segue alcuna regola di raffigurazione, ma che aiuta a inquadrare l'idea del progetto.

In un secondo momento si sono seguiti i principi del *Deployment Diagram UML* che, oltre a raffigurare la parte *hardware* del sistema, sottolinea anche il *software* che viene eseguito su una determinata componente.

Deployment Diagram Informale

Nel seguente diagramma sono rappresentati gli attori della rete : i *watchdog*, gli *edge device* e la *cloud virtual machine*. Inoltre, sono ritratte le aree in cui risiedono le parti hardware. L'area di acquisizione contiene sia *watchdog* che *edge device*: è noto che la comunicazione tra essi avviene tramite protocollo *LoRaWan* e che un *edge device* riceve messaggi da più *watchdog*. I messaggi scambiati tra la rete di acquisizione e la *cloud virtual machine* avvengono tramite il protocollo *MQTT* che viene espresso come un canale di comunicazione. Le *client application* sono tutte quelle applicazioni che possono interagire con le *server application*.

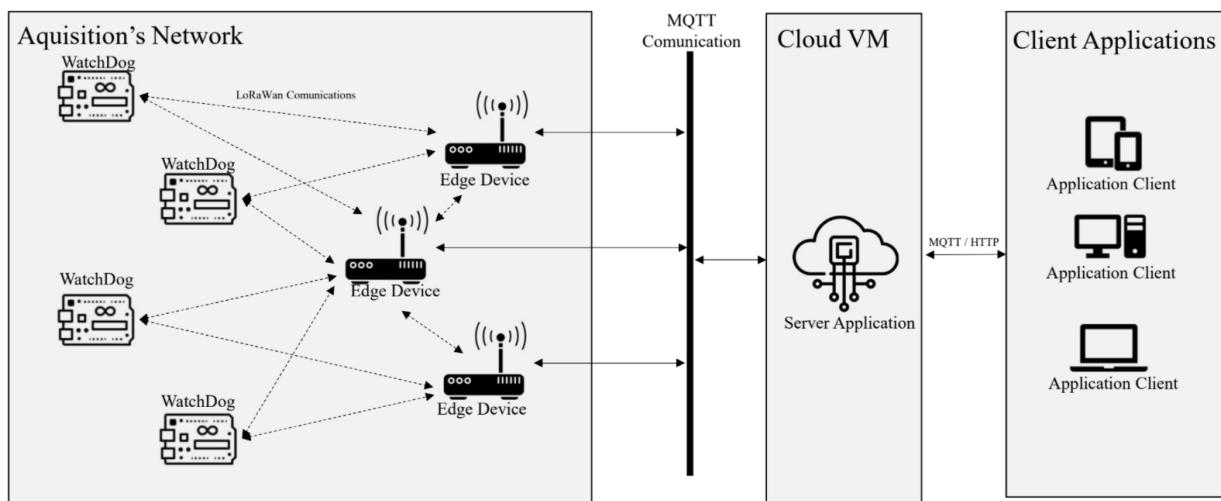


Figura 8 Deployment diagram informale

Deployment Diagram UML

Nel *deployment diagram UML* sono definiti tre *Subsystem* principali:

- *Watchdog*: questo sottosistema contiene due *device hardware*, un microcontrollore Arduino e i relativi sensori. Si può notare che un Arduino può avere uno o più sensori (per esempio di umidità, di temperatura, di CO₂, ...), mentre un sensore può interfacciarsi ad uno ed un solo microcontrollore. Su entrambi i dispositivi girano dei componenti *software embedded*, che verranno simulati successivamente con la tecnica del *digital twin*, discussa nel capitolo introduttivo.
- *Edge-device*: tale sottosistema comunica con il *watchdog* attraverso il protocollo *LoRaWan* e con il *cloud server VM* per mezzo del protocollo *MQTT*. Un *edge device* riceve messaggi provenienti da zero (caso estremo) a molti *watchdog* e li inoltra ad uno ed un solo *cloud server VM*. Il sistema operativo installato su questo dispositivo sarà verosimilmente il *ChirpStack gateway*, uno strumento messo a disposizione da *ChirpStack* per semplificare le operazioni di configurazione del *gateway*. Anche in questo caso verrà sfruttata la tecnica del *digital twin*.
- *Cloud Server*: questo sottosistema comunica con un numero di *edge device* che può variare da zero(caso estremo) a molti. È stato deciso di esplodere alcune componenti software indicate da *ChirpStack*, quali il *broker MQTT*, il *network server*, l'*application server* e l'interfaccia grafica.

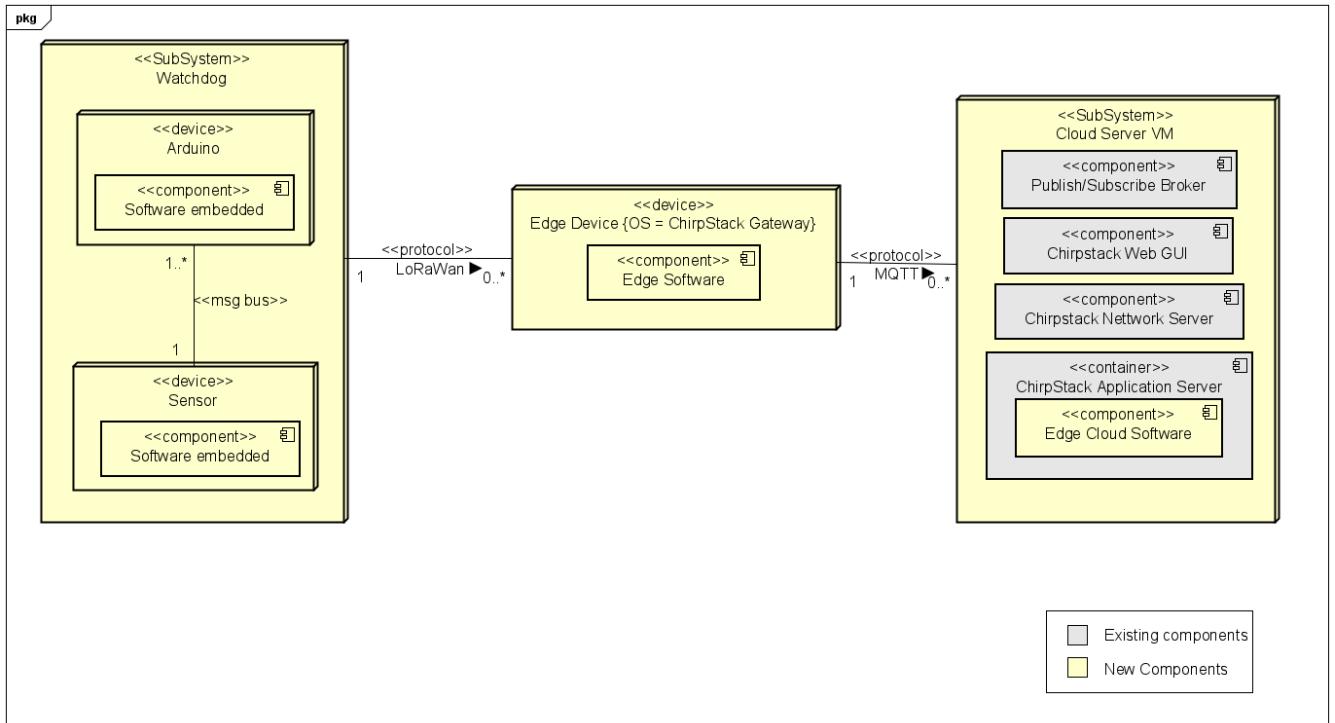


Figura 9 Deployment diagram in UML

Iterazione 1

Goal di iterazione

L'obiettivo principale di questa iterazione è quello di installare lo stack *ChirpStack* su macchina virtuale *Ubuntu* e configurare le tre principali componenti presenti. Questa operazione costituisce il *setup* iniziale di lavoro.

Oltre a questa operazione di *setup* si inizia anche la definizione ad alto livello dell'architettura a componenti del sistema, seguendo un approccio di tipo *top-down*.

Component diagram

Per la definizione delle componenti si è iniziato con una rappresentazione ad alto livello(**black-box**), riportata in Figura 10, del sistema mediante tre macro-componenti principali che comunicano tra loro mediante delle interfacce che verranno implementate nelle iterazioni successive.

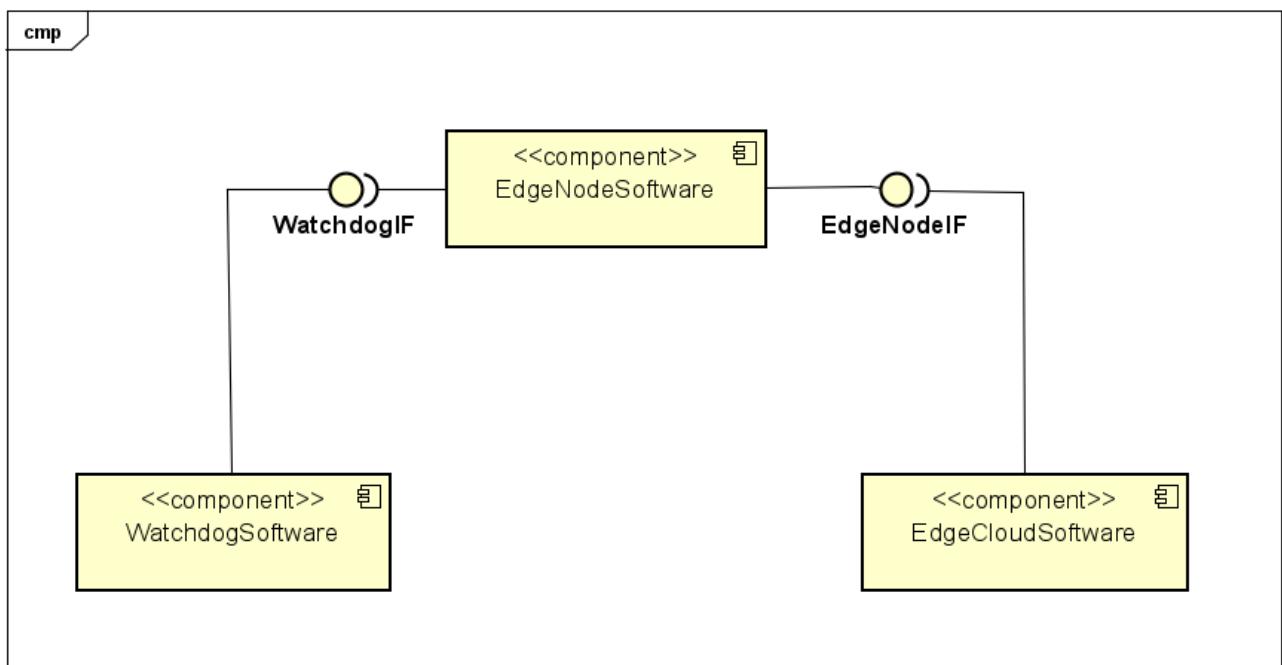


Figura 10 Component diagram iterazione 1

Class diagram

In questa iterazione iniziale, il diagramma delle classi(Figura 11) mostra le interfacce delle tre entità principali, ognuna con una lista di metodi che rappresentano le funzionalità fondamentali che queste dovranno implementare.

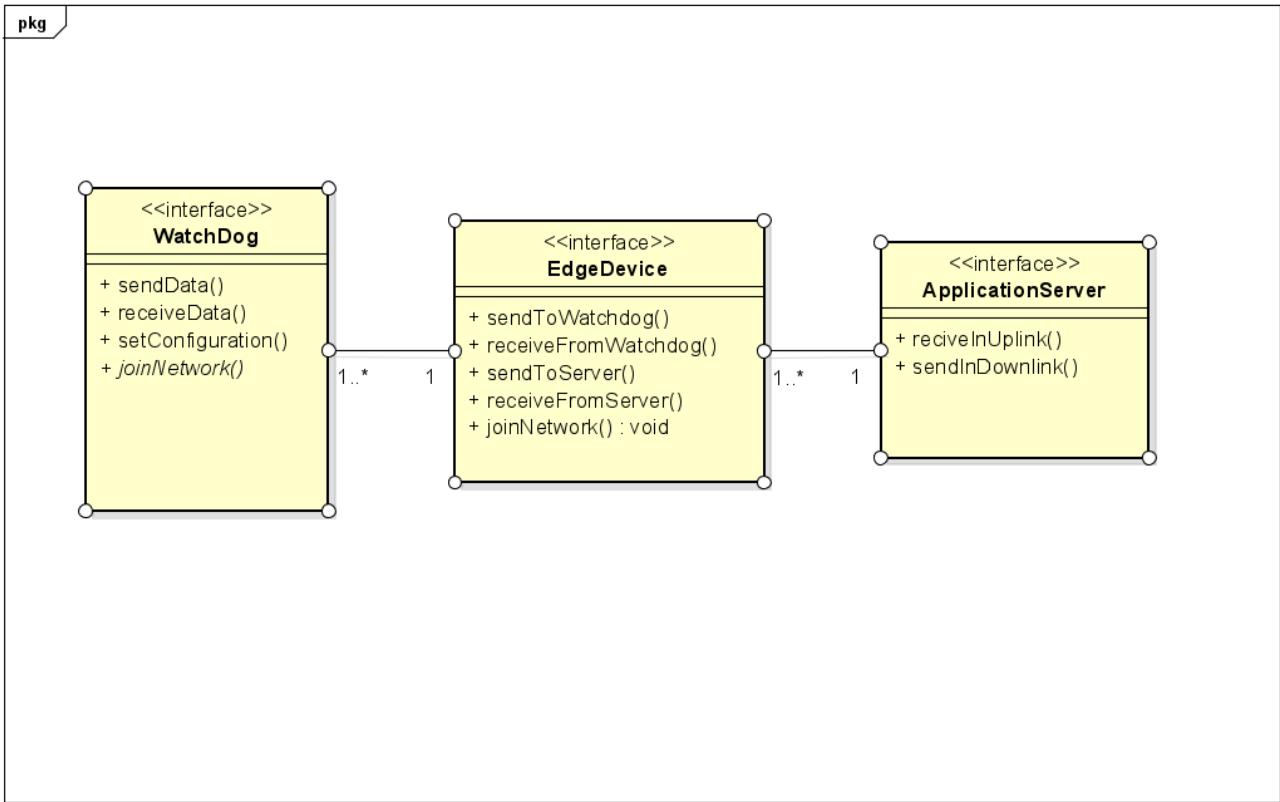


Figura 11 Class diagram iterazione 1

Test con simulatore

LWN-Simulator

Per capire il funzionamento di una rete costituita da *gateway* e *watchdog* e verificare come avviene lo scambio di messaggi tra i due tipi di nodo, è stato utilizzato il simulatore *open source LWN-Simulator*⁷.

LWN-Simulator permette di simulare i nodi una rete *LoRaWan*, la quale comprende i nodi *watchdog* e *edge-node*. Questo simulatore è dotato di un’interfaccia web che ne rende estremamente facile l’utilizzo. Inoltre, è possibile inserire *gateway* virtuali oppure reali. Il secondo caso è utile quando si ha già un’infrastruttura LoRaWan esistente e la si vuole monitorare. Ai fini delle verifiche di questa fase i *gateway* inseriti sono virtuali.

Il simulatore consiste di tre componenti principali:

- I *device*, configurabili in diverso modo
- Il *forwarder*, che riceve i pacchetti dai device e li inoltra ai *gateway*
- I *gateway*, che possono essere reali o virtuali

Dall’interfaccia grafica è possibile costruire una rete *LoRaWan* creando *devices*, *gateway* e posizionandoli sulla mappa. Se questi dispositivi vengono creati correttamente, quando si avvia il simulatore, si può osservare nella *console* l’invio dei pacchetti dai dispositivi al *gateway* per la trasmissione al *ChirpStackNetworkServer*.

Simulazione

Le prove sono state fatte con un *gateway* e tre *watchdogs*, ma *LWN-Simulator* permette di creare quanti dispositivi si desiderano. In Figura 14 viene riportata l’interfaccia grafica web cui è possibile avviare la simulazione. Una volta avviata, cliccando sul pulsante in alto a destra della schermata, si possono vedere messaggi di *log* nell’apposita console(Figura 13).

