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](#_Toc96695007)

[Obiettivo 6](#_Toc96695008)

[W3C 7](#_Toc96695009)

[LoRaWan 8](#_Toc96695010)

[MAPE-K loop 10](#_Toc96695011)

[MQTT 11](#_Toc96695012)

[ChirpStack 13](#_Toc96695013)

[Iterazione 0 15](#_Toc96695014)

[Analisi dei requisiti 16](#_Toc96695015)

[R0 – Azioni di riconfigurazione per allungamento del ciclo di vita delle sentinelle: 16](#_Toc96695016)

[R1 – Azioni di auto-diagnosi manutentiva di risoluzione autonoma dei guasti: 16](#_Toc96695017)

[User-Stories 18](#_Toc96695018)

[Lato edge device: 18](#_Toc96695019)

[Lato application server: 19](#_Toc96695020)

[Lato watchdog: 20](#_Toc96695021)

[Deployment Diagram 21](#_Toc96695022)

[Deployment Diagram Informale 21](#_Toc96695023)

[Deployment Diagram UML 22](#_Toc96695024)

[Iterazione 1 23](#_Toc96695025)

[Component diagram 23](#_Toc96695026)

[Class diagram 24](#_Toc96695027)

[Iterazione 2 25](#_Toc96695028)

[Component diagram 25](#_Toc96695029)

[Class diagram 26](#_Toc96695030)

[Iterazione 3 27](#_Toc96695031)

[Component diagram 27](#_Toc96695032)

[Class diagram 29](#_Toc96695033)

[State machine watchdog 30](#_Toc96695034)

[Iterazione 4 31](#_Toc96695035)

[Component diagram 31](#_Toc96695036)

[Class diagram 33](#_Toc96695037)

[Iterazione 5 34](#_Toc96695038)

[Component diagram 34](#_Toc96695039)

[Class diagram 36](#_Toc96695040)

# 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. Al fine di riuscire nel proprio intento, 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.”*



## 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*



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

*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.



L’obbiettivo è quello di utilizzare questo modello concettuale inserendo un ciclo MAPE-Kper ogni *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:

* Leggero ed efficiente per ridurre al minimo le risorse richieste per il client e la larghezza di banda della rete.
* Consente la comunicazione bidirezionale.
* Consente la trasmissione di messaggi di broadcast a gruppi di client
* Scalabilità.
* Specifica i livelli dei *Quality Of Service* (***QoS***), ad esempio per supportare l’affidabilità della consegna dei messaggi.
* Supporta sessioni persistenti tra dispositivo e server che riduce il tempo richiesto per riconnettere il client al broker su reti inaffidabili
* I messaggi possono essere crittografati 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* vi 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 subsriber 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 component open-source per reti LoRaWAN. Insieme formano una soluzione pronta all’uso che include un’interfaccia web di facile utilizzo per la gestione dei dispositivi e le API per l’integrazione. L’architettura modulare consente l’integrazione all’interno di infrastrutture esistenti. Tutti i componenti sono autorizzati in base alla licenza MIT e possono essere usati a scopi commerciali. Vengono forniti i seguenti componenti:

* ChirpStack Gateway Bridge: gestisce la comunicazione con i gateway LoRaWAN
* ChirpStack Network Server: un’implementazione del server di rete LoRaWAN
* ChirpStack Application Server : un’implementazione di LoRaWAN Application Server

Architettura ChirpStack

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



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 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 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 un server delle applicazioni deve inviare i dati a un dispositivo, il server di rete ChirpStack manterrà questi elementi in coda, finché non sarà in grado di inviare 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, ...

# 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. |

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

Si vuole rendere possibile la rilevazione automatica di guasti sulle sentinelle, 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 dei nodi edge. 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) | Il nodo edge tenta ripristinare il corretto funzionamento del watchdog guasto tramite il suo riavvio. |
| S4 | Allerta in caso di guasto ad un nodo watchdog e identificazione esatta delle sue caratteristiche fisiche e geografiche. | Guasto ad un watchdog non recuperabile. | Il nodo edge allerta un operatore per un intervento manuale sul nodo watchdog .guasto indicando le sue caratteristiche. |
| S5 | Rendere il sistema in grado di gestire guasti ai nodi edge in maniera autonoma.  (*self-recovery*) | Guasto di un nodo edge: non riceve i dati inviati dai watchdog e, quindi, non può inoltrarli al server. | * L*’application server* ridistribuisce il traffico di dati che arrivava al nodo edge guasto verso altri nodi edge. * L*’application server* notifica il personale del guasto avvenuto. |
| S6 | Limitare la perdita di informazioni dovute a gusti all’*application server.* | Malfunzionamenti dell’*application server* che quindi non è in grado di ricevere e/o elaborare i dati trasmessi dai gateway. | * I nodi edge devono mantenere i dati ricevuti dai watchdog in un buffer locale finchè il server non riprende la sua operatività. * I nodi edge devono notificare il guasto ad un indirizzo di rete dedicato e ritenuto affidabile. |

