Relazione del progetto di Informatica III



Laurea Magistrale in Ingegneria Informatica

I Anno

A.A. 2021/2022

Modulo di Progettazione e

Algoritmi

Prof. Patrizia Scandurra

cod. 38068

Wasim Essbai 1060652

Matteo Locatelli 1059210

Nicola Zambelli 1053015

Sommario

[Introduzione 5](#_Toc97079782)

[Obiettivo 6](#_Toc97079783)

[W3C 7](#_Toc97079784)

[LoRaWan 9](#_Toc97079785)

[MAPE-K loop 11](#_Toc97079786)

[MQTT 12](#_Toc97079787)

[ChirpStack 14](#_Toc97079788)

[Iterazione 0 16](#_Toc97079789)

[Analisi dei requisiti 17](#_Toc97079790)

[R0 – Azioni di riconfigurazione per allungamento del ciclo di vita delle sentinelle: 17](#_Toc97079791)

[R1 – Azioni di auto-diagnosi manutentiva di risoluzione autonoma dei guasti: 17](#_Toc97079792)

[User-Stories 19](#_Toc97079793)

[Lato edge device: 19](#_Toc97079794)

[Lato application server: 20](#_Toc97079795)

[Lato watchdog: 21](#_Toc97079796)

[Deployment Diagram 22](#_Toc97079797)

[Deployment Diagram Informale 22](#_Toc97079798)

[Deployment Diagram UML 23](#_Toc97079799)

[Iterazione 1 24](#_Toc97079800)

[Component diagram 24](#_Toc97079801)

[Class diagram 25](#_Toc97079802)

[Iterazione 2 26](#_Toc97079803)

[Goal di iterazione 26](#_Toc97079804)

[Component diagram 27](#_Toc97079805)

[Class diagram 28](#_Toc97079806)

[Test comunicazione 29](#_Toc97079807)

[Iterazione 3 30](#_Toc97079808)

[Goal di iterazione 30](#_Toc97079809)

[Component diagram 31](#_Toc97079810)

[Class diagram 33](#_Toc97079811)

[Il modulo *coder.py* 34](#_Toc97079812)

[State machine watchdog 37](#_Toc97079813)

[Test funzionamento 38](#_Toc97079814)

[Unit test 38](#_Toc97079815)

[Integration test 39](#_Toc97079816)

[Iterazione 4 40](#_Toc97079817)

[Component diagram 40](#_Toc97079818)

[Class diagram 42](#_Toc97079819)

[Iterazione 5 43](#_Toc97079820)

[Component diagram 43](#_Toc97079821)

[Class diagram 45](#_Toc97079822)

[L’algoritmo *check\_nodes* 46](#_Toc97079823)

[Pseudocodice 47](#_Toc97079824)

[Analisi complessità-tempo 48](#_Toc97079825)

[Implementazione 49](#_Toc97079826)

[Analisi statica codice 50](#_Toc97079827)

[Manuale d’installazione e d’uso 52](#_Toc97079828)

# 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 infine il *cloud server*:

* I ***watchdog*** sono i nodi sentinella, hanno il compito di campionare i dati e trasmetterli ai nodi edge. Tali dispositivi, tipicamente, sono mobili e alimentati a batteria; perciò, l’obbiettivo 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 discusso 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 dispostivi 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 va quindi progettata con cura e le cui azioni correttive devono essere 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 i linguaggi di markup che i protocolli di comunicazione.

Nel nostro caso siamo interessati al paragrafo 4.1.10 della guida, sul 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 di nostro 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 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 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 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.

Immagine che contiene tavolo

Descrizione generata automaticamente

Figura Layer del protocollo LoRaWan

*LoRaWan per l'Europa*

*LoRaWan* definisce dieci canali, otto dei quali sono multi-data da 250 bps a 5,5 kbps, un singolo canale LoRa ad alta velocità di trasmissione dati, a 11 kbps e 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 e suddiviso in 4 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,

essa 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 Schema del modello MAPE-K loop

L’obbiettivo è quello di utilizzare questo modello concettuale inserendo un ciclo MAPE-Kper 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 che *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 consegna dei messaggi pubblicati ai *subscriber*, iscritti alle rispettive code topiche. Un *subscriber* può essere iscritto anche a più *topic*, quindi 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 questi *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 un’interfaccia web per la gestione dei dispositivi e 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*

Architettura ChirpStack

Il grafico seguente mostra come sono collegati i componenti del *ChirpStack LoRaWAN Network Server*:



Figura 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 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...

# Iterazione 0

## Analisi dei requisiti

Di seguito vengono elencati gli obiettivi che si vogliono raggiungere con questo progetto.

### 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 *watchdog*. | La batteria dei *watchdog* scende al di sotto di determinate soglie. (50%, 30%, 15%, 10%) | Al superamento di ogni soglia è necessario:   * Diminuire la frequenza di trasmissione dei dati da parte dei *watchdog*. * Diminuire la potenza di trasmissione dei *watchdog*. |

Figura 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 *watchdog* in maniera autonoma.  (*self-recovery)* | *Watchdog* silente, cioè non trasmette dati per un certo intervallo di tempo.  (Ad es. 60s) | L’*edge-node* tenta di ripristinare il corretto funzionamento del *watchdog* guasto tramite il suo riavvio. |
| S4 | Allerta in caso di guasto ad un *watchdog* e identificazione esatta delle sue caratteristiche fisiche e geografiche. | Guasto ad un *watchdog* non recuperabile. | L’*edge-node* allerta un operatore per un intervento manuale sul nodo *watchdog* guasto indicando le sue caratteristiche. |
| S5 | Rendere il sistema in grado di gestire guasti agli *edge-node* in maniera autonoma.  (*self-recovery*) | Guasto di un *edge-node*: non riceve i dati inviati dai *watchdog* e, quindi, non può inoltrarli all’*application server*. | * L*’application server* ridistribuisce il traffico di dati che arrivava all’*edge-node* guasto verso altri *edge-nodes*. * L*’application server* notifica il personale del guasto avvenuto. |
| S6 | Limitare la perdita di informazioni dovute a gusti sull’*application server.* | Malfunzionamenti dell’*application server* che, quindi, non è in grado di ricevere e/o elaborare i dati trasmessi dai *gateway*. | * Gli *edge-node* devono mantenere i dati ricevuti dai *watchdog* in un *buffer* locale finché il server non riprende la sua operatività. * Gli *edge-node* devono notificare il guasto ad un indirizzo di rete dedicato e ritenuto affidabile. |

Figura 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 indentificare 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 mal 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 fittizi 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 : *watchdog, 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 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 Deployment diagram in UML

# Iterazione 1

Durante questa iterazione si è iniziato lo sviluppo dell’intera architettura software seguendo un approccio di tipo *top-down*.

Parallelamente è stato installato lo stack *ChirpStack* su macchina virtuale *Ubuntu* e si è fatto il *set up* dell'ambiente di lavoro.