⁷ <https://github.com/UniCT-ARSLab/LWN-Simulator>

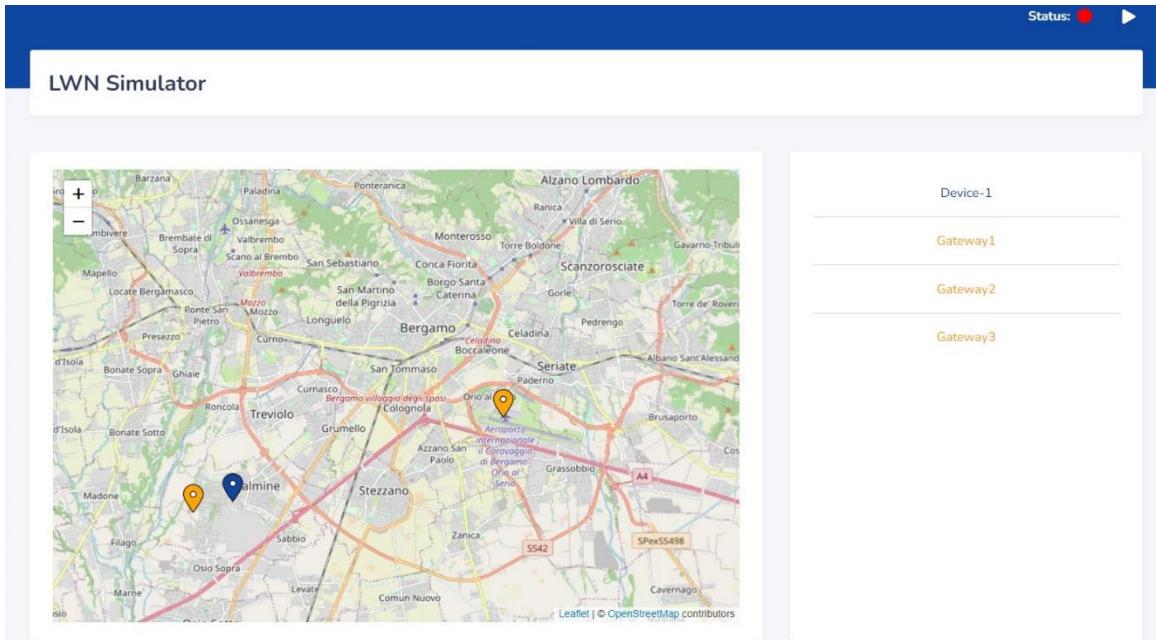


Figura 14 Interfaccia avviamento simulazione con LWN-Simulator

```

Console
[ Mar 1 20:54:50 ] GW[Gateway3]: PULL ACK received
[ Mar 1 20:54:51 ] DEV[Device-1] | Normal | (A): Switch channel from 6 to 5
[ Mar 1 20:54:51 ] DEV[Device-1] | Normal | (A): Uplink sent
[ Mar 1 20:54:51 ] DEV[Device-1] | Normal | (A): Open RXs
[ Mar 1 20:54:54 ] GW[Gateway1]: PUSH DATA send
[ Mar 1 20:54:55 ] GW[Gateway1]: PUSH ACK received
[ Mar 1 20:54:56 ] GW[Gateway2]: Turn OFF
[ Mar 1 20:54:56 ] GW[Gateway3]: Turn OFF
[ Mar 1 20:54:56 ] GW[Gateway1]: PULL RESP received
[ Mar 1 20:54:56 ] DEV[Device-1] | Normal | (A): Downlink Received
[ Mar 1 20:54:56 ] GW[Gateway1]: Turn OFF

```

Figura 13 Esecuzione LWN-Simulator

Dal lato *ChirpStack* si può invece verificare dalla *dashboard*(Figura 12) la corretta ricezione dei pacchetti.

UnconfirmedDataDown	867.7 MHz	SF12	BW125	FCnt: 33	DevAddr: 0131b425	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	867.7 MHz	SF12	BW125	FPort: 1	FCnt: 450	DevAddr: 0131b425
UnconfirmedDataDown	867.1 MHz	SF12	BW125	FCnt: 32	DevAddr: 0131b425	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	867.1 MHz	SF12	BW125	FPort: 1	FCnt: 449	DevAddr: 0131b425
UnconfirmedDataDown	868.1 MHz	SF12	BW125	FCnt: 31	DevAddr: 0131b425	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	868.1 MHz	SF12	BW125	FPort: 1	FCnt: 448	DevAddr: 0131b425
UnconfirmedDataDown	867.3 MHz	SF12	BW125	FCnt: 30	DevAddr: 0131b425	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	867.3 MHz	SF12	BW125	FPort: 1	FCnt: 447	DevAddr: 0131b425
UnconfirmedDataDown	868.1 MHz	SF12	BW125	FCnt: 29	DevAddr: 0131b425	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	868.1 MHz	SF12	BW125	FPort: 1	FCnt: 446	DevAddr: 0131b425
UnconfirmedDataDown	867.7 MHz	SF12	BW125	FCnt: 28	DevAddr: 0131b425	GW: 1f6aa45e9ed77a78

Figura 12 Gateway dashboard ChirpStack

Questa simulazione può essere considerata come una conferma del corretto *setup* dell'ambiente *ChirpStack*.

Iterazione 2

Goal di iterazione

In questa fase del progetto si inizi dettagliando la componente *EdgeNodeSoftware*. In particolare, si vuole implementare il funzionamento di un *edge-node* facendone una simulazione.

L’obiettivo è di sviluppare una componente in grado di simulare il comportamento di un *LoRaGateway*. In un contesto reale gli *edge-node* comunicano attraverso il protocollo *UDP* o sue varianti, tra cui molto noto il protocollo *Semtech UDP*, utilizzato anche nello *stack ChirpStack*. Tuttavia, dal momento che l’interesse di questo progetto è mettere in evidenza il flusso di dati gestito da *ChirpStack*, la componente *edge-node* si interfacerà direttamente con il *Network Server* di *Chirpstack*, implementando quindi anche le funzionalità di interfacciamento, offerte dalla componente *GatewayBridge* messa a disposizione da *Chirpstack*. Pertanto, la comunicazione avverrà direttamente tramite il protocollo *MQTT*, senza l’utilizzo di *Semtech UDP*.

Component diagram

La componente *EdgeNodeSoftware* deve poter comunicare sia con la componente *WatchdogSoftware* che con la componente *EdgeCloudSoftware*. Per fare ciò esso sfrutta due sottocomponenti:

- *WatchDog Communication* per ricevere dati da più *watchdog* mediante l’interfaccia *WatchDogIF*
- *MQTT Communication* per scambiare dati con l’*application server* appoggiandosi sull’interfaccia *BrokerIF* messa a disposizione da un *Broker MQTT* già esistente (non necessita implementazione)

In Figura 15 viene riportata di diagramma delle componenti risultato da questa prima fase di modellazione.

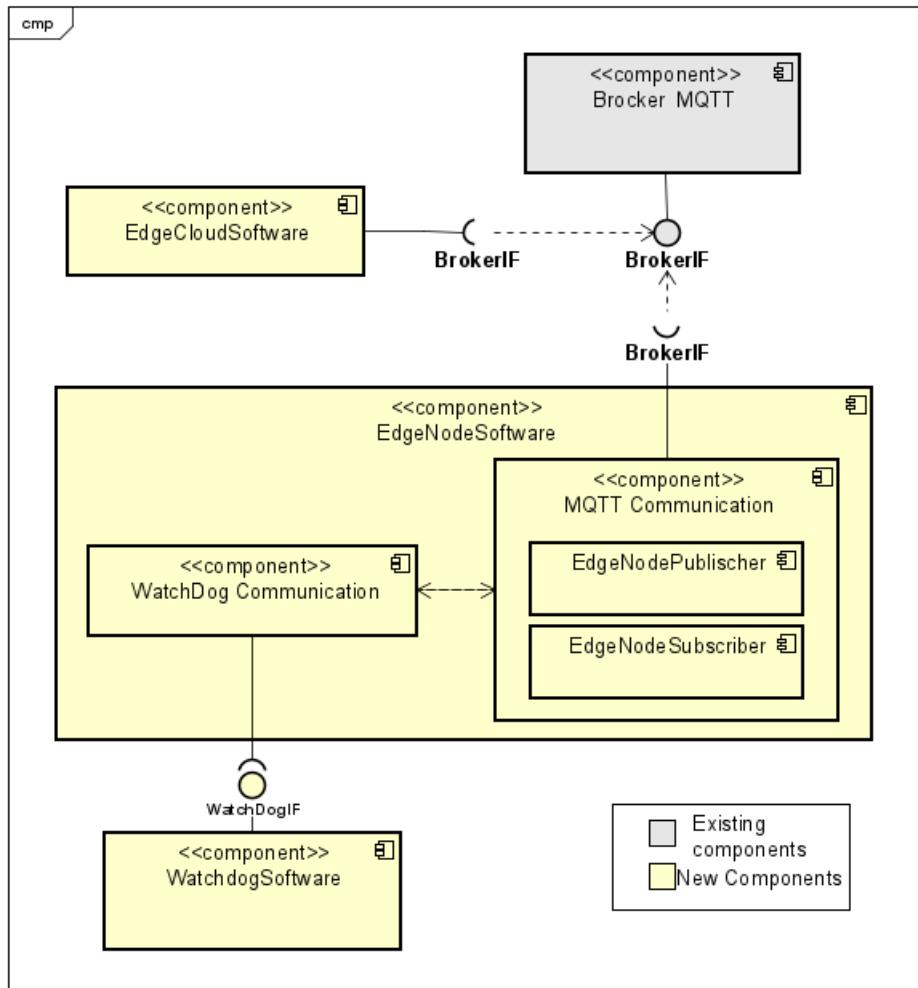


Figura 15 Component diagram iterazione 2

Class diagram

Rispetto al diagramma delle classi precedente è stata aggiunto un'interfaccia *broker* (già esistente) e dettagliando la componente *EdgeNodeSoftware* in:

- Un'interfaccia *EdgeDevice* nella quale sono presenti tutti i metodi esposti dall'*edge-node* relativi alla sua comunicazione con gli altri componenti all'interno della rete
- Una classe *EdgeNode* in cui vengono riportati campi e metodi necessari per il corretto funzionamento di un *edge-node* ed implementati i metodi dell'interfaccia *EdgeDevice*. Questo fa sì che un *edge-node* possa scambiare dati con l'esterno.

Il diagramma delle classi per la seguente iterazione è in Figura 16.

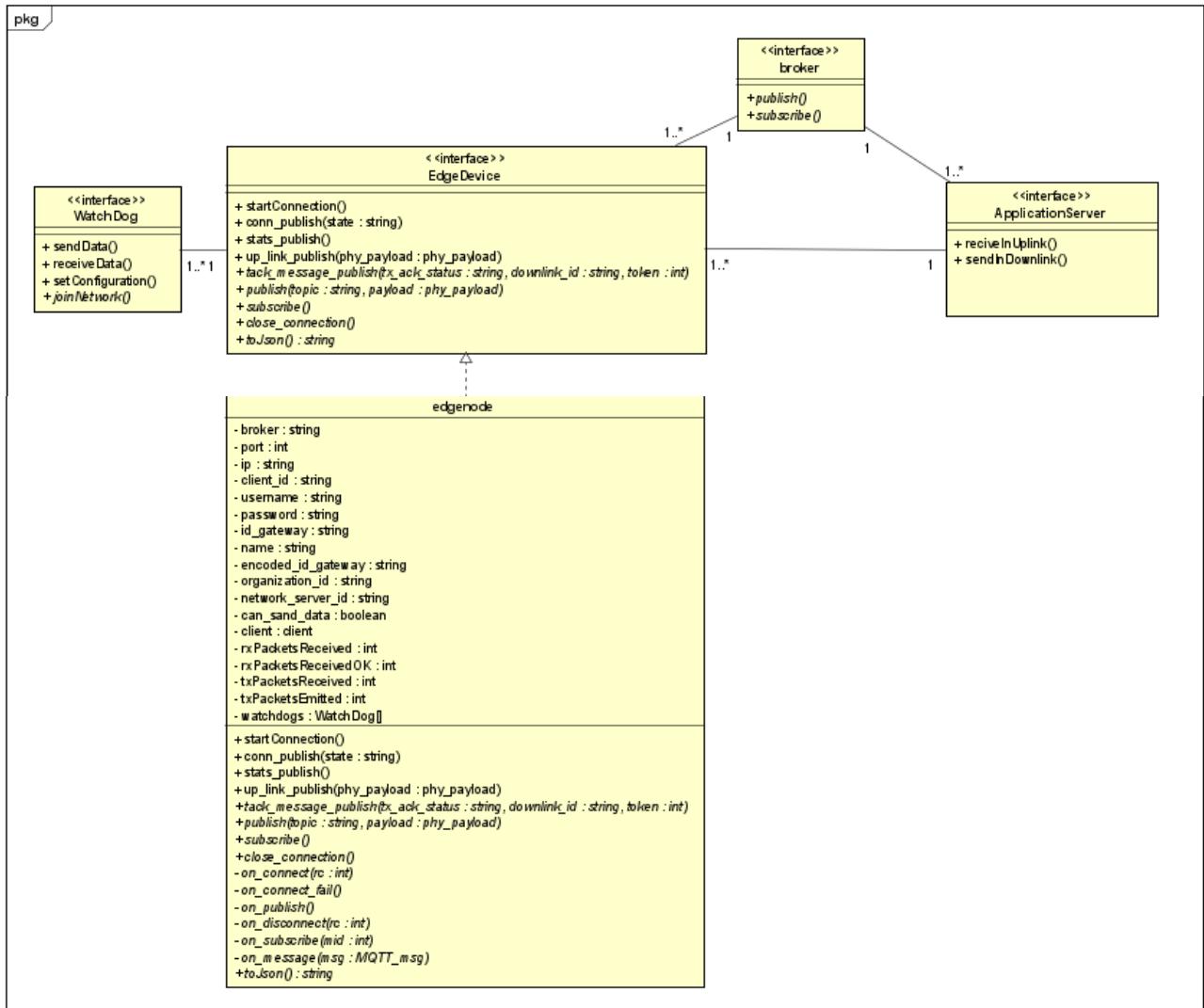


Figura 16 Class diagram iterazione 2

Test comunicazione

La verifica della corretta comunicazione viene fatta avviando la componente sviluppata e controllando che i pacchetti inviati siano ricevuti nella *dashboard* di *Chirpstack*.

Innanzitutto, il programma si presenta con una semplice interfaccia grafica, riportata in Figura 17, con cui si avviano le tre componenti principali. In questa fase l'unica componente attiva è quella dell'*edge-node*, indicata con il termine *gateway*.

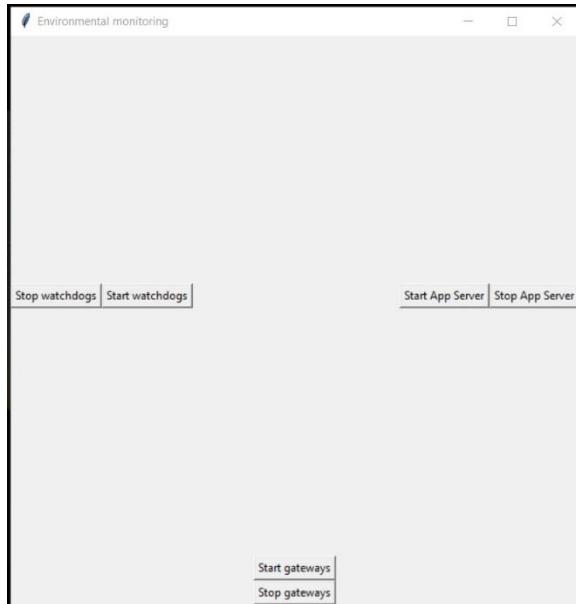


Figura 17 Interfaccia grafica principale

All'avvio della componente si ha la schermata mostrata in Figura 18, dove si vede che la comunicazione avviene correttamente e che i nodi *edge* si collegano correttamente al broker *MQTT*.