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 nodo edge, echo non mandato dal watchdog
* S5: ping mandato dal server e/o da un nodo edge, echo non mandato dal nodo edge destinatario
* S6: ping mandato dai nodi edge, echo non mandato dal 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 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 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 abbiamo optato per un diagramma informale, che non segue alcuna regola di raffigurazione ma aiuta a inquadrare l’idea del progetto.

In un secondo momento abbiamo seguito i principi del Deployment Diagram UML che , oltre a raffigurare la parte hardware del sistema, sottolinea anche il software che gira 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 dove risiedono le parti hardware. L’area di acquisizione contiene sia watchdog che edge device, sappiamo 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 protocolloMQTT che abbiamo espresso come un canale di comunicazione. Le client application sono tutte quelle applicazioni che possono interagire con le server application.



### 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, temperatura, CO2 ecc..), mentre un sensore può interfacciarsi ad uno e un solo microcontrollore. Su entrambi i dispositivi girano dei componenti software embedded, che noi simuleremo 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 da zero (caso estremo) a molti watchdog e li inoltra ad uno e 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 sfrutteremo la tecnica del *digital twin*.
* Cloud Server: questo sottosistema comunica con zero(caso estremo) a molti edge device. Abbiamo deciso di esplodere alcune componenti software indicate da Chirpstack, quali il broker mqtt, il network server, l’application server e l’interfaccia grafica.



# 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 rappresentazione delle componenti si è partito nel rappresentare ad alto livello il sistema con tre macro-componenti che comunicano tra loro mediante delle interfacce che verranno implementate nelle iterazioni successive.



## Class diagram

In questa iterazione il diagramma delle classi mostra tre componenti software ognuno dei quali presenta una lista di metodi che servono solo per indicare le varie funzioni che ciascuno di essi dovrà implementare.

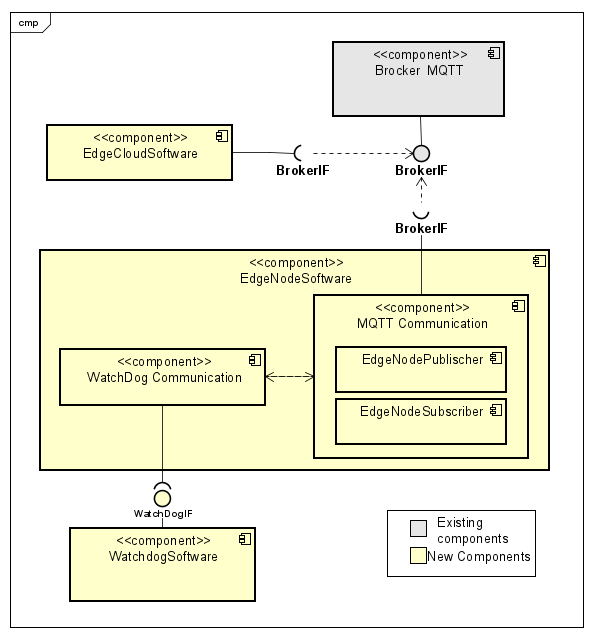


# Iterazione 2

In questa fase del progetto si è passati a dettagliare la componente EdgeNodeSoftware. In particolare si è implementato il funzionamento di un edgenode e se ne è fatta una simulazione.

## 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)

## 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’edgenode 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 edgenode ed implementati i metodi dell’interfaccia EdgeDevice. Questo fa si che un edgenode possa scambiare dati con l’esterno.



# Iterazione 3

In questa iterazione è stato implementato il funzionamento di un watchdog, andando a dettagliare la componente WatchDogSoftware, ed è stata fatta una simulazione dello stesso.