## Component diagram

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



Figura Component diagram iterazione 1

## Class diagram

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



Figura Class diagram iterazione 1

# 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 interfaccerà direttamente con il *NetworkServer* di *Chirpstack*, implementando quindi anche le funzionalità di interfacciamento, offerte dalla componente *GatewayBridge,* sempre fornita 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:

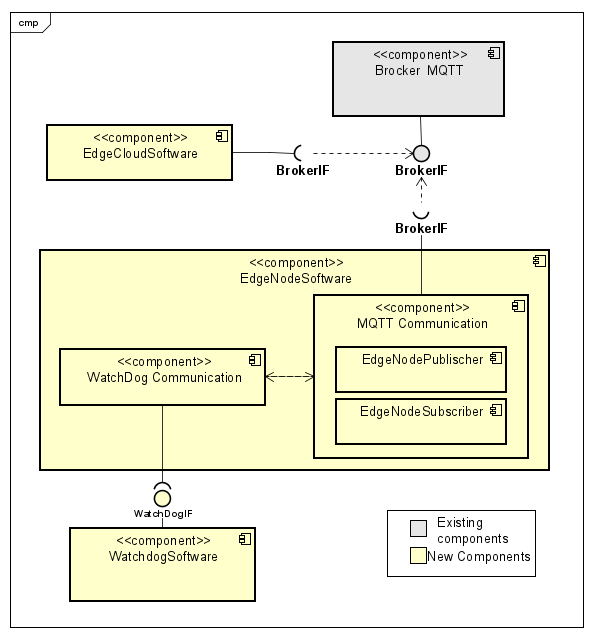
* *WatchDog Communication* per ricevere dati da più *watchdog* mediante l’interfaccia *WatchDogIF*
* *MQTT Communication* per scambiare dati con l’*application server* appoggiandosi sull’interfaccia *BrokerIF* messa a disposizione da un *Broker MQTT* già esistente (non necessita implementazione)

Figura Component diagram iterazione 2

## Class diagram

Rispetto al diagramma delle classi precedente è stato aggiunto un’interfaccia *broker* (già esistente) ed è stata esplosa 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.

Figura Class diagram iterazione 2



## Test comunicazione

La verifica della corretta comunicazione viene fatto avviando la componente e controllare appunto che i pacchetti inviati siano ricevuti nella *dashboard* di *Chirpstack.*

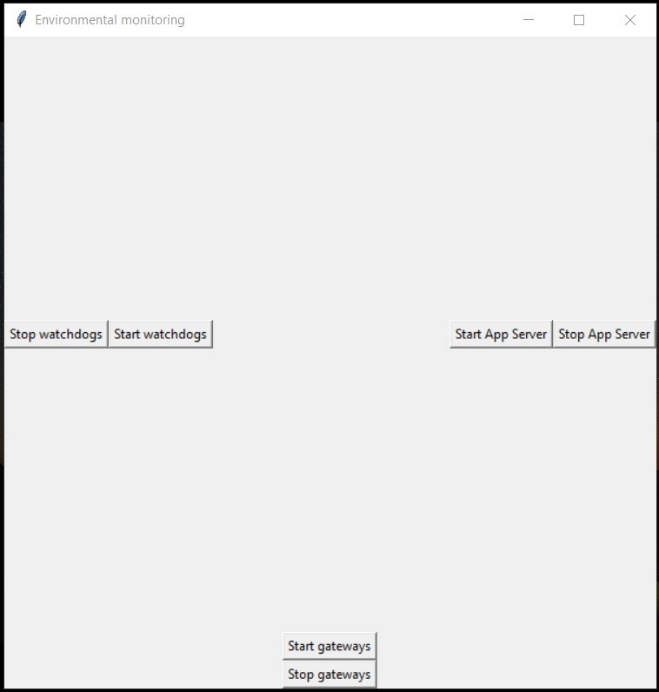
Innanzitutto, il programma si presenta con una semplice interfaccia grafica, riportata in Figura 14*,*con cui si avviano le tre componenti principali. In questa fase l’unica componente attiva è quella dell’*edge-node.*

Figura Interfaccia grafica principale

Immagine che contiene testo

Descrizione generata automaticamenteAll’avvio della componente si ha la schermata mostrata in Figura 15, dove si vede che la comunicazione avviene correttamente, e che i nodi *edge* di collegano correttamente al *broker MQTT*.

Figura Interfaccia di log edge-node

Questo viene verificato guardando dall’interfaccia grafica del ChirpStack*ApplicationServer(*Figura 16*)*¸ dove si vede la corretta connessione.

Immagine che contiene testo

Descrizione generata automaticamente

Figura Chirpstack gateway dashboard

La componente sviluppata è quindi in grado di connettersi al *borkerMQTT* 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 che è il *watchdog*. Si va quindi a definire la componente *WatchdogSoftware*.

L’obiettivo è quello di simulare con una componente software il comportamento di un tardare 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, che rappresenta quindi un altro obiettivo di tale fase.

La necessità di tale livello di dettaglio nella simulazione è dovuta all’interfacciamento con il *ChirpStackNetworkServer*, che si aspetta i dati secondo questo formato. Questa funzionalità anche è incapsulata nel *ChirpStackGatewayBridge*, di cui però non si fa uso in questo progetto, quindi viene implementata direttamente.

Il *watchdog* deve anche essere 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 la decodifica, dei messaggi ricevuti.
* Una componente *Manager* .

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 simulare 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 i seguenti:
  + La frequenza di campionamento.
  + Periodo di trasmissione.
  + Finestra di recezione.

In Figura 17 viene riportato il diagramma delle componenti finale di questa iterazione, che va integra il digramma prodotto nella precedente aggiungendo le componenti descritte.

