Università del Piemonte Orientale

Dipartimento di Scienze e Innovazione Tecnologica

Corso di Laurea Magistrale in Intelligenza Artificiale e Innovazione Digitale

Relazione per la prova finale

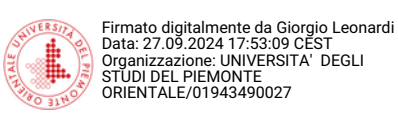
Progettazione e implementazione di architettura cloud basata su micro-servizi, per applicazioni IoT in ambiente industriale

Tutore interno:

Prof. Giorgio Leonardi

Candidato:

Riccardo Francia



Anno Accademico 2023/24

Sommario

Abstract

Il progetto svolto consiste nella creazione di un sistema basato su micro-servizi distribuiti in rete che permettono di eseguire da remoto dei *test* sulle performance di celle coibentate. Il sistema supporta la connessione simultanea di più centraline di controllo e permette di archiviare e rendere disponibili i risultati ottenuti.

1 Introduzione

Il progetto svolto consiste nella creazione di un sistema basato su micro-servizi distribuiti in rete che permettono di eseguire da remoto dei *test* sulle performance di celle coibentate.

Il sistema *CloudREINEU* è stato sviluppato sfruttando tecnologie che derivano dall'ambito dell'*IoT*, portando alla creazione di servizi di rete che integrano il funzionamento delle centraline di controllo, ampliando le funzionalità offerte dai prototipi sviluppati prima del mio ingresso nel progetto.

Il sistema di controllo *CloudREINEU* offre il vantaggio di poter tenere traccia di più centraline da un unico portale, eliminando il vincolo di avere un operatore specializzato in ogni sede in cui viene fatta operare una centralina di controllo. Per rendere una centralina controllabile da remoto diventa necessario disporre solamente di una connessione di rete.

L'utilizzo del protocollo *MQTT* per lo scambio di messaggi è risultata una scelta efficace poiché, per via del paradigma di comunicazione offerto e dal modesto utilizzo di risorse di rete, permette la comunicazione anche nei casi in cui la rete risulta essere lenta o instabile.

Un secondo vantaggio offerto da questo approccio è la possibilità di poter usare in fase di produzione delle centraline, dei microprocessori a bassi consumi e ad alta efficienza poiché la parte di elaborazione dei dati viene delegata a un processo attivo in rete.

Il lavoro svolto è incasellato all'interno del progetto REINEU (REfrigerazione INtelligente EUtettica ibrida), un progetto a più ampio respiro che riguarda la mobilità sostenibile al quale hanno partecipato l'azienda Cold Car, l'azienda Capetti Elettronica e l'Università del Piemonte Orientale. Il lavoro che ho svolto parte da una borsa di studio della durata di 12 mesi e vede il suo perfezionamento durante il completamento del percorso di Laurea Magistrale.

2 Prototipo di centralina di controllo

2.1 Introduzione

Per lo sviluppo del progetto ***CloudREINEU*** è stato necessario poter operare in un ambiente quanto più simile all'ambiente di utilizzo finale del sistema. A tal proposito, mi è stato fornito uno dei prototipi di centralina di controllo realizzati per il progetto **REINEU** che prendono il nome provvisorio *"ktest"*. La serie di prototipi è stata sviluppata dal Professor Attilio Giordana, in un periodo di tempo precedente al mio ingresso nel progetto.

Il prototipo in mia dotazione è basato su una scheda di sviluppo *BeagleBone*. Contiene lo stesso software che si può trovare all'intero della centralina fornita in dotazione all'azienda con la quale abbiamo collaborato. La differenza sostanziale che presenta il modello in mio possesso, rispetto agli altri prototipi, è che vengono simulati tramite l'utilizzo di uno *script* gli strumenti e gli attuatori che sono connessi alla centralina completa, rendendo così virtualmente identiche le due centraline.