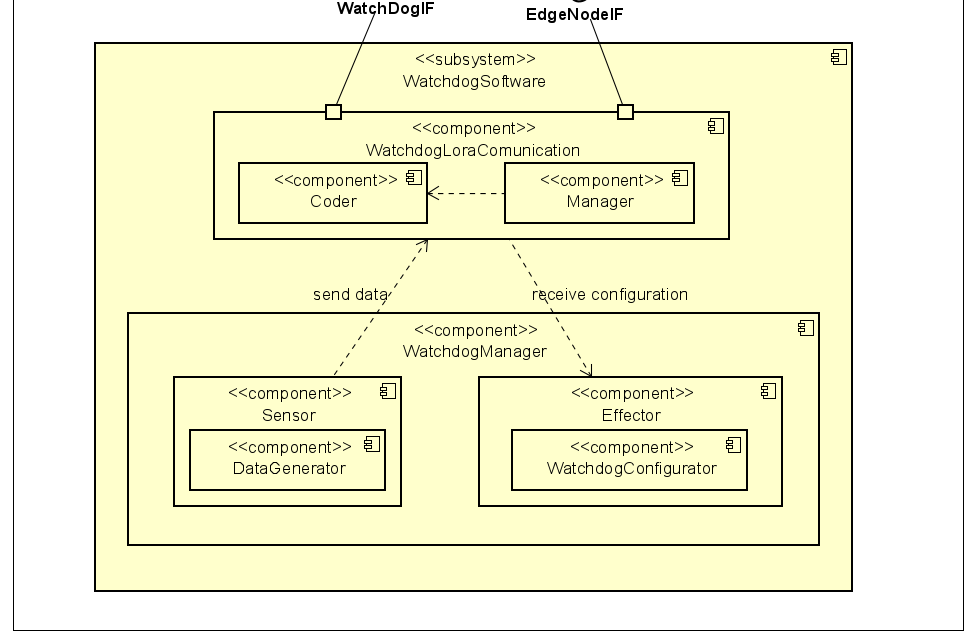
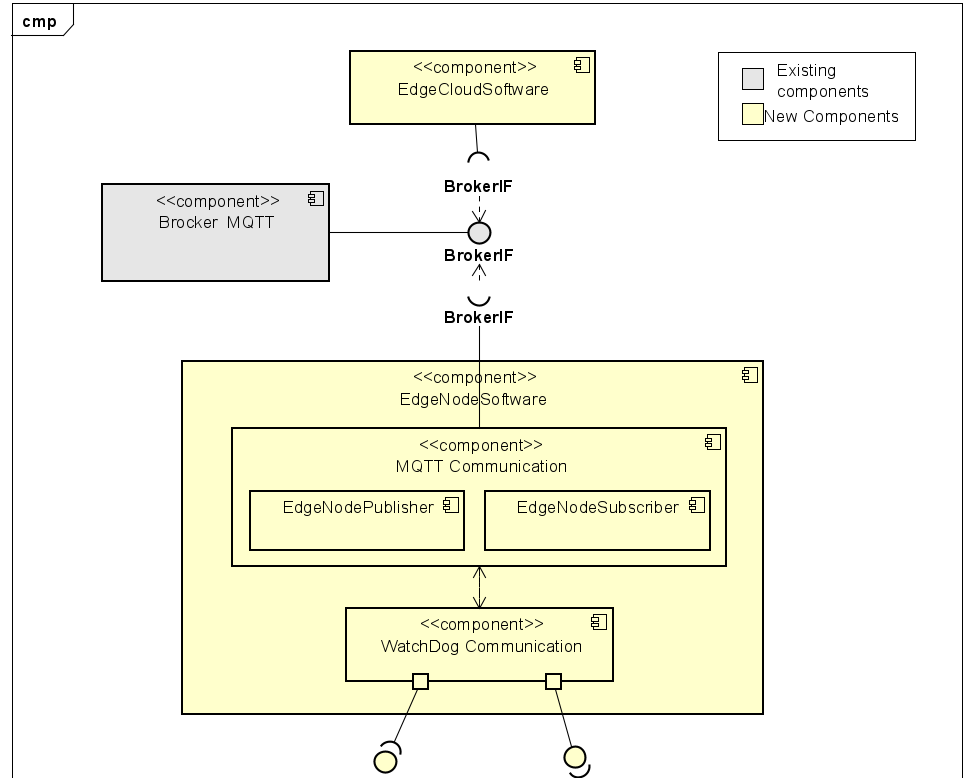
## Component diagram

La componente WatchdogSoftware deve essere in grado di comunicare con l’edgedevice 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 congruo al protocollo LoRaWAN
* Una componente Manager

All’interno di WatchdogSoftware vi è poi un’altra sottocomponente, chiamata WatchdogManager: si tratta di una componente che si occupa della gestione del watchdog mediante due ulteriori sottocomponenti:

* Sensor, il quale ha il compito di campionare i dati ambientali (nel caso di questo progetto i dati verranno simulati attraverso la componente DataGenerator)
* Effector, il quale si occupa del settaggio della configurazione del watchdog mediante la componente WatchdogConfigurator (frequenza di campionamento, potenza di trasmissione,…)



## Class diagram

Come fatto per l’EdgeNodeSoftware, anche per il componente WatchDogSoftware è stato esploso in:

* Un’interfaccia WatchDog nella quale sono presenti tutti i metodi esposti dal watchdog relativi alla sua comunicazione con gli edgenode 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 si che un watchdog possa scambiare dati con l’edgedevice al quale è connesso.