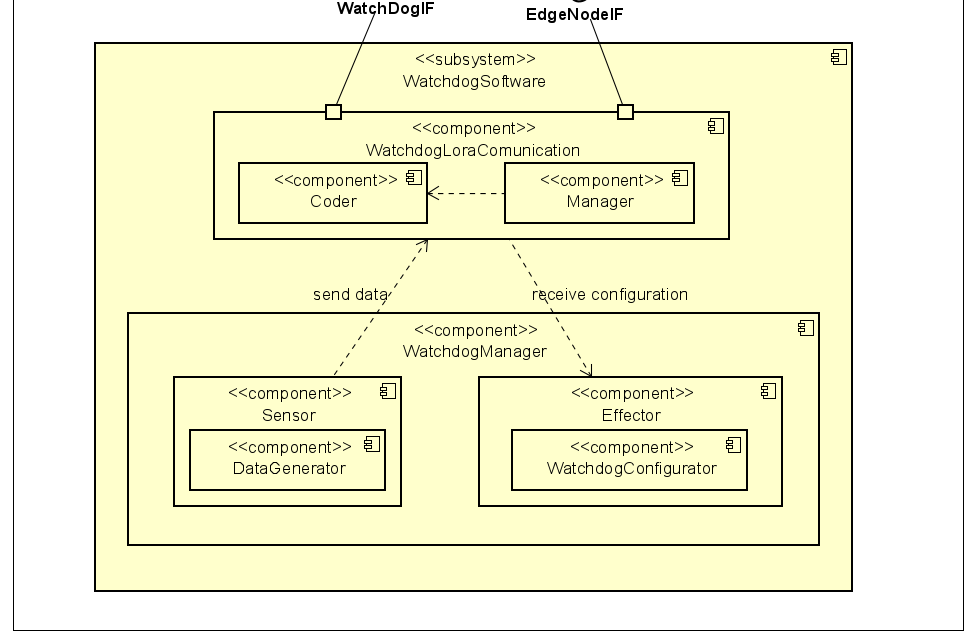
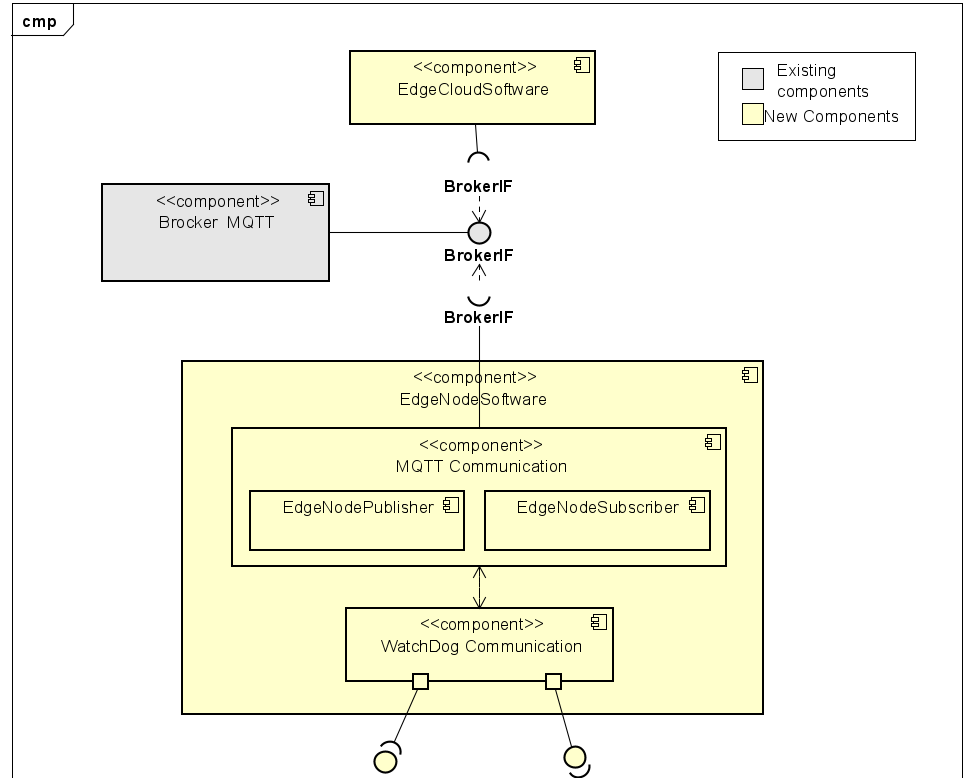


Figura Component diagram iterazione 3

## 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.



## Il modulo *coder.py*

Il modulo *coder.py* 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, con i dati generati dal *watchdog,* e di poterli trasmettere attraverso rete *LoRaWan*.

Immagine che contiene testo

Descrizione generata automaticamente

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

(Mettere il link in bibliografia e mettere il riferimento)

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

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. L’obiettivo del progetto è di mettere in evidenza l’interazione e flusso dei dati ad alto livello.

*PHYPayload* ha la struttura riportata nelle seguente immagine:

Immagine che contiene testo

Descrizione generata automaticamente

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 messsaggio in *downlink*) o codificare (nel caso di un messaggio in *uplink*) i byte nei seguenti campi :

* DevAddr: che contiene le informazioni del device che ha generato il messaggio
* Mic e FHDR: che sono dei campi per la verifica dell’integrità del pacchetto
* PhyPayLoad: che include le informazioni del pacchetto, come messaggi di configurazione (*downlink*), dati campionati (*uplink*), stato della batteria (*uplink*)…

MHDR ( MAC header) -> è di 1 byte e contiene l’informazione sul tipo di messaggio trasmesso e la versione del protocollo

MacPayload -> è quello che invece contiene il payload vero e proprio

FHDR (Frame header)

FPort

ecc…

MIC (Message integrity code) -> è di 4 byte e server per verificare l’integrità del pacchetto e anche la sicurezza di questo. Per calcolarlo servono anche le chiavi con cui vengono cifrati i messaggi

Per le operazioni di codifica/decodifica dal formato base64si utilizza al 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.

base64.b64decode()

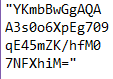
base64.b64encode()

encodePhyPayLoadFromJson()

LoRaWan Packet

Base64

JSON



01101010010101000011010100010100011111000101010100010010001

decodePhyPayLoad()

## State machine watchdog



## Analisi dinamica

La fase di test si suddivide in due parti:

* Test di unità sulla componente *coder* per verificare la corretta codifica e decodifica dei messaggi.
* Test di integrazione globale per verificare che i messaggi generati dal *watchdog* tramessi attraverso un *edge-node* siano recapitati al *ChirpStackApplication* server

### Unit test

Il test di unità viene fatto con l’ausilio del tool *Unittest*, presente anche 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 lo sforzo anche per test di regressione, 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 128bit, che risulta essere una tecnica molto sicura con 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()* anch’esso svolge le due funzioni citate, però sul campo *macPayload*.
* *computeJoinRequestMic()* si occupa di calcolare il campo *MIC* di un messaggio di tipo ***JoinRequest.***
* *decodeDataPayloadToMacCommands ()* decodifica i dati ricevuti nel campo *frmPayload* per ottenere i comandi definiti dal protocollo LoRaWan.
* *computeJoinAcceptMic()* si occupa di calcolare il campo *MIC* di un messaggio di tipo ***JoinAccept.***
* *computeJoinDataMic()* si occupa 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

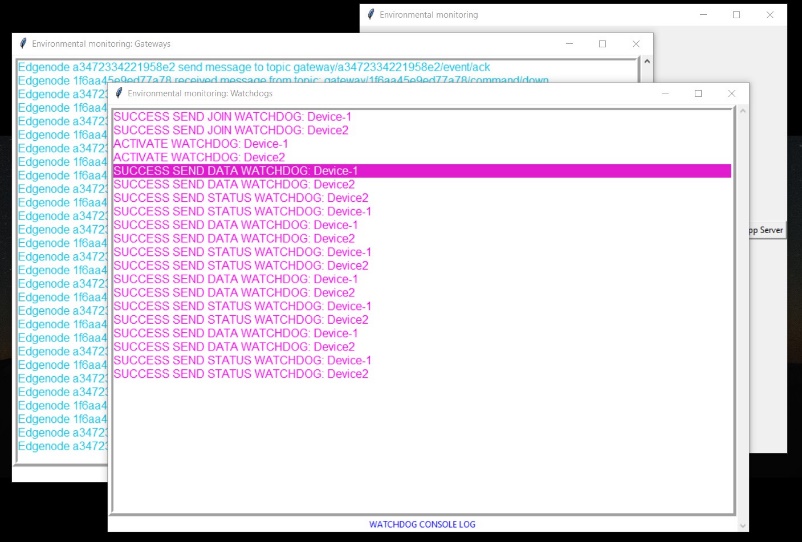
Il test di integrazione è stato fatto avviando semplicemente la componente dall’interfaccia grafica del programma, verificando poi dall’interfaccia grafica di *ChirpStack* la corretta ricezione dei messaggi. In Figura 17 si può vedere come la connessione(processo di ***Join*)** e l’invio dei messaggi vada a buon fine.