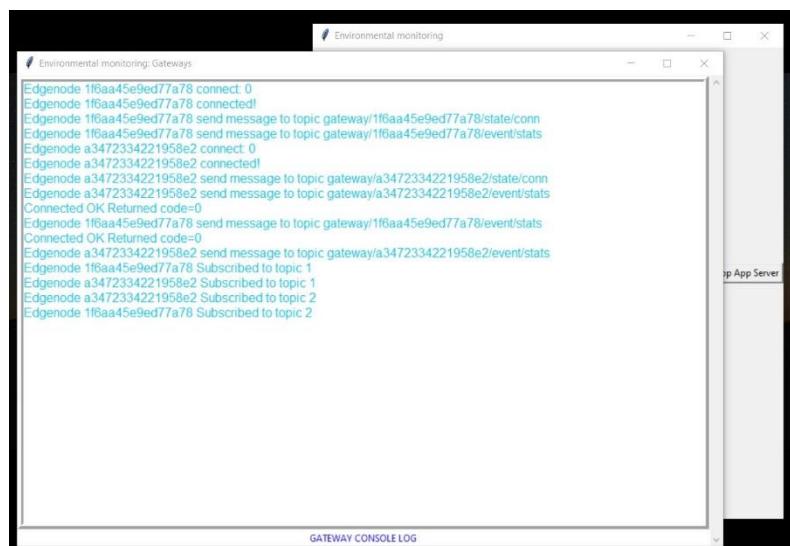


Figura 18 Interfaccia di log edge-node

Questo viene verificato guardando dall’interfaccia grafica del ChirpStack *ApplicationServer*(Figura 19), dove si vede la corretta connessione.

Last seen	Name	Gateway ID	Network server	Gateway activity (30d)
a minute ago	Gateway1	1f6aa45e9ed77a78	test-chirpstack-network-server	
a minute ago	Gateway2	a3472334221958e2	test-chirpstack-network-server	

Figura 19 Chirpstack gateway dashboard

La componente sviluppata è quindi in grado di connettersi al *brokerMQTT* e pubblicare i messaggi correttamente.

Iterazione 3

Goal di iterazione

In questa iterazione viene implementato il funzionamento del secondo nodo principale in una rete LoRaWan: il *watchdog*. Si va quindi a definire la componente *WatchdogSoftware*.

L’obiettivo è quello di simulare con una componente software il comportamento di un hardware fisico *hardware*, in particolare, di un *watchdog* che trasmette dati di acquisizioni da due sensori: uno di temperatura e uno di umidità. I dati trasmessi saranno naturalmente generazioni casuali dal momento che non è questo il focus del progetto.

Oltre alla generazione dei dati è necessario anche la trasmissioni di questi ai nodi *edge* simulati secondo il protocollo LoRaWan. In realtà, per i nostri fini, la cosa importante è che il formato dei dati rispetti le specifiche del protocollo: questo, quindi, rappresenta un altro obiettivo di tale fase.

La necessità di tale livello di dettaglio nella simulazione è dovuta all’interfacciamento con il *ChirpStack NetworkServer*, che si aspetta i dati secondo un determinato formato. Questa funzionalità è anche incapsulata nel *ChirpStack GatewayBridge*, ma, non facendone uso in questo progetto, viene implementata separatamente.

Il *watchdog* deve anche essere in grado di mandare i dati relativi al suo stato, in particolare del livello di batteria.

Component diagram

La componente *WatchdogSoftware* deve essere in grado di comunicare con l'*edge device* connesso tramite il protocollo *LoRaWan*: ciò è stato fatto tramite una sottocomponente *WatchdogLoraCommunication*, la quale scambia dati con l'*EdgeNodeSoftware* attraverso le interfacce *WatchDogIF* ed *EdgeDeviceIF*. A sua volta questa sottocomponente è stata suddivisa in:

- Una componente *Coder* che si occupa della codifica dei dati da inviare. Essa prende in pasto i messaggi in formato *JSON* e li converte in un formato conforme al protocollo *LoRaWan*. Oltre alla codifica, si occupa anche dell'operazione duale, cioè della decodifica, dei messaggi ricevuti.
- Una componente *Manager* per la gestione interna del componente.

All'interno di *WatchdogSoftware* vi è poi un'altra sottocomponente, chiamata *WatchdogManager*: si tratta di una componente che si occupa della simulazione del comportamento mediante due ulteriori sottocomponenti:

- *Sensor*, la quale ha il compito di campionare i dati ambientali. Nel caso di questo progetto i dati verranno simulati attraverso la componente *DataGenerator*. I campioni vengono generati secondo una distruzione normale, che in prima approssimazione possono rappresentare valori di temperatura e umidità.
- *Effector*, il quale si occupa dell'operazione di configurazione del *watchdog* mediante la componente *WatchdogConfigurator*. Tra i possibili parametri, sono stati previsti:
 - La frequenza di campionamento.
 - Il periodo di trasmissione.
 - La finestra di recezione.

In Figura 20 viene riportato il diagramma delle componenti finale di questa iterazione, che integra il digramma prodotto nella precedente iterazione aggiungendo le componenti descritte.

Guardando la componente *WatchdogSoftware* si può osservare l'applicazione dello stile architettonico **esagonale**. La logica operativa e quella di comunicazione vengono incapsulate in due componenti distinte e disaccoppiate. Infatti, è possibile usare le stesse funzionalità anche con differenti protocolli di comunicazione. Questo lascia le porte aperte a sviluppi futuri del software, con il quale si possono offrire le stesse funzionalità in diversi contesti.

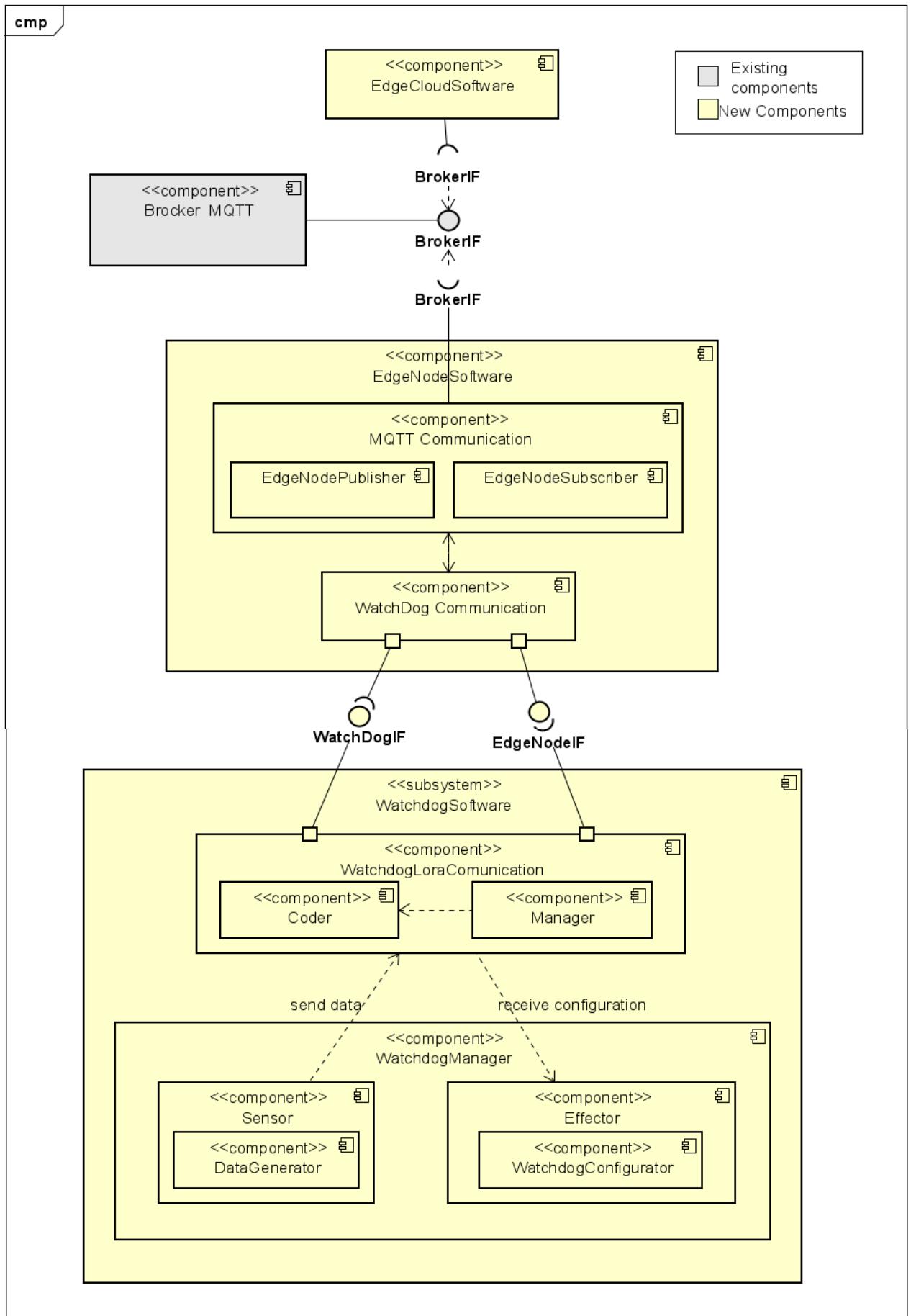


Figura 20 Component diagram iterazione 3

Class diagram

Come fatto per l'*EdgeNodeSoftware*, anche per l'implementazione della componente *WatchdogSoftware* è stata divisa in:

- Un'interfaccia *WatchDog* nella quale sono presenti tutti i metodi esposti dal *watchdog* relativi alla sua comunicazione con gli *edge-node* all'interno della rete
- Una classe *watchdog* in cui vengono implementati campi e metodi necessari per il corretto funzionamento di un *watchdog* ed i metodi dell'interfaccia *WatchDog*. Questo fa sì che un *watchdog* possa scambiare dati con l'*edge-device* al quale è connesso.

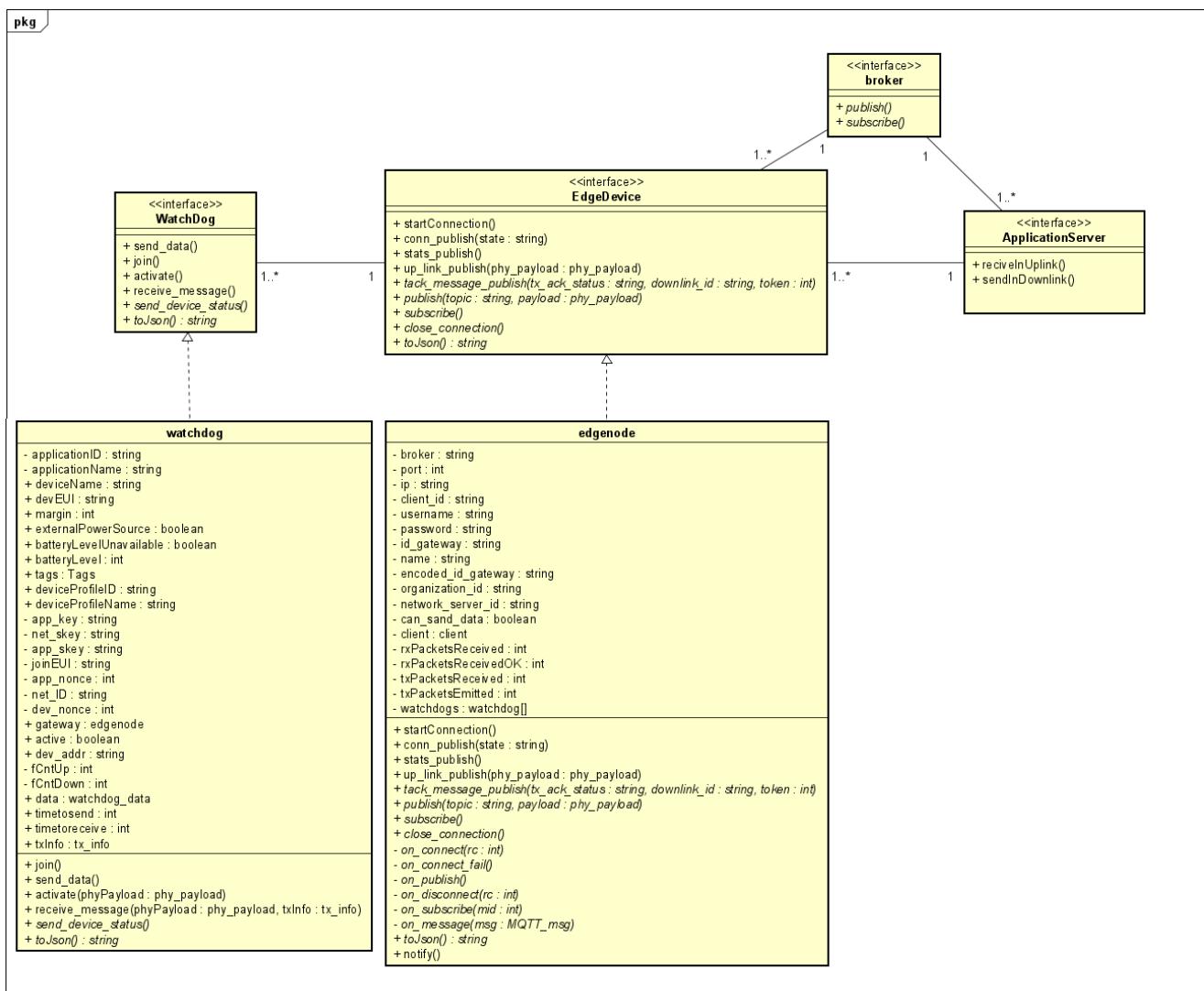


Figura 21 Class Diagram Iterazione 3

Il modulo *coder.py*

Il modulo *coder.py* (Figura 22) è la componente che ha il compito della codifica/decodifica dei pacchetti che attraversano il *watchdog*. In particolare, permette di decodificare i messaggi che arrivano in *downlink* dalla rete *LoRaWan*, così da poterli trattare e rappresentare in un formato standard come il *JSON*. Analogamente, la classe consente di codificare i pacchetti, contenenti i dati generati dal *watchdog*, e di poterli trasmettere attraverso rete *LoRaWan*.

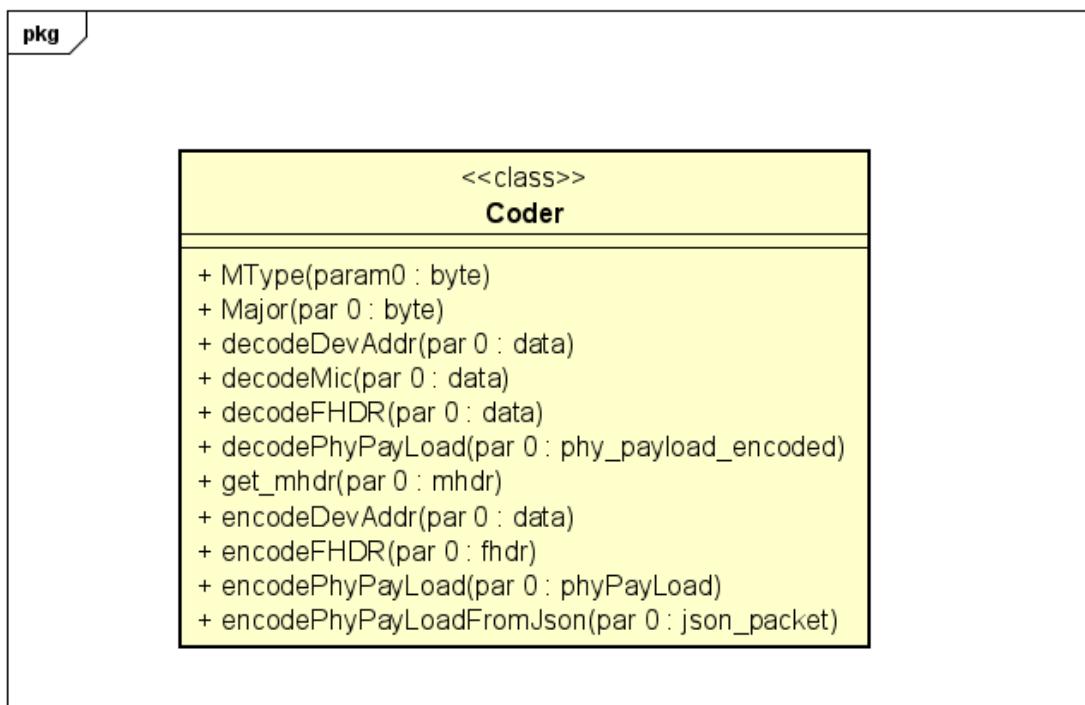


Figura 22 Class diagram modulo Coder

Le logiche di codifica fanno riferimento direttamente alle specifiche *LoRaWan*.⁸

In particolare, il *payload* viene codificato secondo il sistema *Base64*, che consente la traduzione di dati binari in stringhe di testo *ASCII*, rappresentando i dati sulla base di 64 caratteri *ASCII* diversi.