## State machine watchdog



# Iterazione 4

In questa fase del progetto si è sviluppato il funzionamento dell’AppServer 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.

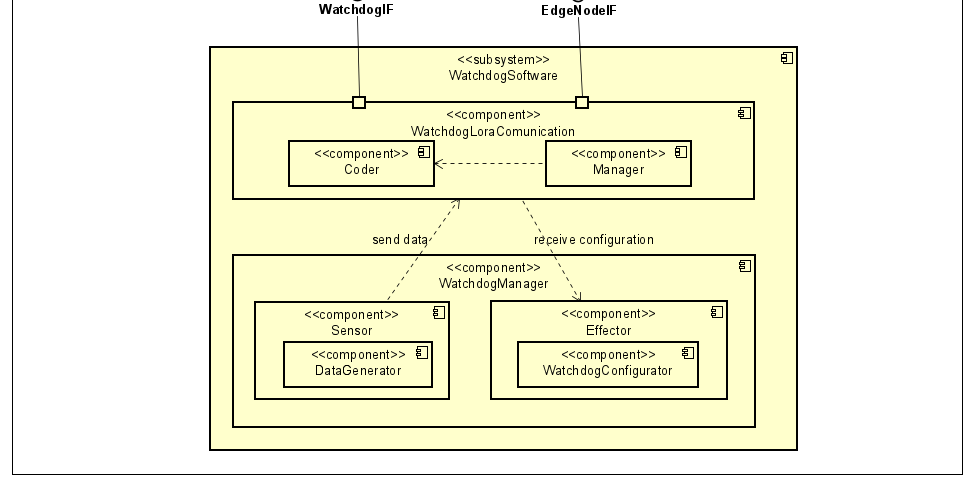
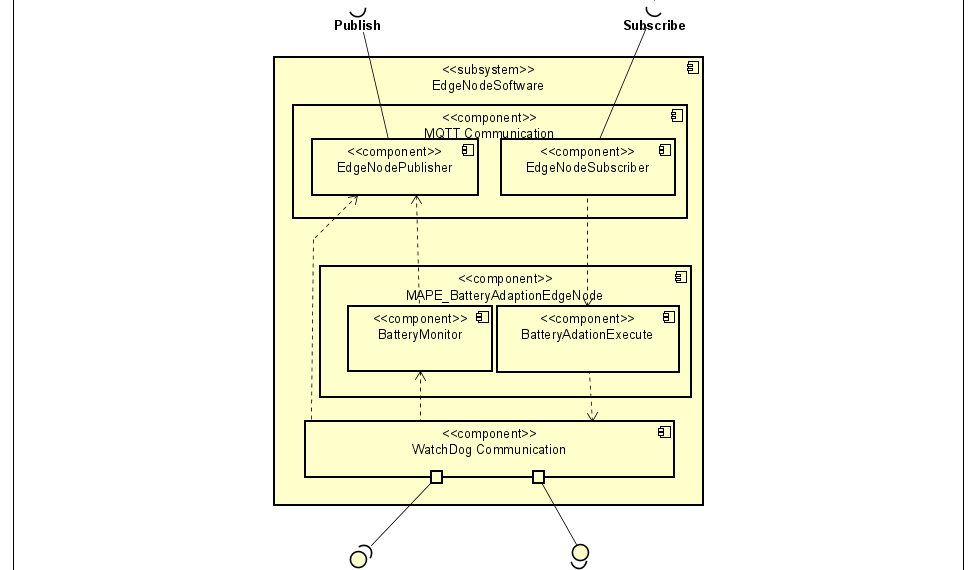
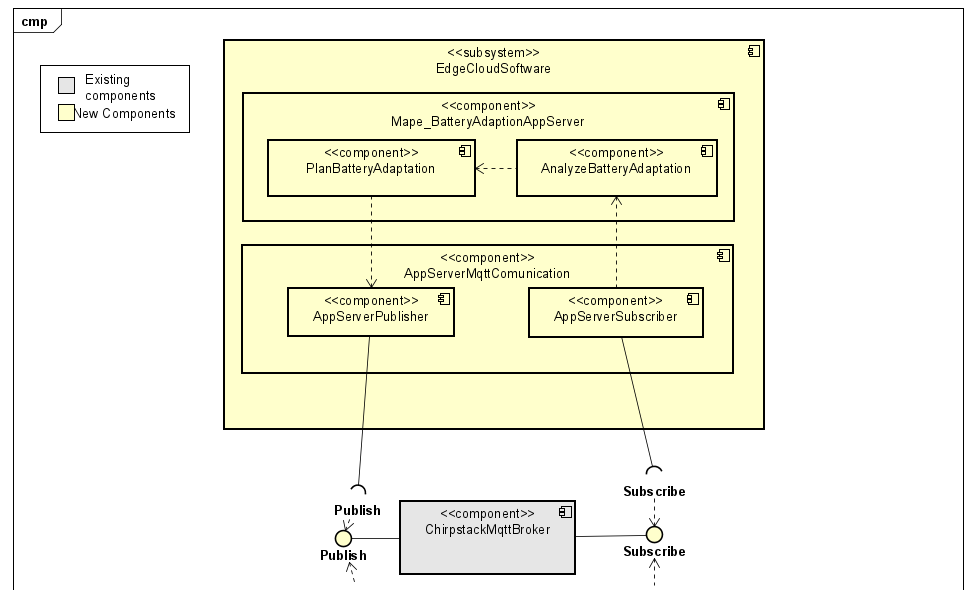
## Component diagram