Figura Interfaccia grafica watchdog

Immagine che contiene testo

Descrizione generata automaticamenteIl corretto funzionamento può essere riscontrato osservando ancora una volta i pacchetti che arrivano nella *dashboard* dei *dispositivi*:

Figura Dashboard di ChirpStack per i device

Si può quindi concludere dicendo 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, come sfruttare le funzionalità base dello stack *ChirpStack* per ottenere un sistema *self-adaptive,* aggiungendo nuovi *layer*.

# Iterazione 4

In questa fase del progetto si è sviluppato il funzionamento dell’*application server* e, in parallelo, è stato implementato il requisito R0. Per fare ciò è stato necessario dettagliare ulteriormente la componente *EdgeCloudSoftware* ed introdurre un *MAPE\_K loop* per gestire l'*adaptation goal* di configurazione per allungare il ciclo di vita dei *watchdog*.

## Component diagram

Rispetto al *component diagram* dell’iterazione precedente sono stati dettagliati tre componenti:

* Il *ChirpstackMQTTBroker*, il quale comunica con l’*application server* e gli *edge-nodes* non con una sola interfaccia (chiamata precedentemente *BrokerIF*), ma mediante due interfacce distinte, una di *publish* e una di *subscribe*
* L’*EdgeNodeSoftware*, nel quale è stata aggiunta la componente *MAPE\_BatteryAdaptationEdgeNode* con lo scopo di
  + Monitorare la batteria dei *watchdog* tramite il sottocomponente *BatteryMonitor* (*Monitor*)
  + Configurare i vari *watchdog* in base alle analisi svolte sul livello della batteria eseguite dall’*EdgeCloudSoftware* tramite il componente *BatteryAdaptationExecute* (*Execute*)

Inoltre, avendo distinto le funzioni di *publish* e di *subscribe* nelle interfacce esposte dal *broker*, si è dettagliato il componente di *MQTTCommunication* con due sottocomponenti *EdgeNodePublisher* ed *EdgeNodeSubscriber*, che richiedono rispettivamente l’interfaccia *Publish* e l’interfaccia *Subscribe* del *broker MQTT*

* L’*EdgeCloudSoftware* è composto da due sottocomponenti:
  + *AppServerMQTTCommunication* 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* del *broker MQTT*
  + *MAPE\_BatteryAdaptationAppServer*, invece, costituisce le funzioni di *Analize* e di *Plan* del *MAPE loop* tramite le sottocomponenti *AnalizeBatteryAdaptation* e *PlanBatteryAdaptation*

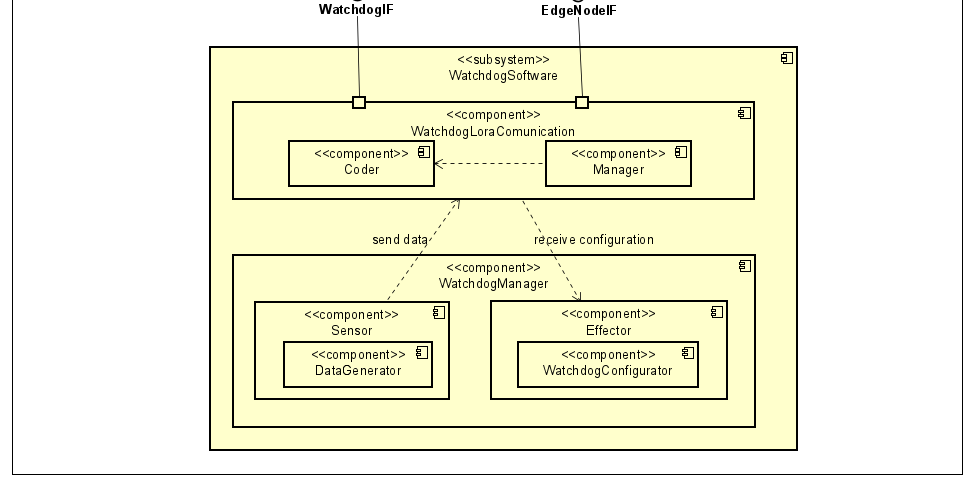
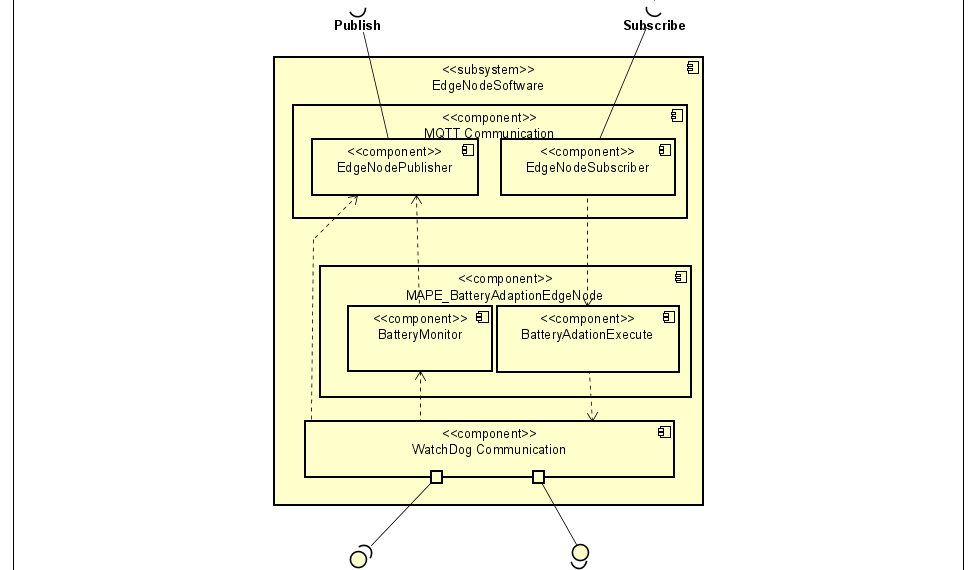
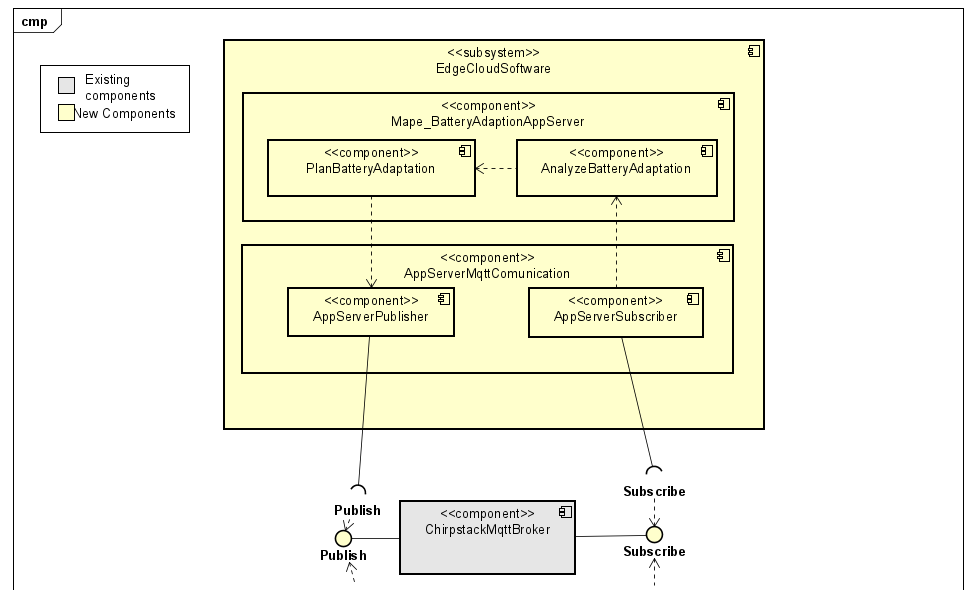