Il contenuto informativo di un pacchetto *LoRaWan* è contenuto nel campo *PHYPayload*. In questo contesto di simulazione si usa direttamente questo campo come *payload* totale, trascurando il resto dei campi che sono di interesse per il livello fisico, in quanto l'obiettivo del progetto è di mettere in evidenza l'interazione e flusso dei dati ad alto livello.

PHYPayload ha la struttura riportata nelle seguenti immagini:

⁸ https://lora-alliance.org/wp-content/uploads/2020/11/2015_-_lorawan_specification_1r0_611_1.pdf

PHYPayload	(MHDR MACPayload MIC)
MACPayload	(FHDR FPort FRMPayload)
FHDR	(DevAddr FCtrl FCnt FOpts)

Quindi, una volta ottenuto il pacchetto LoRaWan in *byte*, dal formato *Base64*, la classe *coder.py* implementa i metodi per decodificare (nel caso di un messaggio in *downlink*) o codificare (nel caso di un messaggio in *uplink*) i byte nei seguenti campi:

- **MHDR (MAC header)**: è composto da *1 byte* e contiene l'informazione sul tipo di messaggio trasmesso e la versione del protocollo.
- **MIC (Message integrity code)**: è di *4 byte* e serve per verificare l'integrità e la sicurezza del pacchetto. Per calcolarlo servono anche le chiavi con cui vengono cifrati i messaggi.
- **MacPayload** : è il campo che contiene il *payload* con il contenuto informativo, che a sua volta suddiviso in:
 - **FPort (Port field)**: è un campo opzionale di *1 byte*, di valore non negativo: Se posto a 0 indica che *FRMPayload* contiene dei comandi da eseguire, altrimenti *FRMPayload* contiene dei dati.
 - **FRMPayload**: anch'esso un campo opzionale che contiene dei comandi definiti a livello *MAC*, i quali devono essere implementati dai nodi della rete, utili per la trasmissione dei messaggi (risposte *ACK*, qualità di ricezione del segnale, status del dispositivo, settare gli slot di comunicazione...). La lunghezza di questo campo può essere variabile in funzione della lunghezza del messaggio.
 - **FHDR (Frame header)** è a sua volta composto da:
 1. **DevAddr**: Campo di *4 bytes* che contiene l'indirizzo del *device* a cui è destinato il messaggio.
 2. **FCtrl**: un *frame* composto da *8 bits* di controllo.
 3. **FCnt**: Campo di *2 bytes*, che è un contatore di messaggi. In particolare conta il numero di messaggi in *Uplink* e in *Downlink*.
 4. **FOpts** : contiene dei comandi MAC opzionali per il trasporto. La sua dimensione è contenuta dei primi *4 bits* del campo **FCtrl**, che può quindi varia tra 0 e 15.

Per le operazioni di codifica/decodifica dal formato base64 si utilizza la libreria `base64` offerta dal linguaggio Python. Di seguito vengono riassunte graficamente le operazioni che vengono svolte per passare da una rappresentazione dei dati all'altra(Figura 23).

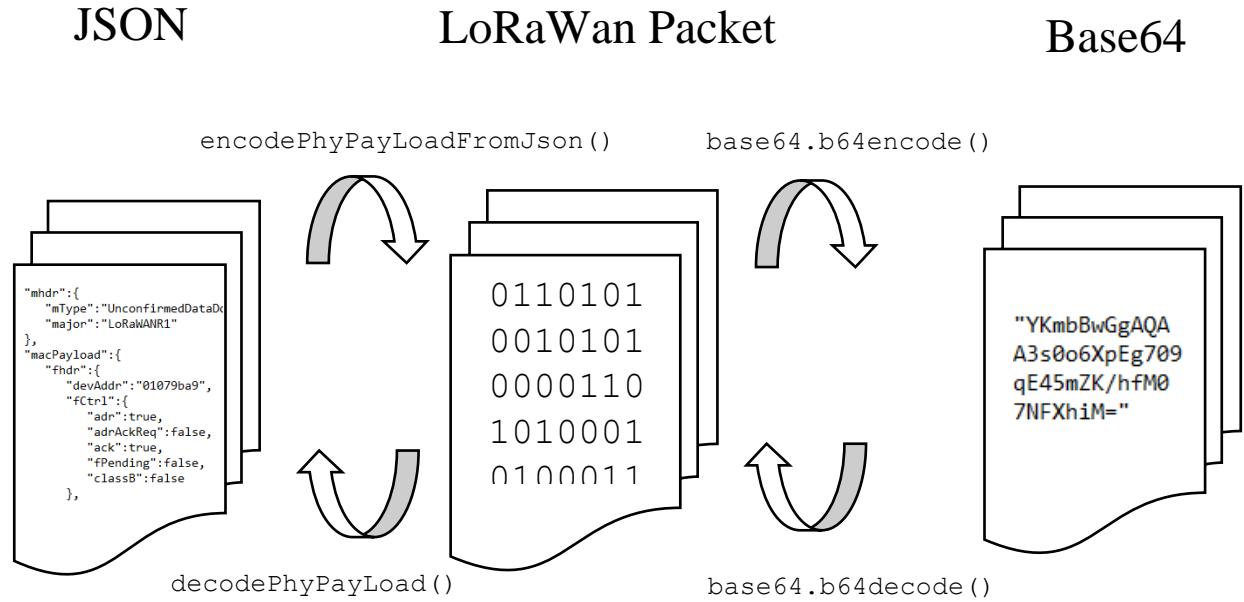


Figura 23 Rappresentazione dei messaggi

State machine watchdog

Per descrivere il funzionamento di un *watchdog* è stata realizzata una macchina a stati, riportata in Figura 24.

Lo stato iniziale è quello di *Turned on*. Non appena il *watchdog* viene acceso e posizionato, entra nello stato di *Idle* in cui aspetta che i suoi sensori raccolgano i dati dall'ambiente (temperatura, umidità, ...).

Non appena i dati sono pronti, il *watchdog* entra nello stato di *Stream*: in questo stato la sentinella trasmette i dati, raccolti dai sensori, ad un *nodo-edge*. Terminata la comunicazione tra i due nodi, il *watchdog* ritorna nello stato di *Idle*, in attesa che i suoi sensori raccolgano nuovi dati.

Quello descritto è il comportamento corretto della sentinella. Essa, però, può essere soggetta a guasti di varia natura: quando viene rilevato un guasto il *watchdog* entra nello stato di *Fault* e non è detto che vi possa uscire:

- Se è possibile intraprendere delle azioni correttive determinate, il *watchdog* viene riparato e ritorna nello stato di *Idle*, in attesa che i suoi sensori acquisiscano dati da mandare al *gateway*;
- Altrimenti, se non si riescono a determinare delle azioni che possano riparare il *watchdog*, esso rimane nello stato di *Fault* e si rende necessario un intervento manuale per farlo ritornare in funzione correttamente.

Quando la batteria del *watchdog* si scarica o quando viene semplicemente spento, si entra nello stato finale *Turned off*. In caso di esaurimento della batteria è necessario l'intervento di un operatore che provvederà alla sua sostituzione.

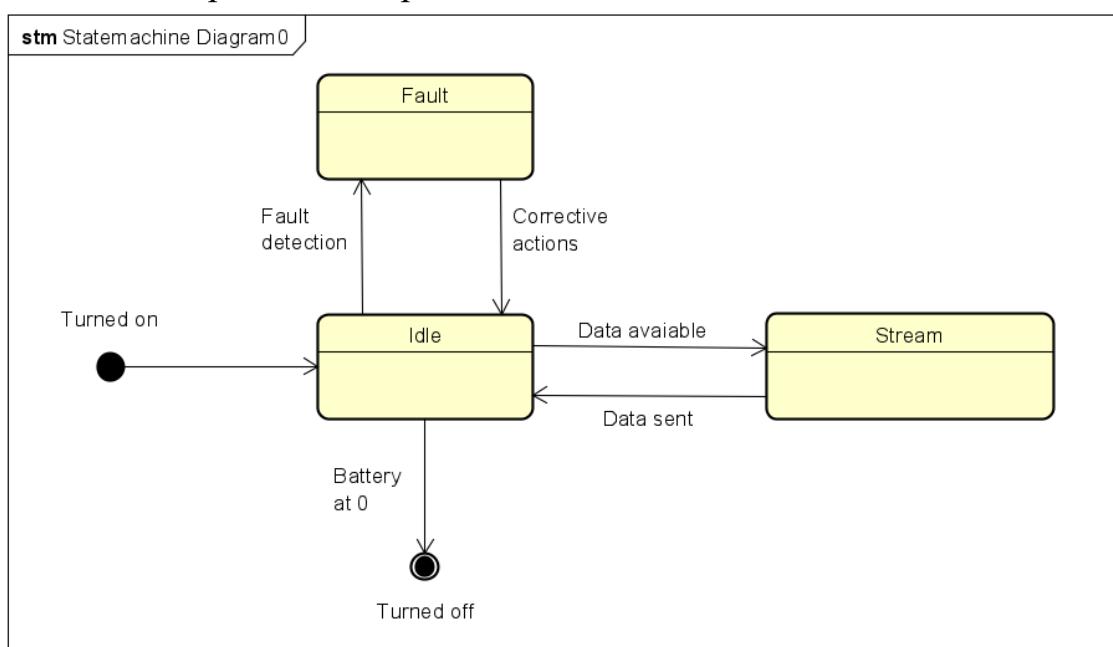


Figura 24 Macchina a stati finiti del funzionamento di un *watchdog*

Analisi dinamica

La fase di test si suddivide in due parti:

- Test di unità sulle componenti *coder* (discussa precedentemente) e *phy_payload_util*, la quale offre dei servizi di supporto al modulo *coder.py*. Questi moduli hanno un ruolo cruciale nel progetto in quanto svolgono operazioni di codifica e decodifica delle informazioni
- Test di integrazione globale per verificare che i messaggi generati dal *watchdog* trasmessi attraverso un *edge-node* siano recapitati al *ChirpStack ApplicationServer*

Unit test

Il test di unità viene fatto con l'ausilio del *tool Unittest*, integrato nell'*IDE PyCharm*. Questo *tool* permette di definire i casi di test e verificare che il risultato ottenuto sia coerente con quello atteso. I casi definiti possono essere poi essere rieseguiti automaticamente a seguito di modifiche del codice, riducendo così lo sforzo per test di regressione e favorendo la manutenibilità del codice.

La prima classe di test comprende i metodi:

- *encodePhyPayloadFromJson(jsonPayload)* che opera l'operazione di codifica.
- *decodePhyPayLoad(payload)* che decodifica il payload ricevuto.

Per ognuno di essi sono stati definiti quattro casi di test con differenti dati di input, ottenendo esito positivo e una totale copertura sia delle istruzioni che dei rami di controllo.

Viene realizzata anche una seconda classe di test per verificare il funzionamento delle funzionalità di cifratura. Infatti, la comunicazione viene resa sicura tramite una tecnica di crittografia simmetrica. In particolare, viene utilizzato l'algoritmo **AES**(*Advanced Encryption Standard*) a *128 bit*, che risulta essere una tecnica molto sicura in quanto è dotata di un basso *overhead* di esecuzione. La procedura di cifratura implementata fa riferimento direttamente alle specifiche *LoRaWan*.

I metodi verificati sono:

- *encryptFrmPayload()*, che opera una duplice funzione: cripta e decripta il contenuto del campo *frmPayload*, usando con le stesse operazioni.
- *encryptMacPayload()*, il quale svolge anch'esso le due funzioni citate, ma sul campo *macPayload*.
- *computeJoinRequestMic()*, che si occupa di calcolare il campo *MIC* di un messaggio di tipo **JoinRequest**.

- *decodeDataPayloadToMacCommands()*, il quale decodifica i dati ricevuti nel campo *frmPayload* per ottenere i comandi definiti dal protocollo LoRaWan.
- *computeJoinAcceptMic()*, che si occupa di calcolare il campo *MIC* di un messaggio di tipo **JoinAccept**.
- *computeJoinDataMic()*, che ha lo scopo di calcolare il campo *MIC* dei messaggi che contengono dati.

Per ogni metodo sono stati definiti almeno tre casi di test ottenendo una copertura esaustiva.

Integration test

Il test di integrazione è stato fatto semplicemente avviando la componente dall’interfaccia grafica del programma e, in seguito, verificando dall’interfaccia grafica di *ChirpStack* la corretta ricezione dei messaggi. In Figura 25 si può vedere come la connessione(processo di **Join**) e l’invio dei messaggi vada a buon fine.

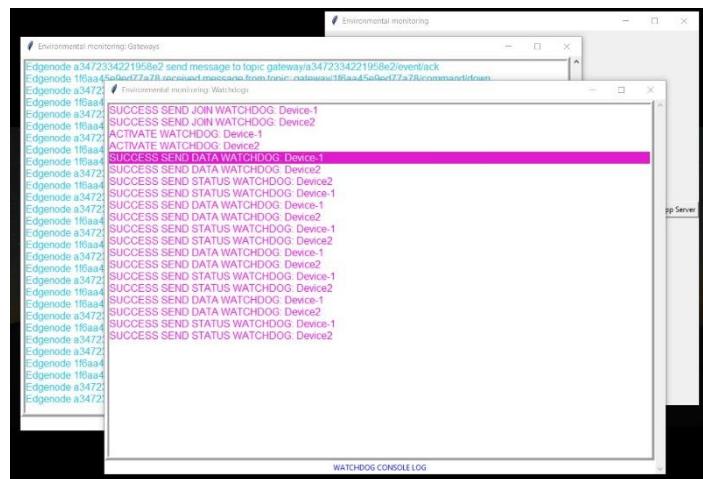


Figura 25 Interfaccia grafica watchdog

Il corretto funzionamento può essere riscontrato osservando ancora una volta i pacchetti che arrivano nella *dashboard*(Figura 26) dei dispositivi:

UnconfirmedDataDown	868.1 MHz	SF7	BW125	FCnt: 3	DevAddr: 0158efcd	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	868.1 MHz	SF7	BW125	FPort: 0	FCnt: 4	DevAddr: 0158efcd
UnconfirmedDataDown	868.1 MHz	SF7	BW125	FCnt: 2	DevAddr: 0158efcd	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	868.1 MHz	SF7	BW125	FPort: 1	FCnt: 3	DevAddr: 0158efcd
UnconfirmedDataDown	868.1 MHz	SF7	BW125	FCnt: 1	DevAddr: 0158efcd	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	868.1 MHz	SF7	BW125	FPort: 0	FCnt: 2	DevAddr: 0158efcd
UnconfirmedDataDown	868.1 MHz	SF7	BW125	FCnt: 0	DevAddr: 0158efcd	GW: 1f6aa45e9ed77a78
UnconfirmedDataUp	868.1 MHz	SF7	BW125	FPort: 1	FCnt: 1	DevAddr: 0158efcd
JoinAccept	868.1 MHz	SF7	BW125			GW: 1f6aa45e9ed77a78
JoinRequest	868.1 MHz	SF7	BW125			DevEUI: 0ac14aad3e6391a1

Figura 26 Dashboard di ChirpStack per i device

Si può quindi concludere che l’obiettivo è stato raggiunto. Con questa componente si ha a disposizione il necessario per mettere in evidenza le logiche e i flussi dati di interesse per questo progetto. In particolare, si possono sfruttare le funzionalità base dello stack *ChirpStack* per ottenere un sistema *self-adaptive*, aggiungendo nuovi *layer*.

Iterazione 4

Goal di iterazione

In questa fase del progetto viene implementato il primo requisito, denotato con R0, che consiste nella configurazione automatica dei *watchdog*, con obiettivo l'allungamento del loro ciclo di vita.

A tal fine viene dettagliato il progetto dell'ultima componente rimasta, l'*EdgeCloudSoftware*, che avrà il ruolo di encapsulare le logiche di controllo e di adattamento dei nodi della rete. In particolare, viene utilizzato il modello di controllo detto *MAPE-K loop*, che trova forte applicazione nelle architetture di sistemi autonomi e auto adattativi.