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

* Il ChirpstackMQTTBroker, il quale comunica con l’AppServer e gli edgenode 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 2 sottocomponenti:
  + AppServerMQTTCommunication serve per la comunicazione tra AppServer 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 tramite le sottocomponenti AnalizeBatteryAdaptation e PlanBatteryAdaptation

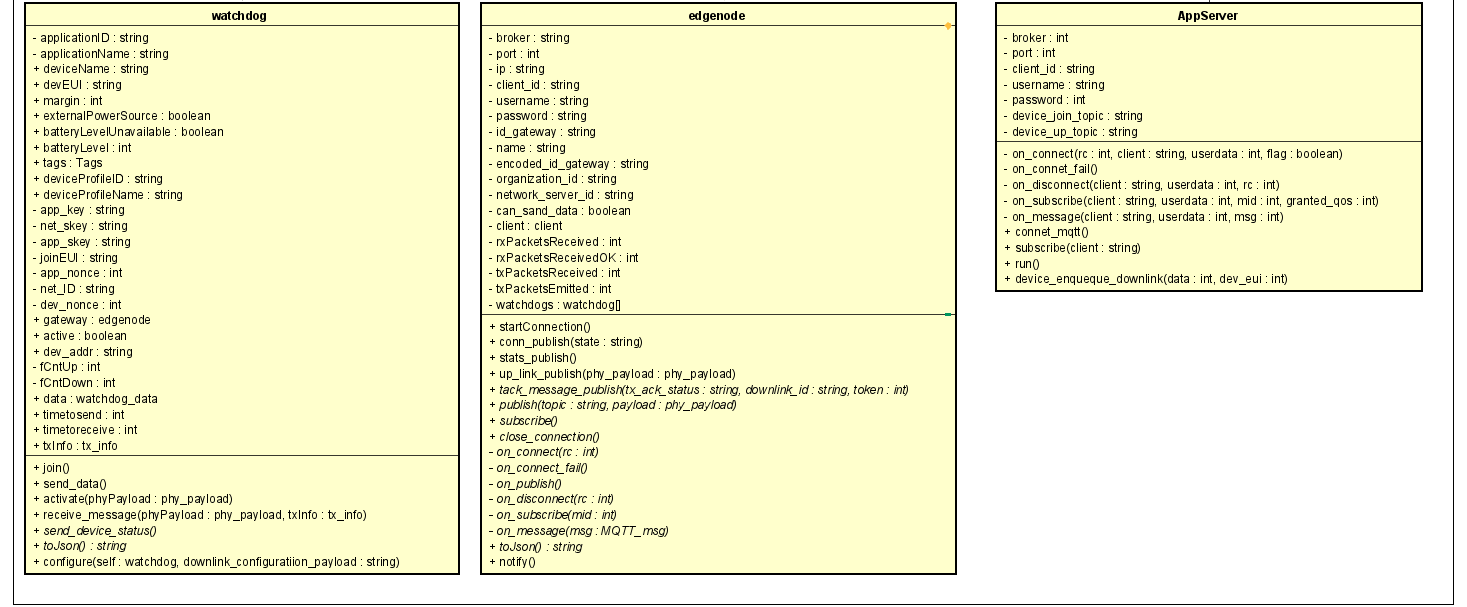
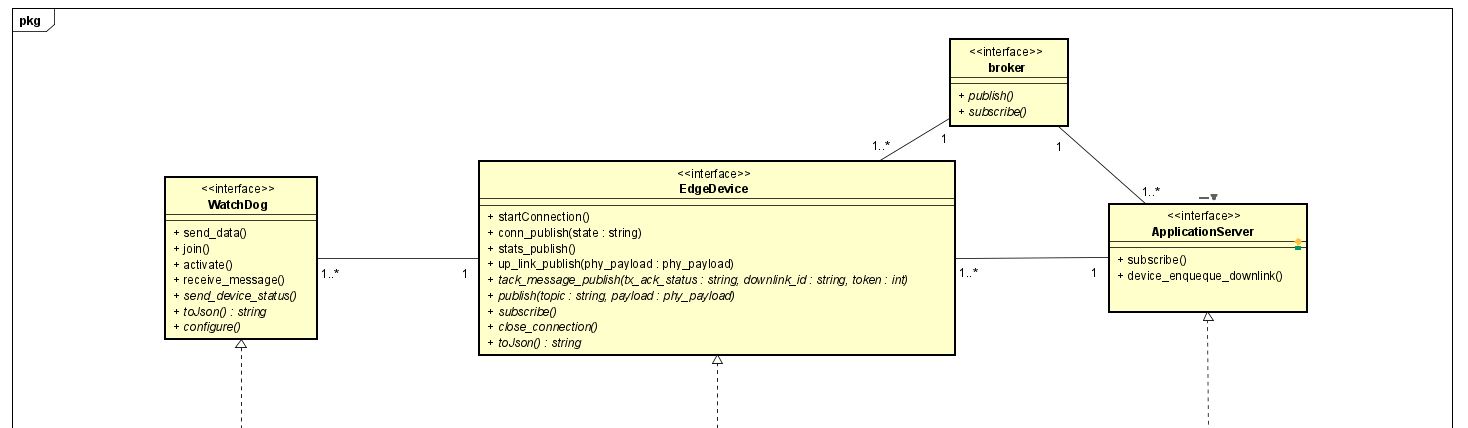