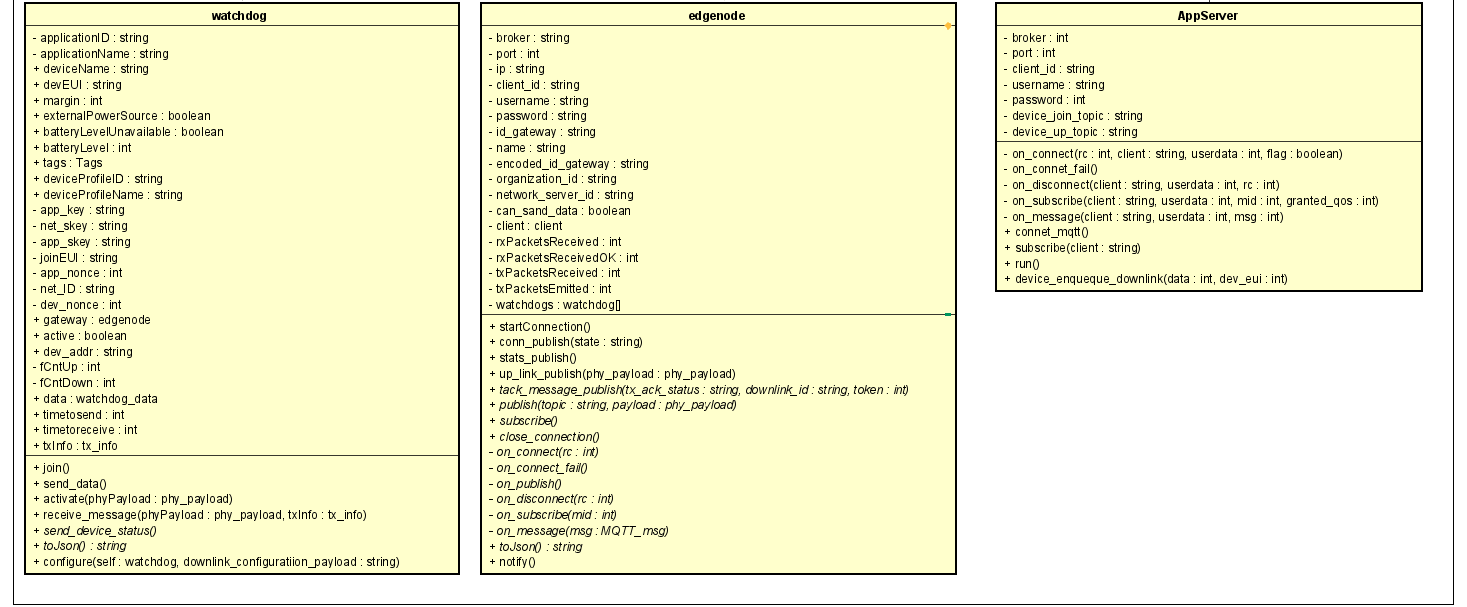
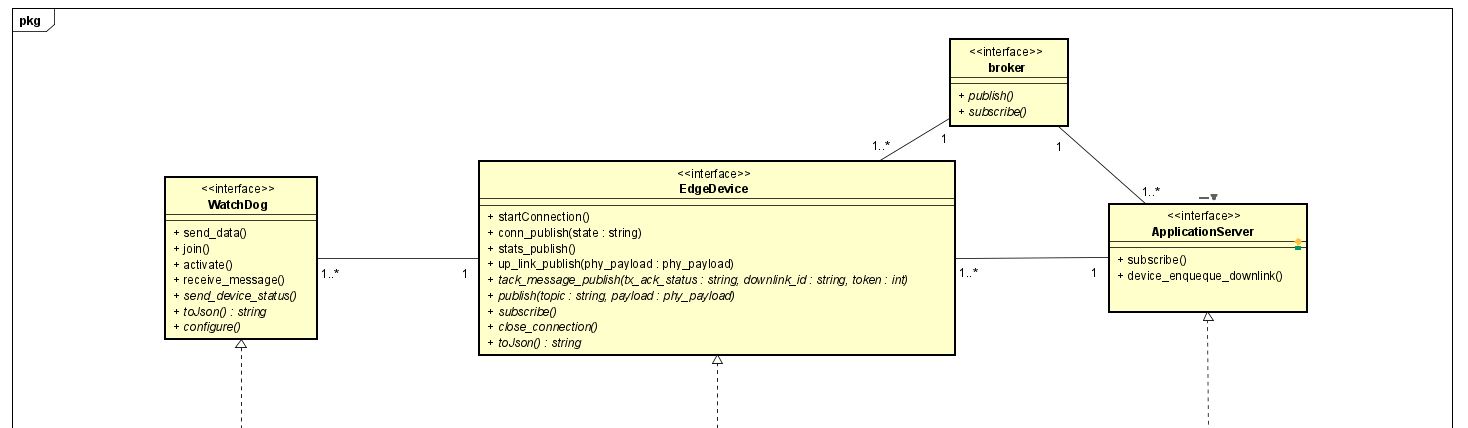
Figura Component diagram iterazione 4

## 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 MQTT*

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.