Component diagram

Il ciclo *MAPE-K* introdotto viene distribuito su due componenti separate.

Come criterio di progetto le funzioni **M** (*Monitoring*) ed **E** (*Execute*) vengono assegnate al nodo *edge* proprio perché quest'ultimo, solitamente, ha una bassa potenza di calcolo, e, talvolta, anche di autonomia di batteria. Le due funzione **ME** richiedono una carico computazione solitamente basso. Inoltre, essendo che le due funzioni richiedono l'interfacciamento diretto con i nodi *watchdog*, assegnarle ai nodi *edge* risulta essere una scelta più efficace anche dal punto di vista del risultato, riducendo l'*overhead* dell'*EdgeCloudSoftware*.

Le funzioni **A** (*Analyze*) e **P** (*Planning*) vengono invece assegnate all'*EdgeCloudSoftware*, dal momento che risultano essere più pesanti dal punto di vista computazionale.

Questa architettura segue una sorta di approccio *master-slave* distribuito, dove *EdgeCloudSoftware* opera da *master* interpellando i nodi *edge* che sono gli *slave*. L'idea di questa linea di progetto deriva dal desiderio di ottimizzare l'utilizzo di risorse ed avere un buon livello di *dependability* del sistema.

Di seguito vengono descritte le componenti riportate nel diagramma delle componenti riportato in Figura 27.

Rispetto al *component diagram* dell'iterazione precedente sono stati dettagliati due componenti:

- L'*EdgeNodeSoftware*, nel quale è stata aggiunta la componente *MAPE_BatteryAdaptationEdgeNode* che ha sua volta è dettagliata in due sottocomponenti:
 - **BatteryMonitor** → Si occupa di monitorare la batteria dei *watchdog* e trasmettere i dati all'*EdgeCloudSoftware*, svolgendo quindi la funzione **M** del modello *MAPE-K*.
 - **BatteryAdaptationExecute** → Si occupa di configurare i *watchdog* in base alle decisioni di *planning* dell'*EdgeCloudSoftware*, realizzando la funzione **E**.

Inoltre, avendo distinto le funzioni di *publish* e di *subscribe* nelle interfacce esposte dal *brokerMQTT*, si è dettagliato la componente di *MQTTCommunication* con le due sottocomponenti *EdgeNodePublisher* ed *EdgeNodeSubscriber*, che per la comunicazione richiedono rispettivamente l'interfaccia *Publish* e l'interfaccia *Subscribe*.

- L'*EdgeCloudSoftware* è composto da due sottocomponenti:
 - *AppServerMQTTCommunication*, che serve per la comunicazione tra *application server* e *broker MQTT*. Come fatto per l'*EdgeNodeSoftware* si sono distinte due sottocomponenti *AppServerPublisher* ed *AppServerSubscriber*, che richiedono rispettivamente l'interfaccia *Publish* e l'interfaccia *Subscribe*.
 - *MAPE_BatteryAdaptationAppServer*, invece, svolge le funzioni di *Analyze* e di *Plan* del *MAPE loop*, incapsulate nelle seguenti componenti:
 - ~ **AnalyzeBatteryAdaptation** → Si occupa di analizzare i dati sullo stato della batteria ricevuti dai nodi *edge*. Svolge la funzione di **A**.
 - ~ **PlanBatteryAdaptation** → Pianifica le azioni di controllo per raggiungere l'obiettivo desiderato. Rappresenta quindi la funzione **M**.

Anche in questo caso viene applicato lo stile esagonale per la modellazione delle nuove componenti. *EdgeCloudSoftware* ha un componente di comunicazione indipendente dalle logiche implementate con il ciclo *MAPE*, rendendo possibile esporre gli stessi servizi attraverso altri connettori con uno sforzo minimo.

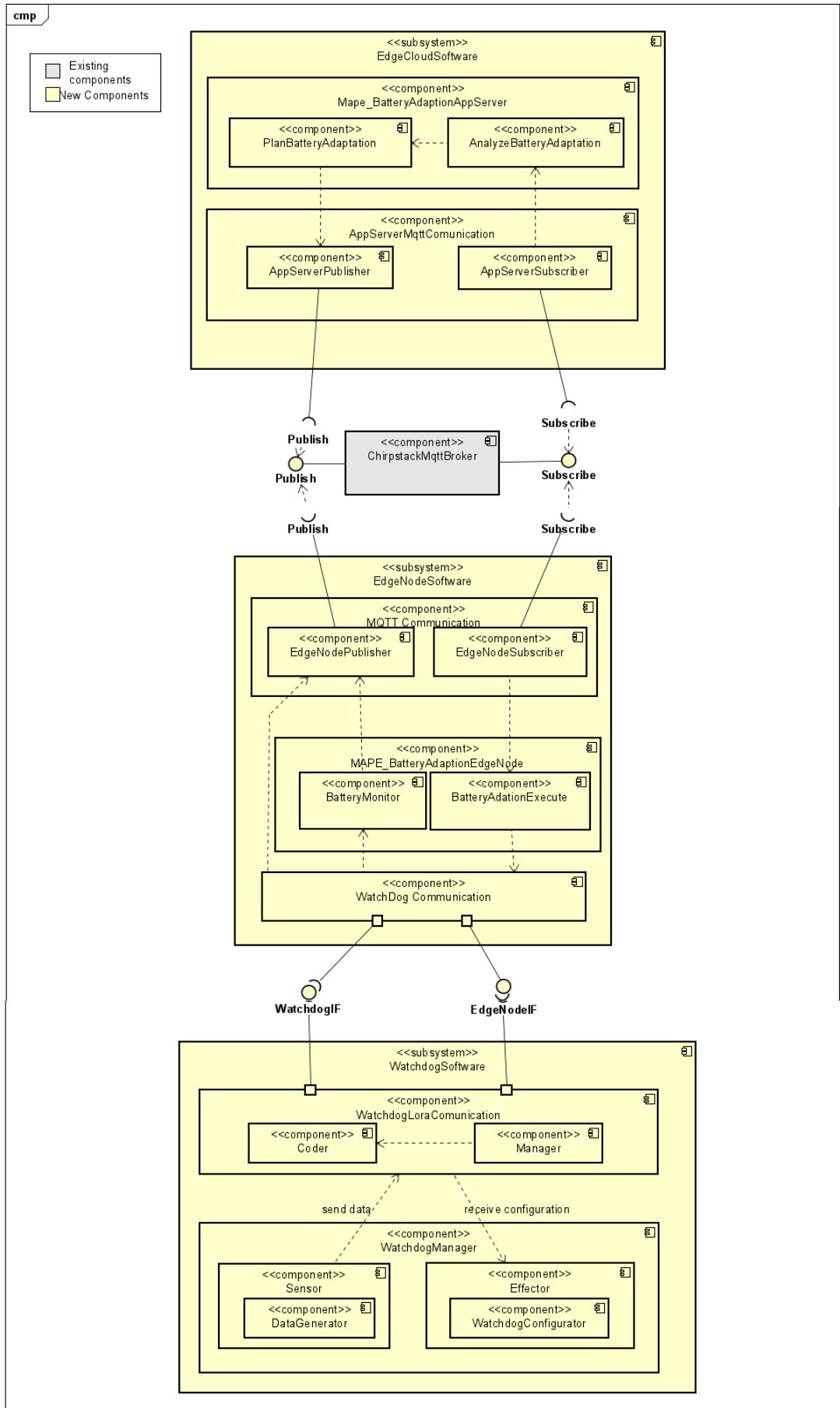


Figura 27 Component diagram iterazione 4

Class diagram

Come fatto per l'*edge-node* ed il *watchdog*, l’interfaccia *ApplicationServer* presente nei diagrammi delle iterazioni precedenti è stata esplosa in:

- Un’interfaccia *ApplicationServer* nella quale sono esposti due metodi che permettono all’*application server* di inviare e di ricevere dati
 - Una classe *AppServer* in cui vengono implementati campi e metodi necessari per il corretto funzionamento dell’*application server* ed i metodi dell’interfaccia *AppServer*. In questo modo l’*application server* può mandare dati a tutti i componenti della rete passando attraverso il *broker MOTT*

Inoltre, è stato aggiunto il metodo `configure()` alla classe `watchdog`, fondamentale per poter configurare tale dispositivo dopo che l'*application server* ha elaborato i dati che gli sono arrivati dallo stesso. La Figura 28 contiene il diagramma delle classi prodotto.

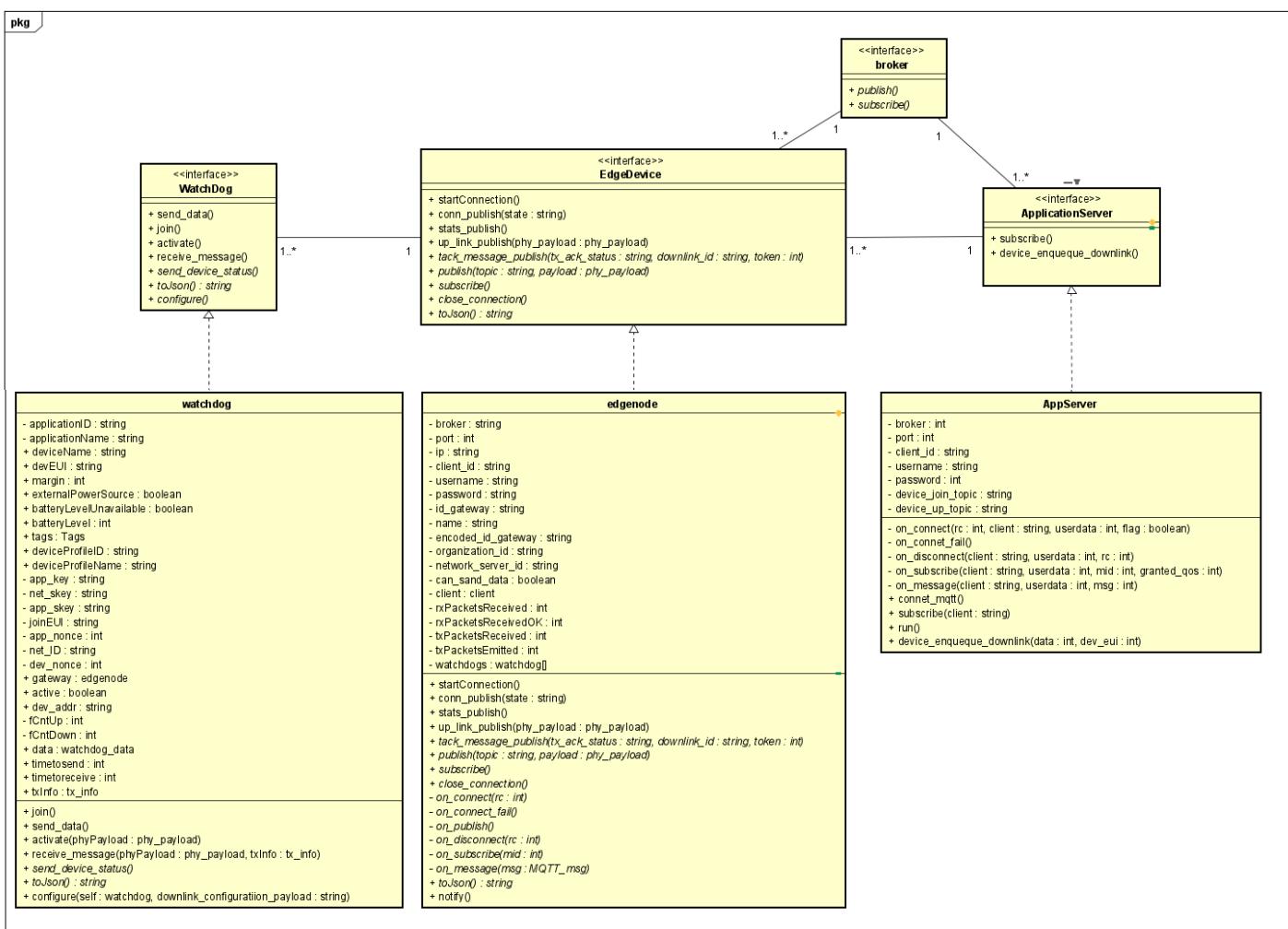


Figura 28 Class diagram iterazione 4

Analisi e verifica del funzionamento

La verifica del funzionamento viene fatta realizzando due scenari: in uno la componente di controllo viene disabilitata e nell'altro, invece, quest'ultima viene messa in funzione.

Assenza di controllo

In assenza di controllo i *watchdog* operano senza cambiare mai la loro configurazione. A questo proposito viene simulata la situazione di scarica della batteria mediante funzione lineare che diminuisce nel tempo secondo un indice di consumo definito da:

$$\text{indice di consumo} = 1 + \frac{2000}{\text{timetosend}}$$

dove *timetosend* è il periodo di trasmissione del *watchdog*. Questo indice tiene conto di una componente costante data dal termine “1”, che modella, in prima approssimazione, il consumo in assenza di operatività, dovuto appunto al funzionamento minimale del dispositivo. Il termine che dipende da *timetosend* modella, invece, il consumo dovuto all’invio di messaggi, che, appunto, è inversamente proporzionale al periodo di trasmissione. La costante “2000” viene usata solo a scopo di normalizzazione. Questo indice non rispecchia alcun parametro reale, ma serve solo per fare una prima analisi sull’efficacia del sistema di controllo.

Eseguendo la componente *watchdog* si ottiene un risultato finale (Figura 29), dove l’autonomia, con un periodo di trasmissione *timetosend* = 5000ms, è di circa 2 minuti.

```
SUCCESS SEND STATUS WATCHDOG: Device2
SUCCESS SEND STATUS WATCHDOG: Device-1
THREAD WATCHDOG Device2 STOPPED - LIFE TIME 2.04 minutes
THREAD WATCHDOG Device-1 STOPPED - LIFE TIME 2.04 minutes
```

Figura 29 Autonomia senza controllo adattativo

Introduzione del controllo

La seconda simulazione viene fatta abilitando il controllo. In Figura 30 è mostrata l’interfaccia grafica di log della componente *EdgeCloudSoftware*, implementata con la classe Python *AppServer*. La prima cosa che si osserva è che l’*AppServer* si collega effettivamente al flusso dati di *ChirpStack*, tramite il protocollo *MQTT*. In particolare, i dati di interesse per la funzionalità sviluppata sono quelli sullo *status* del *device*, che vengono ricevuti dalla coda topica che termina con *status*. A seguito della ricezione di un messaggio sullo stato del dispositivo l’*AppServer* determina la nuova

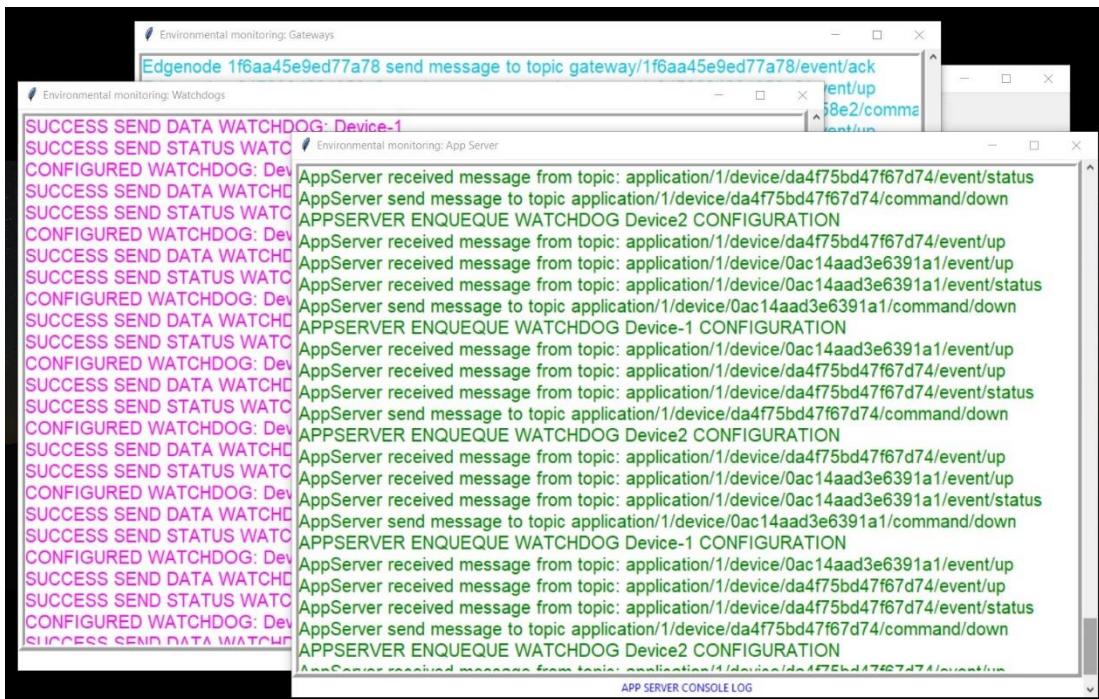


Figura 30 Interfaccia log EdgeCloudSoftware

configurazione che deve assumere il *watchdog* interessato. Si può anche notare, tra i log in Figura 30, l’invio di queste configurazioni da attuare.