## Class diagram

Come fatto per l’edgenode 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

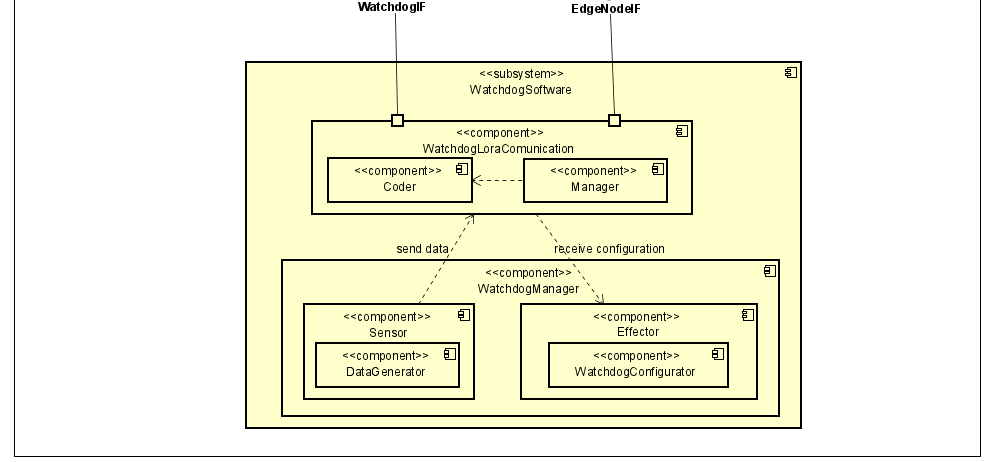
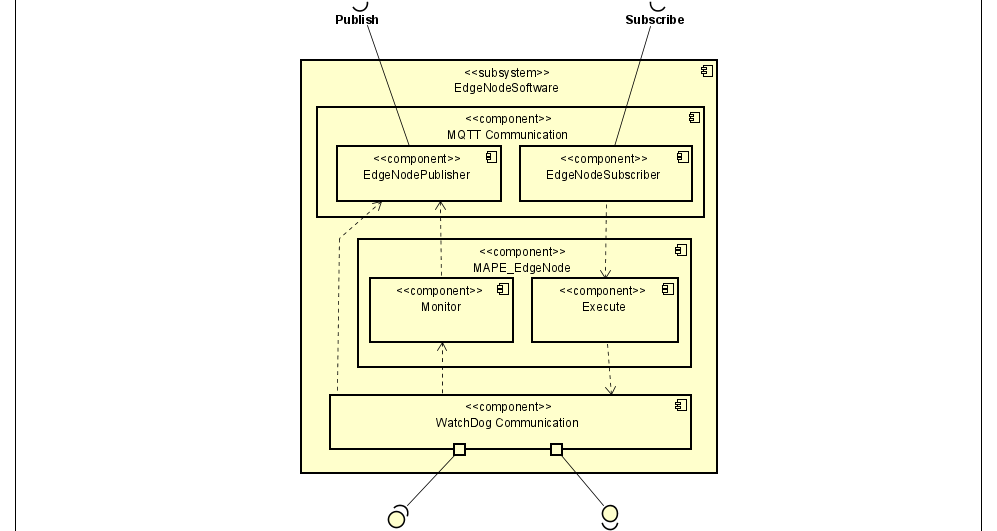
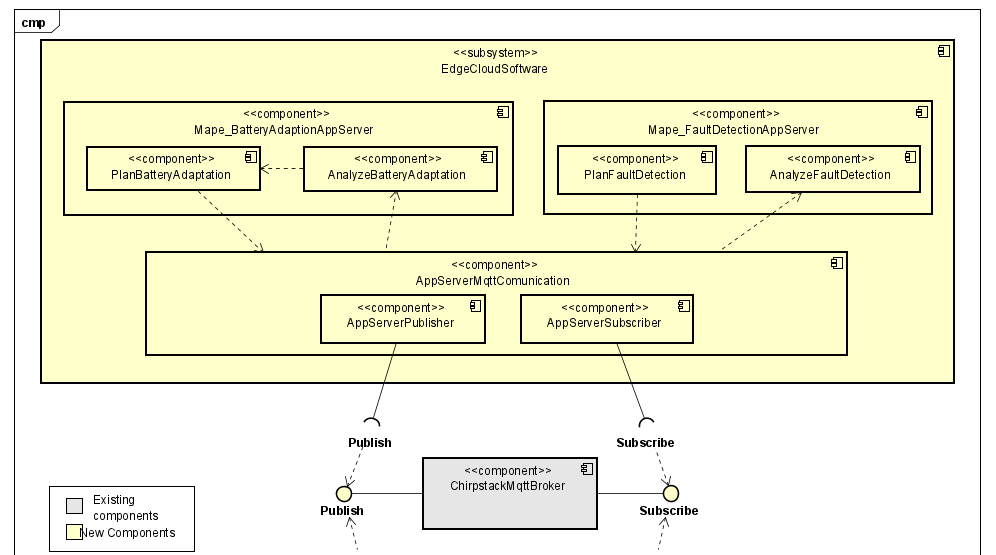
Quest’ultima iterazione del progetto ha l’obiettivo di implementare il requisito R1. Per fare ciò è stato necessario dettagliare ulteriormente la componente *EdgeCloudSoftware* ed introdurre un ulteriore **MAPE-K** **loop** per gestire il goal di auto-diagnosi dei guasti.

## Component diagram

La struttura di questo diagramma delle componenti 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 alla gestione dei guasti.