# Iterazione 5

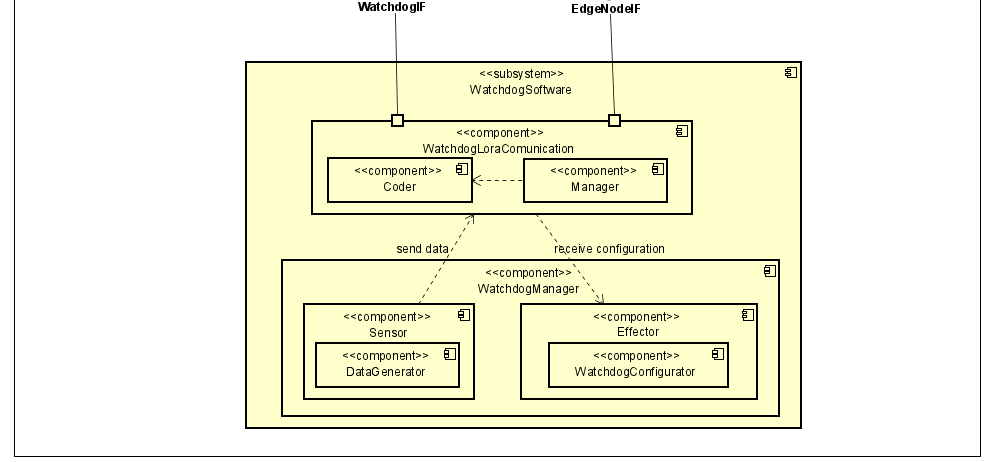
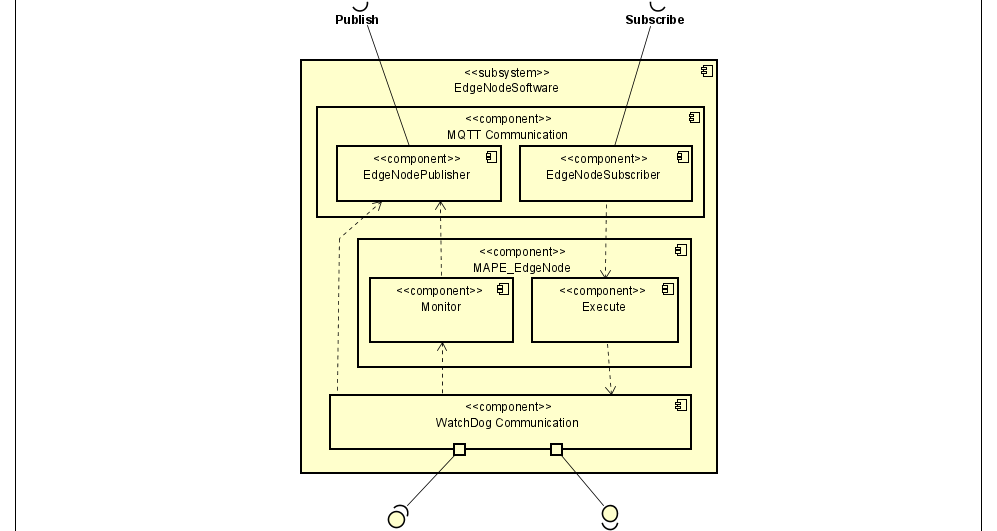
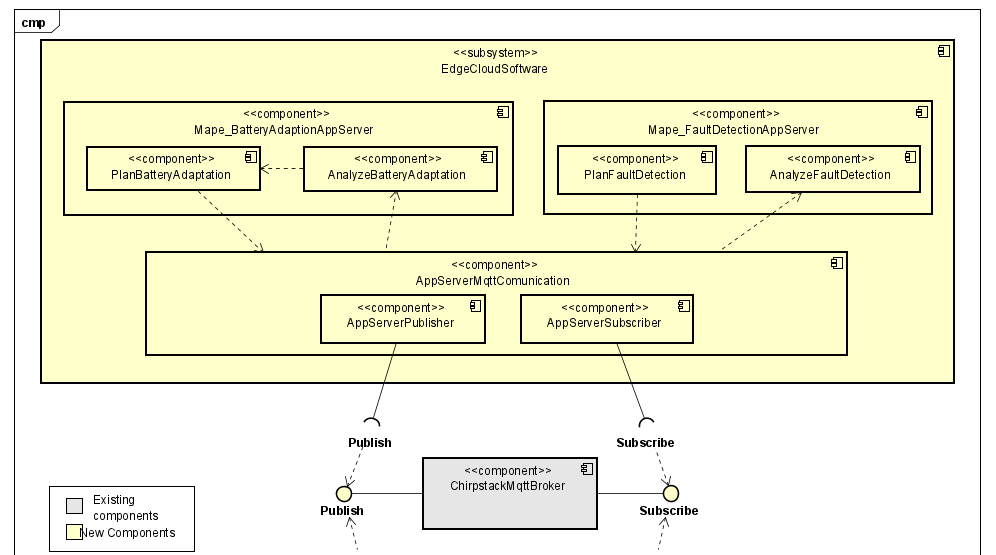
Quest’ultima iterazione del progetto ha l’obiettivo di implementare il requisito R1. Per fare ciò è stato necessario dettagliare ulteriormente la componente *EdgeCloudSoftware* ed introdurre un ulteriore *MAPE-K loop* per gestire il goal di auto-diagnosi dei guasti.

## Component diagram

La struttura di questo *component diagram* è la medesima rispetto a quello dell’iterazione 4. L’unica differenza si nota nella componente *EdgeCloudSoftware* nel quale è stato introdotto *Mape\_FaultDetectionAppServer*.

Quest’ultimo è costituito da due sottocomponenti, *AnalyzeFaultDetection* e *PlanFaultDetection*, le quali corrispondono alle funzioni di *Analyze* e di *Plan* per il *MAPE\_K loop* relativo al goal della gestione dei guasti.