Dal lato *watchdog* si può osservare il comportamento duale: La configurazione inviata viene poi recapitata al dispositivo come si può vedere in Figura 31. Alla ricezione di questa, il dispositivo si configura con i parametri ricevuti che sono *timetosend*, il periodo di trasmissione, e *timetoreceive*, che è la finestra di ricezione. Sempre in Figura 31 si vede come il valore dei parametri di configurazione cambia nel tempo, proprio a seguito del consumo della batteria e, quindi, dell’attuazione delle azioni di controllo.

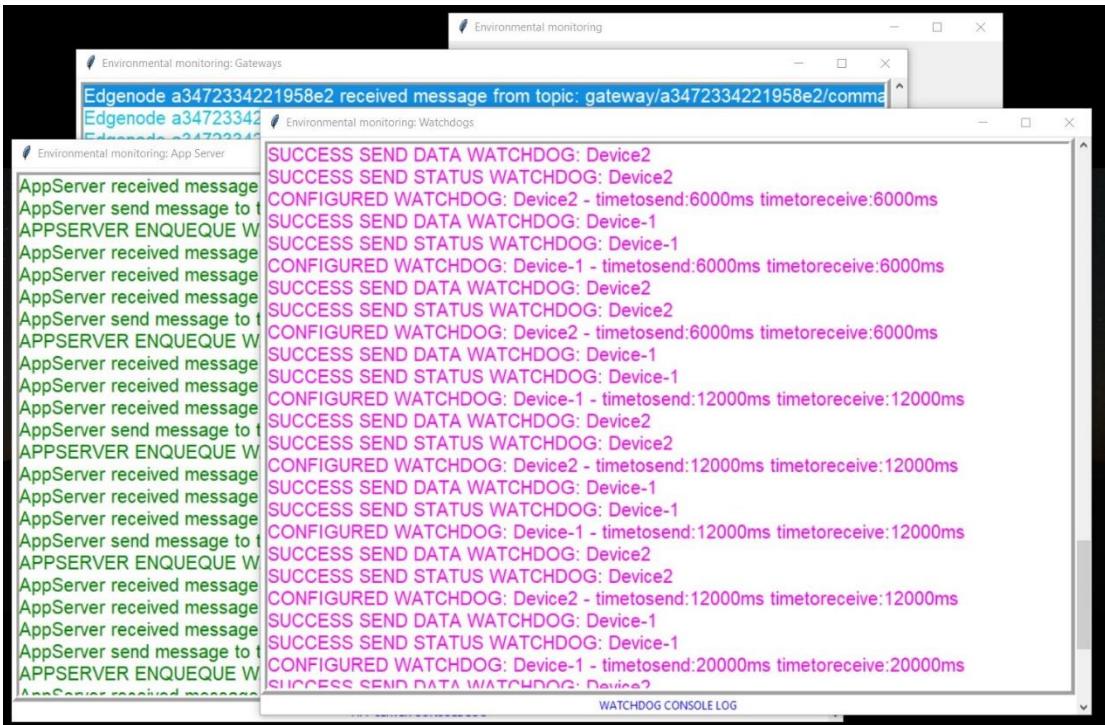


Figura 31 Configurazione watchdog

Il valore dei due parametri aumenta nel tempo, quindi l’indice di consumo (secondo la precedente definizione) diminuisce. Questo è confermato anche dal risultato della simulazione riportato Figura 32, dove questa volta l’autonomia è di circa 2.5 minuti, contro il valore di 2 minuti ottenuto con *timetosend* costante. Registrando un miglioramento del 25%.



Figura 32 Autonomia controllo adattativo

Questa prima analisi a *black-box* è molto semplice ed utile per avere un’idea dell’efficacia del software. Non è però da intendersi come un risultato globale dal momento che la simulazione fa approssimazioni molto semplicistiche. Inoltre, questo tipo di controllo agisce solo sui due parametri temporali del *watchdog*. Questo sicuramente degrada le prestazioni in termini di monitoraggio dal momento che si trasmettono meno dati, cosa che potrebbe essere più o meno accettabile a seconda del

contesto in cui si inserisce il software: la valutazione, quindi, va fatta a seconda del caso reale concreto. Logiche più complesse possono agire su altri parametri, quali la potenza del segnale di trasmissione, su cui però risulta molto più difficile agire in quanto ci sono sia limiti posti dallo standard LoRaWan, in funzione della posizione geografica del dispositivo, che limiti dovuti alla trasmissione fisica, rischiando quindi di perdere dei pacchetti.

Tuttavia, si può concludere affermando che il software si dimostra comunque efficace nel controllo, portando chiari benefici in termini di allungamento del ciclo di vita in modo autonomo.

Iterazione 5

Goal di iterazione

Questa rappresenta l'iterazione finale del progetto e ha lo scopo di implementare il requisito R1. L'obiettivo è di rendere il sistema in grado di rilevare i guasti in maniera autonoma e tempestiva, dando anche la possibilità di effettuare interventi di manutenzione mirati ed efficienti.

Seguendo la linea di progetto adottata in precedenza, viene introdotto un secondo un *MAPE-K loop* per soddisfare questo requisito. In particolare, la tattica architetturale da applicare è quella di *ping/echo*.

Component diagram

In Figura 33 è riportato il diagramma delle componenti che modella la *feature* introdotta. Il ciclo *MAPE-K* per la rilevazione dei guasti è rappresentato dalla componente *Mape_FaultDetectionAppServer*. La struttura di questa è più complessa del *MAPE-K* per l'adattamento della batteria in quanto la diagnosi dei guasti viene fatta su due livelli:

- 1) Monitoraggio dei nodi *edge*
- 2) Monitoraggio dei nodi *watchdog*

Per i *watchdog*, in realtà, il monitoraggio viene fatto in modo indiretto sfruttando la periodicità di invio di messaggi da parte di questi. La rilevazione dei guasti avviene analizzando direttamente il flusso dati già presente e se un *watchdog* è silente per un certo intervallo di tempo, detto *time-out*, allora viene dichiarato come guasto. Quindi le componenti *Monitor* ed *Execute* sono le stesse definite per il *goal* di allungamento del ciclo di vita dei *watchdog*, oggetto della precedente iterazione.

La stessa assunzione non vale per i nodi *edge*, per i quali è necessario un monitoraggio diretto. Questo compito è svolto dalla sottocomponente *MonitorFaultDetection* che si occupa di inviare dei messaggi di *ping* agli *edge-node*. In caso di un certo numero di mancate risposte, il nodo in questione viene dichiarato guasto e, quindi, la componente *ExecuteFaultDetection* si occupa di attuare le operazioni decise dalla funzione di *plan*.

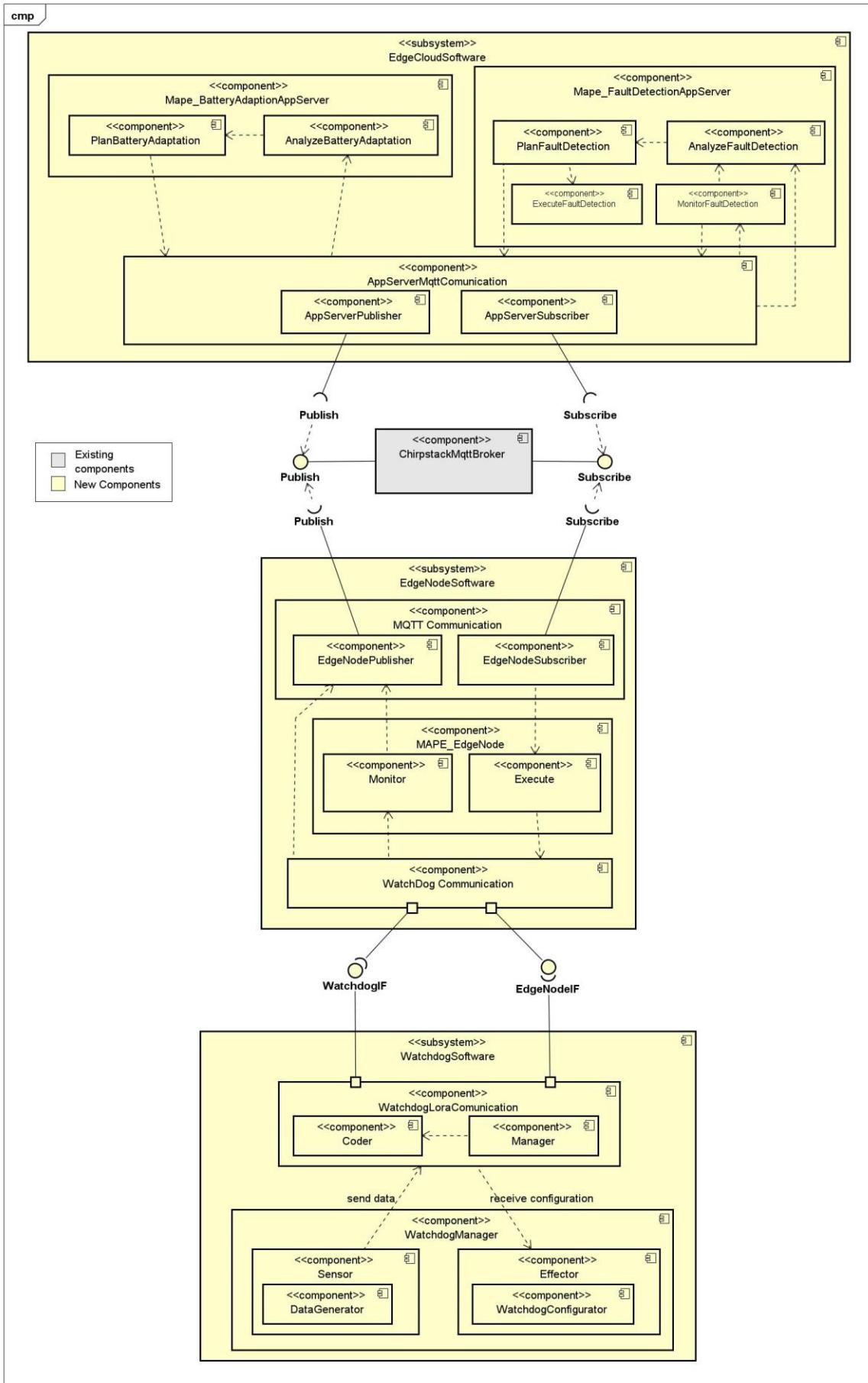


Figura 33 Component diagram iterazione 5

L'aggiunta delle funzioni *Monitor* ed *Execute* lato *EdgeCloudSoftware* vanno contro la linea di progetto adottata, ovvero di distribuire le funzioni del ciclo *MAPE-K*. Tuttavia, questa scelta è dovuta e anche ovvia dal momento che, se l'obiettivo è quello di monitorare gli *edge-node*, non è possibile implementare la funzionalità su di essi: infatti, in caso di guasti, che è il fenomeno da rilevare, non si avrebbe più un sistema di controllo affidabile.

Il ciclo di controllo ha quindi una struttura a doppio anello:

- ✓ Il primo anello di controllo che coinvolge tutte e tre le componenti *software*, con cui vengono rilevati i guasti sui *watchdog*.
- ✓ Il secondo coinvolge le componenti *EdgeCloudSoftware* e *EdgeNodeSoftware* ed è pensato per la rilevazione di guasti sui nodi *edge*.

Class diagram

Il diagramma delle classi risultato dall'implementazione di tale requisiti è riportato in Figura 34. Si può notare come non siano state apportate rilevanti modifiche alle classi presenti dalla precedente iterazione (Figura 28), grazie al riutilizzo delle componenti già esistenti. L'aggiunta consiste nel metodo della classe *AppServer* necessario per effettuare l'operazione di *ping*.