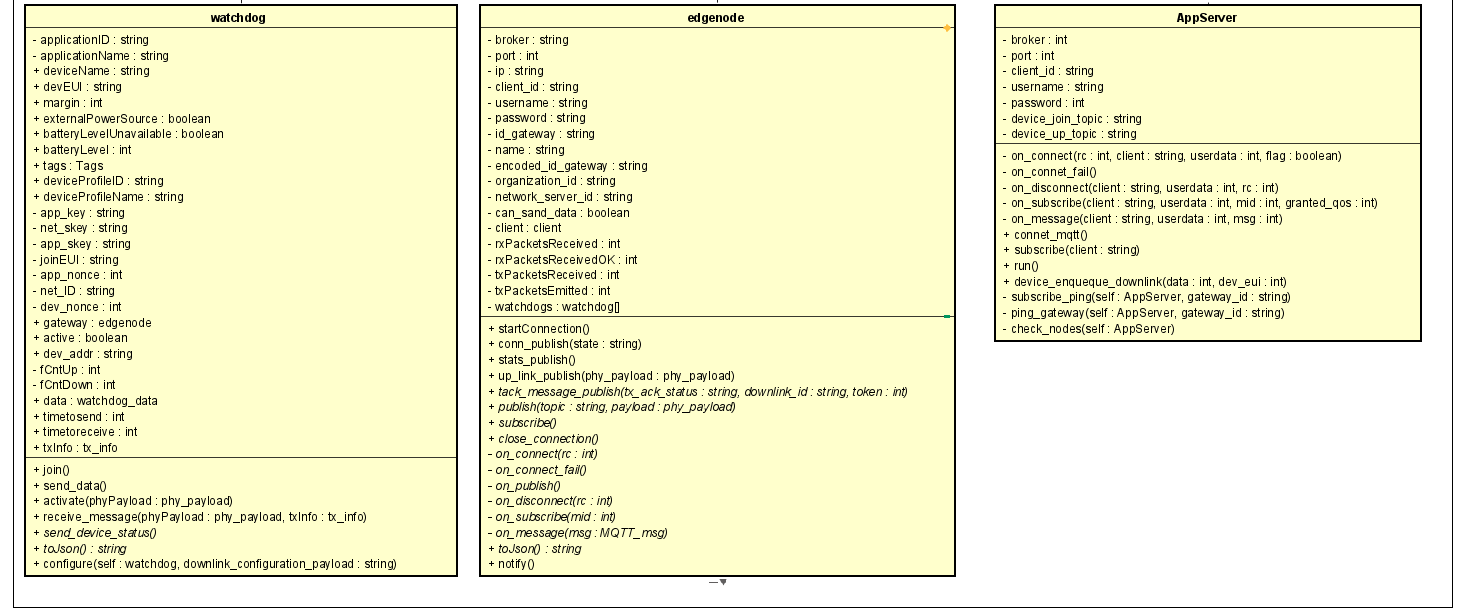
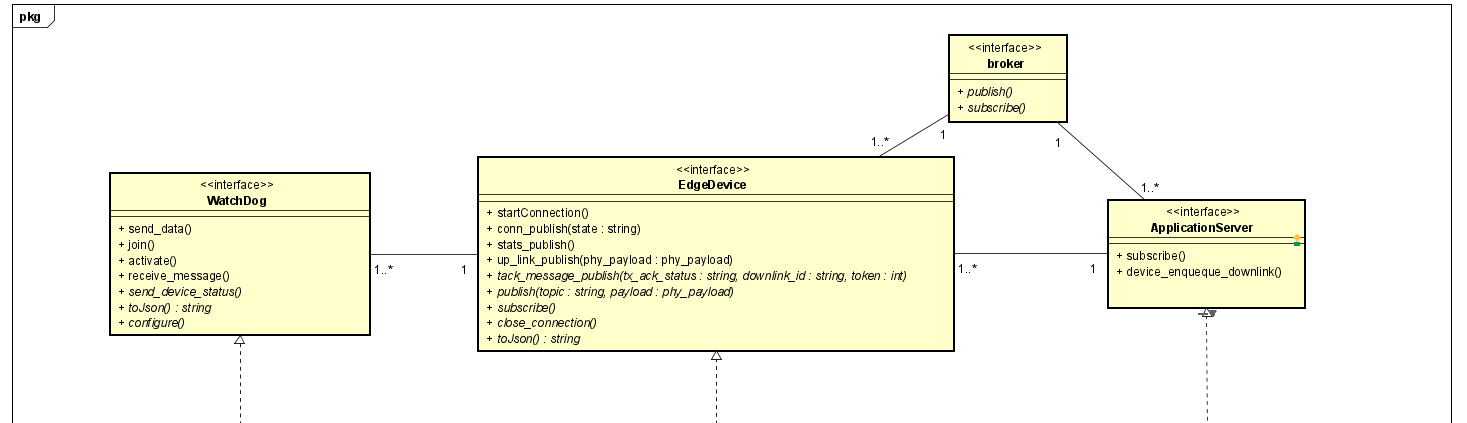
Nell’EdgeNodeSoftware, invece, si sono mantenute le componenti di Monitor e di Execute introdotte nell’iterazione 4.



## Class diagram

La struttura è identica a quella del diagramma delle classi 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 si sfrutta 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.

[Inserire pseudo codice prima e rifare screen dell’implementazione(ci sono state modifiche)]

Immagine che contiene testo

Descrizione generata automaticamente

O(m)

O(n)

O(c)

O(c)

O(c)

O(c)

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)*