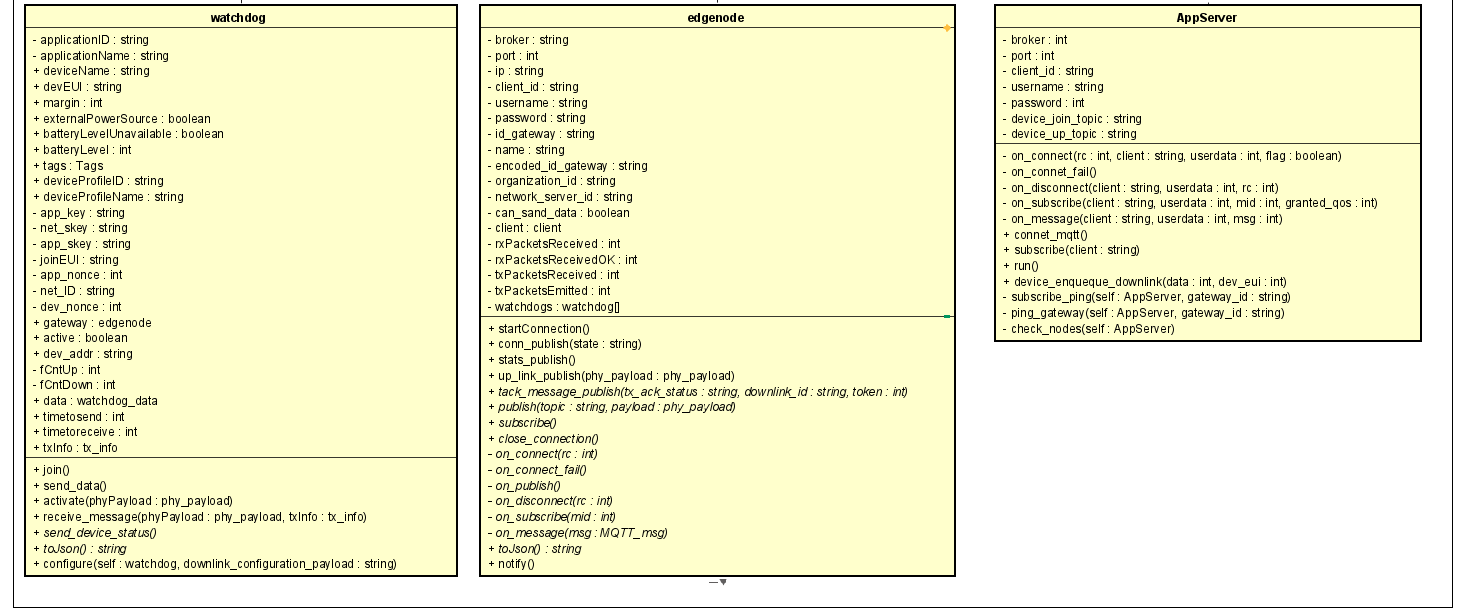
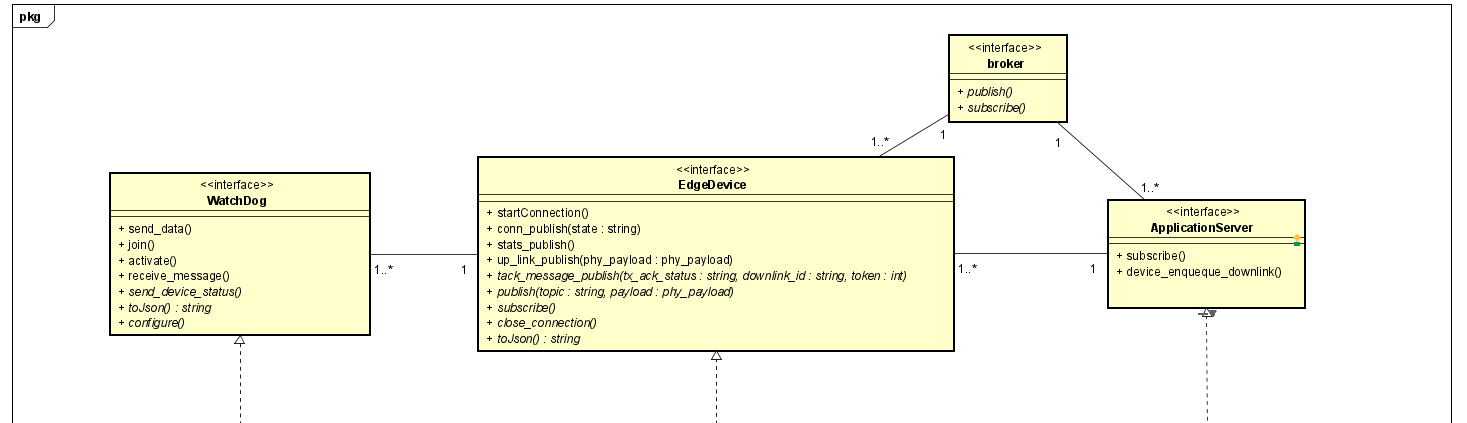
Nell’*EdgeNodeSoftware*, invece, si sono mantenute le componenti di *Monitor* e di *Execute* introdotte nell’iterazione 4.



## Class diagram

La struttura è identica a quella del *class diagram* dell’iterazione precedente.

L’unica differenza si può trovare nella classe *AppServer*, nella quale sono stati aggiunti dei metodi necessari per implementare la funzione di *ping-echo*.



## L’algoritmo *check\_nodes*

L’algoritmo *check\_nodes* è stato implementato nell’iterazione 5 con l’obbiettivo 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 che 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, che risulta quindi essere una ***precondizione*** di questo algoritmo di controllo. Lo stesso discorso non vale necessariamente anche per i nodi *edge* dal momento che questo sono “al servizio” delle sentinelle; 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)

O(c)

watchdog.num\_failure =+ 1;

**endif**

O(n)

**if** (watchdog.num\_failure >= 3

watchdog.state = KO

O(c)

watchdo.active = false

**endif**

**endfor**

O(n+m)

**for-each** edgenode **in** active edgenode **do**

sleep\_time = current\_time – last\_response;

**if** (sleep\_time > timeout)

edgenode.ping(); //invio segnale di ping

O(c)

**endif**

O(m)

**if** (edgenode.pending\_ping >= 3)

edgenode.state = KO

O(c)

edgenode.active = false

**endif**

**endfor**

}

### Analisi complessità-tempo

Nel valutare la complessità tempo dell’algoritmo procediamo con un approccio *bottom-up*: anzitutto consideriamo le complessità dei blocchi interni, dopodiché valutiamo quelle esterne per iterazione, finché non abbiamo valutato l’intero algoritmo.

Definiamo una costante *c*, è facile notare che negli *if-statement* vengono 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 abbiamo la necessità di conservare le risorse energetiche, che invece avremmo dovuto 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

Immagine che contiene testo

Descrizione generata automaticamente

O(m)

O(n)

O(c)

O(c)

O(c)

O(c)

## 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*[[1]](#footnote-1), 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*:

Immagine che contiene testo

Descrizione generata automaticamente

* Complessità di *McCabe* nell’implementazione dell’*edge-node*:

Immagine che contiene testo

Descrizione generata automaticamente

* Complessità di *McCabe* nell’implementazione dell’*App Server*:

Immagine che contiene testo

Descrizione generata automaticamente

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 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 generato da *Radon* è di grado A con una complessità intorno a 2 su 61.



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.

# Manuale d’installazione e d’uso

L’installazione di tutte le componenti necessarie per il corretto funzionamento del codice richiede una lunga serie di passaggi che devono essere svolti principalmente da linea di comando.

I primi passaggi di questa installazione verranno svolti su Ubuntu, un programma scaricabile da “Microsoft store” che consiste in un sistema operativo basato su Linux, e, successivamente, ci si sposterà sulla “Windows Subsystem for Linux”.

Requisiti software

Per prima cosa bisogna scaricare “Ubuntu”.

Una volta fatto ciò, si deve avviare l’applicazione e, alla prima apertura, verrà richiesto di creare una partizione Linux tramite un “username” e una “password”.

Questa password sarà fondamentale perché verrà richiesta ogni volta che si deciderà di far ripartire l’application server.

Successivamente bisogna installare WSL 2.0 sulla propria macchina.

***https://www.omgubuntu.co.uk/how-to-install-wsl2-on-windows-10***

Prima di fare ciò, è necessario aprire il “Prompt dei comandi” in modalità “Esegui come amministratore” ed eseguire il comando

* wsl.exe –install

per abilitare le funzioni opzionali di WSL.

Fatto questo, bisognerà riavviare il sistema, aprire Ubuntu ed eseguire i seguenti due comandi

* sudo apt update
* sudo apt upgrade

A questo punto si può passare all’installazione di WSL 2.0. Per eseguire l’upgrade da WSL 1 basta aprire nuovamente il “Promp dei comandi” con i permessi d’amministratore ed eseguire il comando

* wsl.exe --set-version Ubuntu 2

In questo modo si installerà e si setterà di default WSL 2.0 sul proprio computer.