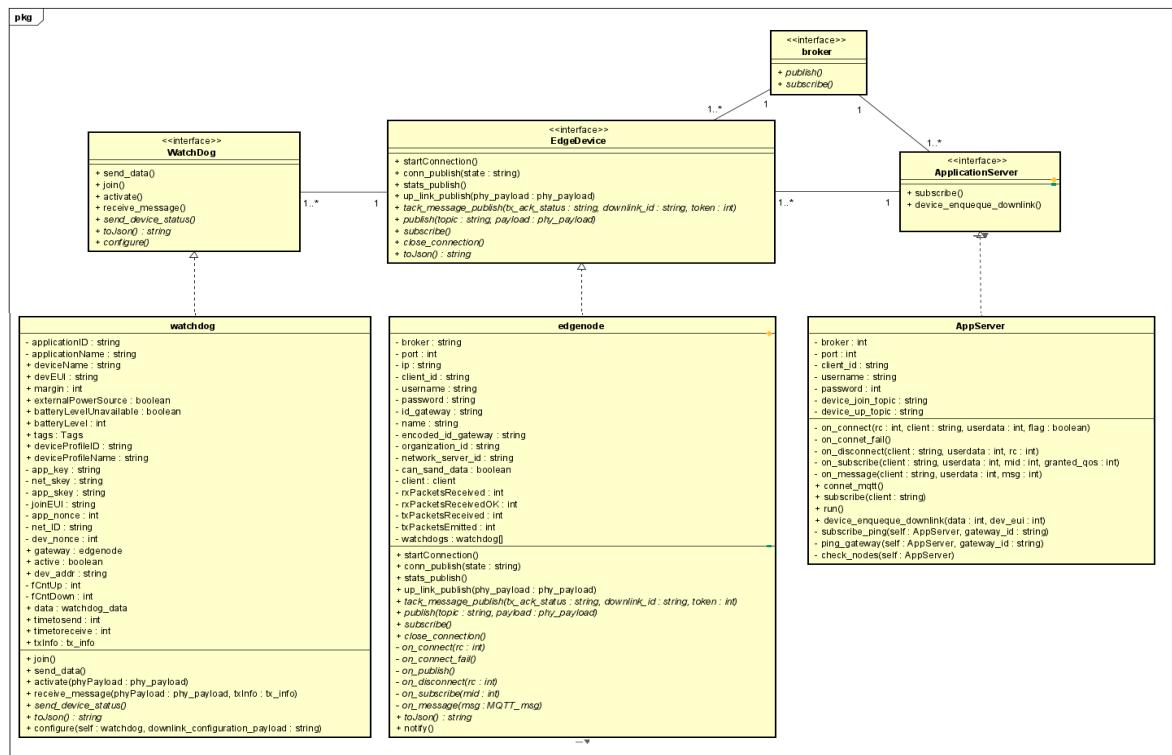


Figura 34 Class diagram iterazione 5

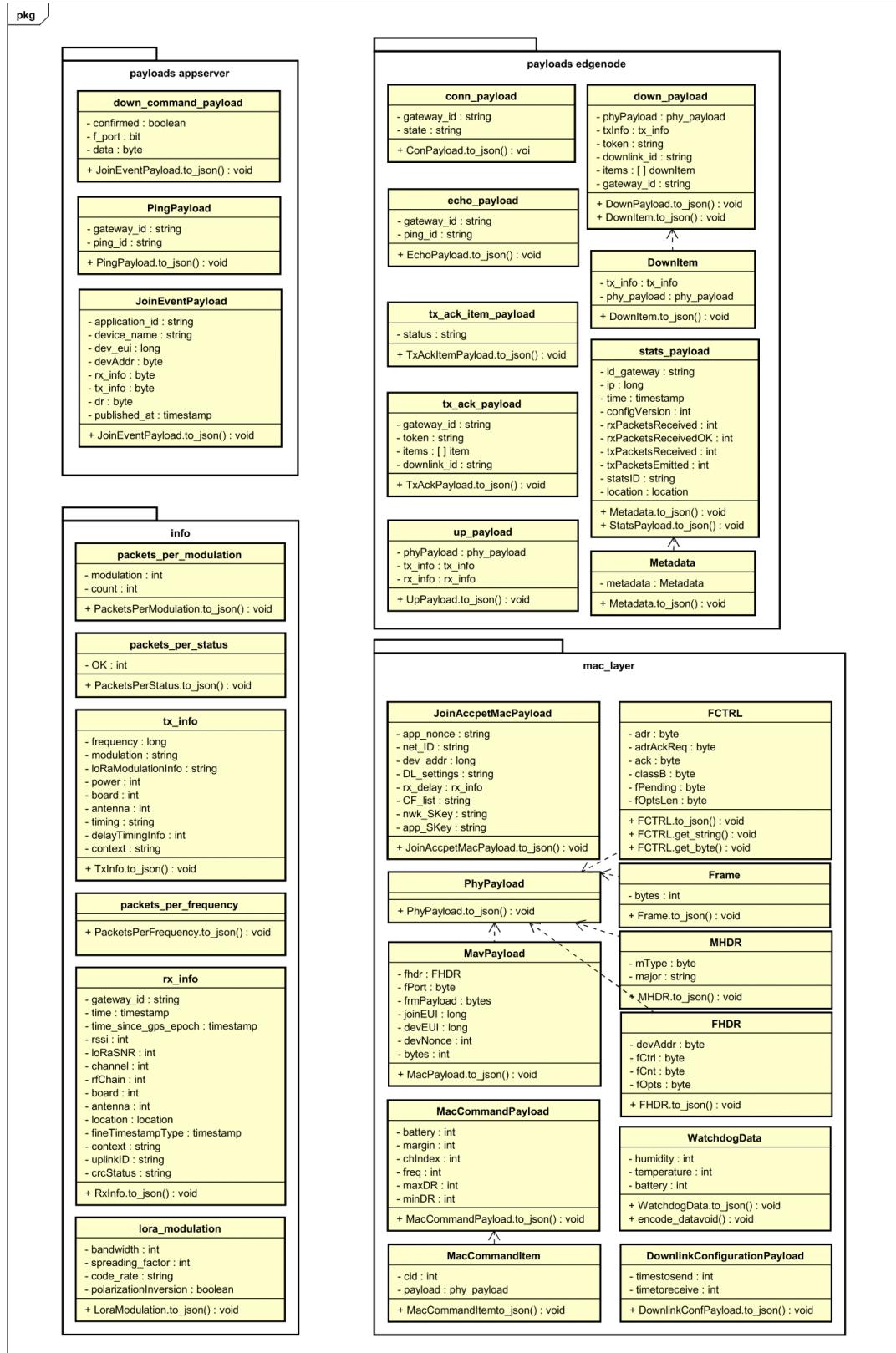
Il *package* Payloads

Il *package* *Payloads* contiene al suo interno tutte le classi necessarie ad implementare i *payloads* utilizzati nel progetto. Il *payload* è il contenuto informativo di un pacchetto *LoRaWan*. Il package è a sua volta suddiviso in altri quattro *package*:

- *appserver* : Contiene i *payloads* di cui si avvale l'*AppServer*.
- *edgenode* : Include i *payloads* utilizzati dall'*edge-node*.
- *info*: Racchiude la rappresentazioni delle informazioni di trasmissione (come la modulazione, potenza di trasmissione, frequenza, ecc..).
- *mac_layer*: Contiene la rappresentazione dei dati utili a costruire il *PhyPayload* (discusso all'iterazione 3 , pag.37)

In Figura 35, viene riportato un diagramma delle classi con le rappresentazioni descritte. In particolare, nel *package* *mac_layer* è presente la classe ***watchdog_data*** che è la rappresentazione dei dati generati dal *watchdog*, che informato *JSON* è:

```
{  
    "humidity": 10,  
    "temperature": 10,  
    "battery": 12  
}
```



1 / 1

Figura 35 Class diagram package payloads

L’algoritmo *check_nodes*

L’algoritmo *check_nodes* è stato implementato nell’iterazione 5 con l’obiettivo di soddisfare il requisito “*R1: Azioni di auto-diagnosi manutentiva di risoluzione autonoma dei guasti*”. Tale algoritmo ha il compito di eseguire un controllo su tutti i nodi della rete. In particolare, vengono eseguiti due cicli *for*: il primo controlla i *watchdog* e, analogamente, il secondo fa un check sui nodi *edge-device (gateway)*. Questi ultimi, se non inviano pacchetti, vengono interrogati periodicamente con un messaggio di *ping*. I dispositivi *watchdog*, invece, non vengono interrogati direttamente, ma viene sfruttato l’arrivo dei dati in *uplink* come segnale di vita, aggiornando la marca temporale ad ogni ricezione di un pacchetto. Questa strategia permette di semplificare il controllo sfruttando il fatto che le sentinelle devono inviare i dati periodicamente: questo risulta quindi essere una ***precondizione*** di questo algoritmo di controllo. Lo stesso discorso non vale necessariamente per i nodi *edge* dal momento che questi sono “al servizio” dei *watchdog*; pertanto, un certo nodo *edge* può anche non essere utilizzato, ma essere comunque funzionante. Quindi per robustezza, oltre all’aggiornamento della marca temporale alla ricezioni di pacchetti, viene aggiunto il controllo diretto tramite *ping*, attuando un approccio ibrido.

Per ogni ciclo di controllo, il primo *if-statement* verifica se il dispositivo ha risposto al *ping* entro il *time-out*, altrimenti viene incrementato il numero di mancate risposte (*num_failure*). Il secondo *if-statement* fa un controllo sul numero di mancate risposte: se queste sono maggiori di tre, il nodo deve essere ripristinato manualmente. Il conteggio del numero di fallimenti è utile dal momento che potrebbe succedere che un nodo non risponda ad un *ping* ma sia funzionante (es: *packet lost*, congestione della rete...). Dopo il terzo tentativo non andato a buon fine è improbabile che il dispositivo sia funzionante, ma non riesca a rispondere al ping, perciò viene etichettato come guasto (KO). La condizione di guasto, ovviamente, non è rigida: se in un momento successivo si riceve un segnale di corretto funzionamento del nodo, viene aggiornato il suo stato come funzionante (OK). Questa funzionalità viene implementata nella componente di comunicazione.

Pseudocodice:

```
Alg check_nodes{
    for-each watchdog in active watchdog do
        sleep_time = current_time - last_response;
        if (sleep_time > timeout)
            watchdog.num_failure += 1;
        endif
        if (watchdog.num_failure >= 3
            watchdog.state = KO
            watchdog.active = false
        endif
    endfor
    for-each edgenode in active edgenode do
        sleep_time = current_time - last_response;
        if (sleep_time > timeout)
            edgenode.ping(); //invio segnale di ping
        endif
        if (edgenode.pending_ping >= 3)
            edgenode.state = KO
            edgenode.active = false
        endif
    endfor
}
```

The diagram illustrates the time complexity of the pseudocode. It uses red brackets on the right side to group code segments and label them with their complexity:

- O(c)**: Groups the inner loops for both watchdog and edgenode.
- O(n)**: Groups the outer loop for both watchdog and edgenode.
- O(n+m)**: Groups the entire function.
- O(m)**: Groups the innermost if-block for edgenode.

Analisi complessità-tempo

Nel valutare la complessità tempo dell'algoritmo i procede con un approccio *bottom-up*: anzitutto si considerano le complessità dei blocchi interni. Dopodiché si valutano quelle esterne per iterazione, finché non si è valutato l'intero algoritmo.

SI definisce una costante c . Si può notare come negli *if-statement* vengano eseguite solo operazioni elementari (confronto, assegnamento, incremento...), perciò hanno una complessità $O(c)$, ovvero uno sforzo computazionale costante, indipendentemente dalla cardinalità dei nodi nella rete. Definiamo ora con n il numero di *watchdog* e con m il numero di *edge node* attivi nella rete, i 2 cicli *for* vengono eseguiti tante volte quanti sono i dispositivi attivi, rispettivamente n volte e m volte. La complessità totale dell'algoritmo sarà la somma di quest'ultimi:

$$T(n,m) = O(n+m)$$

Si può osservare che:

- la complessità dipende dal numero totale di nodi della rete.
- $T(n,m)$ vale sia per il caso peggiore che per il caso migliore.
- questo può essere considerato come l'algoritmo più esigente della rete in termini di sforzo computazionale; infatti, esso viene eseguito sul *cloud server* dove non si ha la necessità di conservare le risorse energetiche, che invece si dovrebbe tenere in considerazione sugli altri dispositivi.

Solitamente in una rete di questo tipo si ha $n > m$, ovvero che il numero di dispositivi *watchdog* è maggiore degli *edge node*. Questo porta a poter scrivere la complessità tempo in funzione del solo numero di *watchdog*, sfruttando la regola di semplificazione della notazione asintotica per la somma.

$$T(n) = O(n)$$

Implementazione:

```
def check_nodes(self):
    current_timestamp = round(datetime.now().timestamp())
    for dev_eui in self.watchdogs:
        interval_time = (current_timestamp - self.watchdogs[dev_eui].last_seen) * 1000
        if interval_time > AppServer.watchdog_timeout and self.watchdogs[dev_eui].active:
            self.watchdogs[dev_eui].num_failure += 1
        if self.watchdogs[dev_eui].num_failure >= 3 and self.watchdogs[dev_eui].active:
            self.watchdogs[dev_eui].state = WorkingStateEnum.K0.name
            self.watchdogs[dev_eui].active = False
            print(f"\033[93mWARNING\033[0m WATCHDOG {dev_eui} IS SILENT\033[93mENDC\033[0m")
    for gateway_id in self.gateways:
        interval_time = (current_timestamp - self.gateways[gateway_id].last_seen) * 1000
        if interval_time > AppServer.gateway_timeout and self.gateways[gateway_id].active:
            self.ping_gateway(gateway_id)
        if self.gateways[gateway_id].num_pending_pings >= 3 and self.gateways[gateway_id].active:
            self.gateways[gateway_id].state = WorkingStateEnum.K0.name
            self.gateways[gateway_id].active = False
            print(f"\033[93mWARNING\033[0m EDGENODE {gateway_id} IS NOT WORKING\033[93mENDC\033[0m")
```

The diagram illustrates the time complexity of the `check_nodes` function. It uses red brackets to group different parts of the code and associate them with their respective time complexities:

- `for dev_eui in self.watchdogs:` is grouped by a bracket and labeled $O(c)$.
- `if interval_time > AppServer.watchdog_timeout and self.watchdogs[dev_eui].active:` is grouped by a bracket and labeled $O(n)$.
- `self.watchdogs[dev_eui].num_failure += 1` is grouped by a bracket and labeled $O(c)$.
- `if self.watchdogs[dev_eui].num_failure >= 3 and self.watchdogs[dev_eui].active:` is grouped by a bracket and labeled $O(n)$.
- `self.watchdogs[dev_eui].state = WorkingStateEnum.K0.name` is grouped by a bracket and labeled $O(c)$.
- `self.watchdogs[dev_eui].active = False` is grouped by a bracket and labeled $O(c)$.
- `print(f"\033[93mWARNING\033[0m WATCHDOG {dev_eui} IS SILENT\033[93mENDC\033[0m")` is grouped by a bracket and labeled $O(c)$.
- `for gateway_id in self.gateways:` is grouped by a bracket and labeled $O(c)$.
- `interval_time = (current_timestamp - self.gateways[gateway_id].last_seen) * 1000` is grouped by a bracket and labeled $O(c)$.
- `if interval_time > AppServer.gateway_timeout and self.gateways[gateway_id].active:` is grouped by a bracket and labeled $O(m)$.
- `self.ping_gateway(gateway_id)` is grouped by a bracket and labeled $O(c)$.
- `if self.gateways[gateway_id].num_pending_pings >= 3 and self.gateways[gateway_id].active:` is grouped by a bracket and labeled $O(m)$.
- `self.gateways[gateway_id].state = WorkingStateEnum.K0.name` is grouped by a bracket and labeled $O(c)$.
- `self.gateways[gateway_id].active = False` is grouped by a bracket and labeled $O(c)$.
- `print(f"\033[93mWARNING\033[0m EDGENODE {gateway_id} IS NOT WORKING\033[93mENDC\033[0m")` is grouped by a bracket and labeled $O(c)$.

Figura 36 implementazione Algoritmo `check_nodes`

Analisi statica codice

L'analisi statica del codice è utile sia per scovare i difetti più comuni che per dare un *feedback* sul grado di complessità e qualità del codice. Uno degli indicatori più utilizzati per questo scopo è la complessità ciclomatica, chiamata anche complessità *McCabe*, definita come:

$$CC = E - N + 2P$$

dove N è il numero di nodi nel diagramma di flusso di controllo, E è il numero di archi e P è il numero di nodi di condizione (istruzioni *if*, cicli *while/for*).

Un *tool* utile, impiegato in questo progetto, per effettuare l'analisi statica del codice *Python* è *Radon*.

Radon è uno strumento *Python* che calcola varie metriche dal codice sorgente quali:

- La complessità di *McCabe*, ovvero la complessità ciclomatica
- Metriche grezze (righe di commento, righe vuote...)
- Metriche di Halstead
- Indice di manutenibilità

Una volta effettuata l'installazione di *Radon*⁹, abbiamo dato in pasto il nostro progetto al *tool* tramite il comando `$radon cc our_code.py`. Di seguito, si riportano i report dell'analisi.

- Complessità di *McCabe* nell'implementazione del *watchdog*(Figura 37):

```
M 127:4 Watchdog.__eq__ - C (12)
C 32:0 Watchdog - A (4)
M 199:4 Watchdog.send - A (4)
M 84:4 Watchdog.send_data - A (3)
M 158:4 Watchdog.send_device_status - A (3)
C 19:0 Tags - A (2)
M 23:4 Tags.__eq__ - A (2)
M 66:4 Watchdog.join - A (2)
M 20:4 Tags.__init__ - A (1)
M 28:4 Tags.to_json - A (1)
M 33:4 Watchdog.__init__ - A (1)
M 139:4 Watchdog.activate - A (1)
M 152:4 Watchdog.receive_message - A (1)
M 206:4 Watchdog.configure - A (1)
M 211:4 Watchdog.to_json - A (1)
```

Figura 37 Complessità di *McCabe* *watchdog*

- Complessità di *McCabe* nell'implementazione dell'*edge-node*(Figura 38):

⁹ <https://pypi.org/project/radon/>

```

M 213:4 EdgeNode.on_message - B (7)
M 162:4 EdgeNode.publish - A (4)
M 79:4 EdgeNode.start_connection - A (3)
C 43:0 EdgeNode - A (2)
M 187:4 EdgeNode.close_connection - A (2)
M 194:4 EdgeNode.on_connect - A (2)
M 56:4 EdgeNode.__init__ - A (1)
M 106:4 EdgeNode.conn_publish - A (1)
M 116:4 EdgeNode.stats_publish - A (1)
M 133:4 EdgeNode.up_link_publish - A (1)
M 146:4 EdgeNode.tack_message_publish - A (1)
M 155:4 EdgeNode.echo_message_publish - A (1)
M 181:4 EdgeNode.subscribe - A (1)
M 201:4 EdgeNode.on_connect_fail - A (1)
M 204:4 EdgeNode.on_publish - A (1)
M 207:4 EdgeNode.on_disconnect - A (1)
M 210:4 EdgeNode.on_subscribe - A (1)
M 233:4 EdgeNode.to_json - A (1)

```

Figura 38 Complessità di McCabe edgenode

- Complessità di *McCabe* nell’implementazione dell’*App Server*(Figura 39):

```

M 122:4 AppServer.check_nodes - C (11)
M 159:4 AppServer.on_message - B (7)
C 19:0 AppServer - A (3)
M 49:4 AppServer.start_connection - A (3)
M 82:4 AppServer.publish - A (3)
M 146:4 AppServer.on_connect - A (2)
M 33:4 AppServer.__init__ - A (1)
M 74:4 AppServer.down_link_publish - A (1)
M 99:4 AppServer.subscribe - A (1)
M 108:4 AppServer.subscribe_ping - A (1)
M 112:4 AppServer.ping_gateway - A (1)
M 141:4 AppServer.close_connection - A (1)
M 153:4 AppServer.on_connect_fail - A (1)
M 156:4 AppServer.on_disconnect - A (1)
M 193:4 AppServer.to_json - A (1)

```

Figura 39 Complessità di McCabe AppServer

La prima lettera dell’*output* mostra il tipo di blocco (M per modulo, C per classe). Quindi *Radon* fornisce il numero di riga della componente analizzata, il nome della classe o della funzione e la complessità di *McCabe*, espressa con un voto (da A ad F) ed infine rappresentata sotto forma di numero (da 0 a 61). In genere, una complessità inferiore o all’incirca 10 è un buon segnale. L’indice di complessità dell’intero progetto(Figura 40) generato da *Radon* è di grado A con una complessità intorno a 2 su 61.

```

357 blocks (classes, functions, methods) analyzed.
Average complexity: A (2.0616246498599438)

```

Figura 40 Complessità di McCabe dell’intero progetto

Si può osservare che l’algoritmo *check_nodes*, di cui si è discussa la complessità-tempo nel capitolo precedente ha un grado di complessità di McCabe pari a C (11/61), che è tollerabile.

Simulazione guasto

Al fine di effettuare un test della *feature* implementata viene fatta una simulazione *ad hoc* dove, durante il funzionamento, si interrompe l'esecuzione di un *nodo-edge* e/o *watchdog*.

In Figura 41 vengono mostrati i *log* dell'AppServer con l'implementazione delle funzionalità di rilevazione dei guasti. Si può vedere come periodicamente viene inviato un messaggio di *ping* ai nodi *edge*, tramite il protocollo *MQTT*, sulla coda topica definita appositamente per tale scopo. Questa schermata mostra il normale funzionamento del sistema senza guasti.

```
Environmental monitoring: Gateways
Edgenode 1f6aa45e9ed77a78 send message to topic gateway/1f6aa45e9ed77a78/event/up
Edgenode 1f6aa45e9ed77a78 send message to topic gateway/1f6aa45e9ed77a78/event/ack
Edgenode a3472334221958e2 received message from topic: gateway/a3472334221958e2/command/down
Edgenode a3472334221958e2 send message to topic gateway/a3472334221958e2/event/up
Edgenode a3472334221958e2 send message to topic gateway/a3472334221958e2/event/ack

Environmental monitoring: App Server
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/up
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/status
AppServer send message to topic application/1/device/0ac14aad3e6391a1/command/down
APP SERVER ENQUEUE WATCHDOG Device-1 CONFIGURATION
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/up
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/status
AppServer send message to topic application/1/device/da4f75bd47f67d74/command/down
APP SERVER ENQUEUE WATCHDOG Device2 CONFIGURATION
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/up
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/up
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/up
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/status
AppServer send message to topic application/1/device/0ac14aad3e6391a1/command/down
APP SERVER ENQUEUE WATCHDOG Device-1 CONFIGURATION
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/up
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/up
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/status
AppServer send message to topic application/1/device/da4f75bd47f67d74/command/down
APP SERVER ENQUEUE WATCHDOG Device2 CONFIGURATION
AppServer send message to topic gateway/f23ad78a721d2334/command/ping
PING EDGENODE f23ad78a721d2334
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/up
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/up
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/status
AppServer send message to topic application/1/device/0ac14aad3e6391a1/command/down
APP SERVER ENQUEUE WATCHDOG Device-1 CONFIGURATION
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/up
AppServer received message from topic: application/1/device/da4f75bd47f67d74/event/status
AppServer send message to topic application/1/device/da4f75bd47f67d74/command/down
```

Figura 41 Schermata di log AppServer

Scenario di guasto *edge-node*

Il guasto di un nodo *edge* può essere simulato tramite la schermata in Figura 42. In qualsiasi momento dell'esecuzione è possibile terminare l'esecuzione di uno dei nodi *edge* simulati, denotati con il loro codice identificativo univoco (*id*).

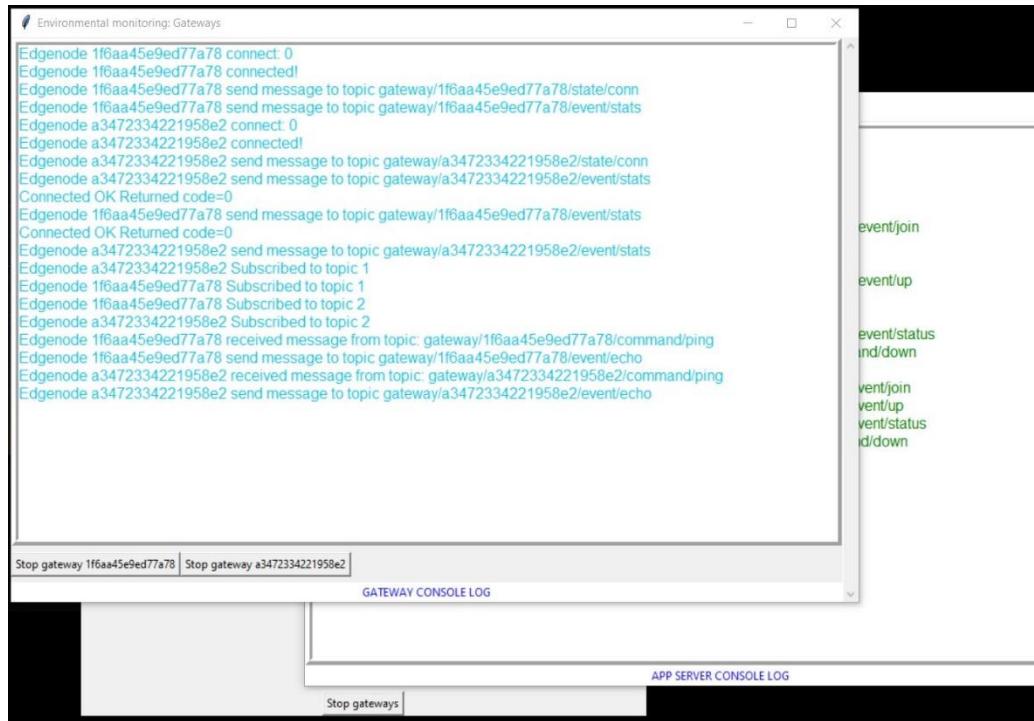


Figura 42 Schermata di log simulazione guasto nodi edge

Nella simulazione eseguita viene terminato il nodo *edge* con *id=1f6aa45e9ed77a78*.

In Figura 43 si può vedere che dopo un certo numero di *ping* l'*AppServer* classifica il nodo *edge* come non funzionante, emettendo un avviso di errore con il suo codice identificativo.

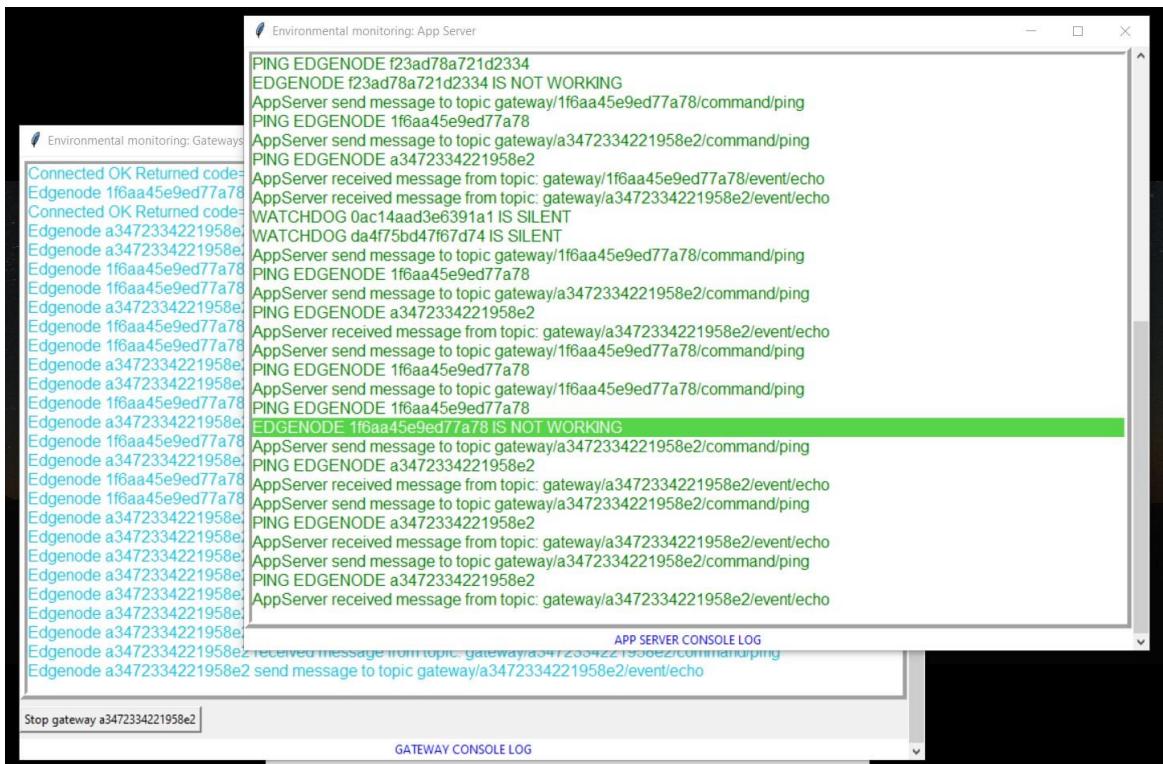


Figura 43 Rilevazione guasti

Contestualmente a questo messaggio di errore relativo al nodo *edge* terminato, sono presenti due avvisi, non previsti inizialmente dalla simulazione, che informano che i due *device* sono guasti. Questo messaggio è corretto: infatti, verificando il loro stato di esecuzione, i due *watchdog* erano in condizione di batteria scarica e quindi non trasmettevano più messaggi; quindi, l'*AppServer* li ha classificati come guasti.

Scenario di guasto *watchdog*

A conferma di questo è stata fatta una seconda simulazione (Figura 44), dove questa volta non vengono terminati nodi *edge*, con lo scopo di verificare la rilevazione dei guasti su nodi *watchdog*. Tale simulazione può essere fatta senza un particolare *setup* dal momenti che dopo un certo tempo i *watchdog* (simulati) terminano da soli perché si trovano in condizione di batteria scarica.

```

Environmental monitoring: App Server
AppServer send message to topic application/1/device/0ac14aad3e6391a1/command/down
APPSERVER ENQUEUE WATCHDOG Device-1 CONFIGURATION
AppServer received message from topic: application/1/device/0ac14aad3e6391a1/event/up
AppServer send message to topic gateway/1f6aa45e9ed77a78/command/ping
PING EDGENODE 1f6aa45e9ed77a78
AppServer send message to topic gateway/a3472334221958e2/command/ping
PING EDGENODE a3472334221958e2
AppServer received message from topic: gateway/1f6aa45e9ed77a78/event/echo
AppServer received message from topic: gateway/a3472334221958e2/event/echo
WATCHDOG 0ac14aad3e6391a1 IS SILENT
WATCHDOG da4f75bd47f67d74 IS SILENT
AppServer send message to topic gateway/1f6aa45e9ed77a78/command/ping
PING EDGENODE 1f6aa45e9ed77a78
AppServer send message to topic gateway/a3472334221958e2/command/ping
PING EDGENODE a3472334221958e2
AppServer received message from topic: gateway/a3472334221958e2/event/echo
AppServer received message from topic: gateway/1f6aa45e9ed77a78/event/echo
AppServer send message to topic gateway/1f6aa45e9ed77a78/command/ping
PING EDGENODE 1f6aa45e9ed77a78
AppServer send message to topic gateway/a3472334221958e2/command/ping
PING EDGENODE a3472334221958e2
AppServer received message from topic: gateway/1f6aa45e9ed77a78/event/echo
AppServer received message from topic: gateway/a3472334221958e2/event/echo
AppServer send message to topic gateway/1f6aa45e9ed77a78/command/ping
PING EDGENODE 1f6aa45e9ed77a78
AppServer send message to topic gateway/a3472334221958e2/command/ping
PING EDGENODE a3472334221958e2
AppServer received message from topic: gateway/1f6aa45e9ed77a78/event/echo
AppServer received message from topic: gateway/a3472334221958e2/event/echo

```

Figura 44 Rilevazione guasto *watchdog*

Risultati simulazione

Il risultato delle simulazioni mostra quindi come il sistema di controllo sia in grado di rilevare correttamente i guasti tempestivamente. Tale modello può essere complicato ulteriormente fornendo anche informazioni di dettaglio sulla natura del guasto.

Questa informazione, con questa architettura, è già immediatamente ricavabile dai dati che arrivano dal *monitoring*. Ovviamente questo scenario è semplificato: In uno scenario reale bisognerebbe tenere conto anche della numerosità dei nodi.

L'approccio applicato, però, resta comunque valido grazie alla semplicità di applicazione e alla possibilità di generalizzazione.

Conclusioni

Il progetto ottenuto, sviluppato per iterazioni seguendo la metodologia agile *AMDD* (*Agile Model Driven Development*), è in grado di simulare una piattaforma di *edge-cloud computing* per il monitoraggio ambientale.

In particolare, durante l’iterazione 0 si è fatta luce sulle tecnologie esistenti nell’ambito delle reti *IoT* e si sono gettate le basi per le iterazioni successive.

L’iterazione 1 è stata dedicata all’installazione delle componenti *Chirpstack* e alla verifica dell’ avvenuta comunicazione tra esse.

Durante le iterazioni 2 e 3 è stata portata a termine la realizzazione delle componenti simulate, rispettivamente *edgenode* e *watchdog* e si sono svolti i test di unità, comunicazione e integrazione.

Nelle ultime due iterazioni, sono stati implementati i requisiti di auto-adattamento definiti all’iterazione 0 e validati attraverso scenari di test.

Il progetto realizzato si basa sullo *stack Chirpstack*, che fornisce componenti *open-source* per reti *LoRaWan*. Il suo utilizzo ha permesso di sfruttare i servizi e l’infrastruttura esistente collaudata, come le *API* esposte per trasmettere un *payload* in *downlink* e la visualizzazione dei messaggi in *uplink*. Le funzionalità offerte, ovviamente, non si fermano a queste citate: di particolare interesse sono le *features* che riguardano la gestione a più basso livello dei nodi della rete e consistono nel mettere in atto delle azioni di configurazione per ottimizzare l’impiego della stessa. Queste funzionalità, definite dallo standard *LoRaWan*, sono implementate dalla componente *Chirpstack NetworkServer*. Questo meccanismo prende il nome di *Adaptive Data Rate*¹⁰ e consiste nel controllo dei seguenti parametri di trasmissione:

- *Spreading factor*
- *Bandwidth*
- *Transmission power*

Questo meccanismo è relativo a dettagli di basso livello, non simulati dal *software* realizzato, che potrebbe però avere applicazioni interessanti ai fini dell’ottimizzazione dell’utilizzo della rete con il minimo consumo di energia da parte dei *device*.

¹⁰ <https://www.chirpstack.io/network-server/features/adaptive-data-rate/>

Sviluppi futuri

Questo progetto può avere diversi sviluppi futuri a seconda degli scopi di interesse.

Un primo obiettivo di applicazione può essere la **simulazione** di reti *LoRaWan* che siano le più realistiche possibili e porre, quindi, l'enfasi sulle relative componenti. Questo richiederebbe la definizione del comportamento dei nodi in modo estremamente dettagliato.

Un secondo scenario, invece, interamente oggetto di questo progetto, è il *design* e la validazione di **logiche di auto-adattamento** per il sistema, astraendo le complicazioni introdotte dall'utilizzo di dispositivi fisici che, a volte, non sono facilmente disponibili o lo sono in modo limitato. Si parla sempre di simulazione, ma con maggiore enfasi sulle logiche di controllo.

L'argomento trattato è sicuramente molto di interesse e con un'applicazione sempre più crescente nel mondo. Si tratta, inoltre, di una tecnologia non ancora completamente matura che, quindi, apre le porte alla ricerca. La sua evoluzione e standardizzazione si potranno osservare solo negli anni avvenire.