Requisiti per le componenti del ChirpStack LoRaWAN Network Server

***https://www.chirpstack.io/project/install/requirements/***

Per prima cosa bisogna aprire Ubuntu ed eseguire i seguenti 3 comandi

* sudo apt install mosquitto

Questo comando installa Mosquitto, un broker dei messaggi che implementa il protocollo MQTT

* sudo apt install postgresql

Questo comando installa PostgreSQL, un DBMS (DataBase Management System)

* sudo apt install redis-server

Questo commando installa Redis, un archivio di strutture dati in memoria open source usato come database, cache e broker di messaggi.

Installazione application server

***https://www.chirpstack.io/application-server/install/debian/***

A questo punto si può passare all’installazione dei componenti messi a disposizione dallo stack ChirpStack.

I seguenti comandi verranno eseguiti su Ubuntu con i permessi d’amministratore:

* sudo -u postgres psql

Per creare un database per l’application server e spostarsi nel prompt di PostgreSQL

* create role chirpstack\_as with login password ‘dbpassword’

Per creare l’user chirpstack\_as (inserire una password al posto di ‘dbpassword’).

* create database chirpstack\_as with owner chirpstack\_as

Per creare il database chirpstack\_as

* \c chirpstack\_as
* create extension pg\_trgm
* create extension hstore
* \q

Per verificare il corretto setup, si può eseguire il comando

* psql -h localhost -U chirpstack\_as -W chirpstack\_as

I successivi comandi sono usati per installare ed avviare le varie componenti:

* sudo apt install chirpstack-gateway-bridge

sudo systemctl start chirpstack-gateway-bridge

* sudo apt install chirpstack-network-server

sudo systemctl start chirpstack-network-server

* sudo apt install chirpstack-application-server

sudo systemctl start chirpstack-application-server

“start” serve per avviare la particolare componente. Se al suo posto si scrive “restart” o “stop” o “status”, il comando, rispettivamente, esegue un riavvio o termina l’esecuzione o fornisce lo stato del componente.

I seguenti comandi non servono per l’installazione delle componenti, ma permettono di visualizzare a video l’output di quella particolare componente:

* journalctl -u chirpstack-gateway-bridge -f -n 50
* journalctl -u chirpstack-network-server -f -n 50
* journalctl -u chirpstack-application-server -f -n 50

Quando si crea un componente, viene generato il suo corrispondente “Configuration file”. Per potervi accedere bisogna eseguire i comandi che seguono:

* chirpstack-gateway-bridge.toml
* chirpstack-network-server.toml
* chirpstack-application-server.toml

A questo punto si devono eseguire i seguenti tre comandi:

* cd /tmp
* wget --content-disposition \https://gist.githubusercontent.com/djfdyuruiry/6720faa3f9fc59bfdf6284ee1f41f950/raw/952347f805045ba0e6ef7868b18f4a9a8dd2e47a/install-sg.sh chmod +x /tmp/install-sg.sh
* /tmp/install-sg.sh && rm /tmp/install-sg.sh

Fatto ciò si deve aprire il “Prompt dei comandi” tramite “Esegui come amministratore” ed eseguire questi altri tre comandi:

* wsl –shutdown

Per terminare l’esecuzione di Ubuntu

* wsl genie -s
* sudo systemctl status time-sync.target

In questo modo l’interfaccia grafica dell’application server sarà visibile all’indirizzo

* localhost:8080

Una volta spento il computer, per poter visualizzare nuovamente l’application server in locale sarà necessario eseguire il comando

* wsl genie -s

dal “Prompt dei comandi” (avviato sempre con “Esegui come amministratore”), altrimenti cercando direttamente “localhost:8080” il motore di ricerca dirà che a tale indirizzo non è possibile trovare nulla.

Piccola nota su questo comando: se dovessero vedersi a video una serie di “!” in numero sempre crescente, vuol dire che il comando non funzionerà.

Il trucco sta nell’interrompere l’esecuzione del comando (tramite la combinazione di comandi “CTRL” + ”c”) ed eseguirlo nuovamente.

Configurazione dell’interfaccia grafica

Una volta inseriti “username” e “password” ed entrati nell’application server, bisogna passare alla configurazione della rete.

Organization

Applications

Network-servers

Gateway-profiles

Gateway

Device-profile

Device in applications

Modifiche da apportare al codice sorgente

Configurato il tutto, se si prova a far girare il codice sorgente esso non funzionerà. Infatti bisogna apportare delle modifiche al codice ogni qual volta che si esegue il comando per avviare l’application server.

Innanzitutto bisogna accedere alla macchina virtuale che gira si WSL 2.0 dal codice: questo creerà la connessione tra le componenti del codice e quelle create da linea di comando.

***https://www.youtube.com/watch?v=yCK3easuYm4&ab\_channel=DavidBombal***

Su Ubuntu eseguire i seguenti comandi

* wsl -l -v

Per vedere la verisone id Ubuntu running

* ip addr | grep eth0

Per prelevare l’indirizzo ip della virtual machine (indirizzo ip dopo la voce “inet”)

Per vedere l’indirizzo ip di Windows, eseguire su “Prompt dei comandi”

* ipconfig

e guardare sotto la sezione “Scheda LAN wireless Wi-Fi indirizzo IPv4”.

Arrivati a questo punto bisogna consentire la connessione da Ubuntu a Windows. Questo viene fatto attraverso due passaggi.

Il primo consiste nel creare una nuova regola sul firewall per gestire la connessione di dispositivi diversi alla macchina virtuale Ubuntu. Infatti, a questo punto, l’unico dispositivo che conosce della Virtual Machine che gira su Ubuntu è il computer sul quale Ubuntu è scaricato. Bisogna rendere quella Virtual Machine visibile anche ad altri dispositivi. Per fare ciò bisogna seguire questi passaggi:

Pannello di controllo --> Rete internet --> Centro connessioni di rete --> Windows defender firewall -> impostazioni avanzate --> Regole connessioni in entrata --> Nuova regola (con tasto dx) --> Selezionare porta, avanti --> mettere porta 3390(o una qualsiasi libera) --> avanti fino alla fine

Il secondo passo, invece, deve essere eseguito da “Windows PowerShell” dotato dei permessi d’amministratore. Bisogna eseguire il comando

* netsh interface portproxy add v4tov4 listenport=3390 listenaddress=0.0.0.0 connectport=’port’ connectaddress=’ubuntu\_addr’

Dove bisogna sostituire ‘port’ con il numero di porta che si è inserito al passo precedente (es.: 3390) e ‘ubuntu\_addr’ con l’indirizzo della Virtual Machine.

Questi passi andranno eseguiti tutti una sola volta, tranne quello che consente di prelevare l’indirizzo della Virtual Machine da Ubuntu: tale indirizzo, infatti, andrà poi inserito nel codice in un determinato punto.

Successivamente, bisognerà prelevare dall’interfaccia web tre chiavi

* Application Session Key (OTAA)
* Application Session Key (Activation)
* Network Session Key (Activation)

Ed inserirle nel codice sorgente.

Se tutti questi passaggi sono stati fatti correttamente, il codice comunica con l’interfaccia grafica dell’application server e sarà possibile vedere i messaggi.

1. https://pypi.org/project/radon/ [↑](#footnote-ref-1)