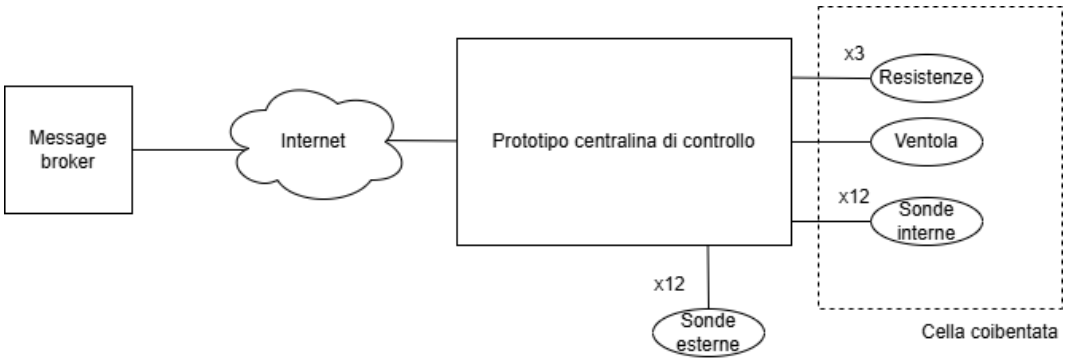


Figura 1. Schema del prototipo

Lo schema generale della centralina di controllo prevede che ai pin *GPIO* del prototipo siano collegati rispettivamente: 24 sensori di temperatura per monitorare le temperature interne e esterne alla cella coibentata; 3 attuatori per azionare le resistenze riscaldanti; 1 attuatore per azionare le ventole; 1 wattametro per misurare la potenza istantanea assorbita dalle stufe.

2.1.1 Stati di funzionamento di una centralina di controllo

La centralina può assumere tre stati:

- "ON": la centralina è connessa al sistema ed è in esecuzione un *test* delle performance della cella eseguito localmente
- "OFF": la centralina è connessa al sistema ma non è in esecuzione un *test* delle performance della cella.
- "PASSIVE": la centralina è connessa al sistema ed è possibile eseguire un *test* delle performance della cella in *cloud*, tramite il sistema ***CloudREINEU***.

2.1.2 Moduli in esecuzione su una centralina di controllo

Il codice in esecuzione sul prototipo di centralina è organizzato in più moduli indipendenti sviluppati nel linguaggio *C++*. La comunicazione tra i moduli avviene tramite lo scambio di messaggi *MQTT* sfruttando l'istanza del *message broker Mosquitto* installato. Il prototipo prevede la possibilità di connettersi a una rete esterna per poter inviare una copia dei messaggi *MQTT* su un *message broker* esterno.

I moduli in esecuzione più significativi sono:

- ***meters*:** rileva le temperature interne e esterne alla cella tramite i sensori connessi e calcola la potenza istantanea assorbita dagli elementi riscaldanti.
- ***kmeters:*** calcola la media delle temperature interne e esterne, il coefficiente di dispersione termica e il consumo di energia.
- ***heater*:** si occupa di mantenere costante la temperatura della cella coibentata tramite l'azionamento e lo spegnimento degli elementi riscaldanti e della ventola.
- ***archiver***: archivia in *file* di testo i messaggi contenenti le rilevazioni dei sensori durante le sessioni di *test* della performance della cella.

Il modulo *meters* è sempre attivo all’interno del prototipo, il modulo *kmeters*, *heater* e *archiver* vengono avviati in corrispondenza di una nuova sessione di *test*.

In aggiunta, sono presenti altri moduli che contengono il codice necessario per gestire fisicamente gli elementi connessi ai pin GPIO.

La centralina offre l’accesso a una pagina *web* interna che permette di monitorare i valori istantanei rilevati dai sensori connessi, gestire le sessioni di *test* potendo avviare o arrestare una nuova sessione. Permette, inoltre, di navigare all'interno dell’archivio che contiene le sessioni precedenti per avere una rappresentazione grafica dei dati e un calcolo su una finestra temporale a piacere del coefficiente di dispersione termica e degli altri parametri monitorati.

3 Sistema *CloudREINEU*

3.1 Metodologie e strumenti utilizzati

3.1.1 *Spring framework*

Java Spring Framework [1] è un *framework open source* che permette la creazione di applicazioni *standalone* eseguite sulla *Java Virtual Machine* (JVM).

Spring offre un supporto integrato per *data binding*, conversione tra tipi, validazione, gestione delle eccezioni, gestione delle risorse e degli eventi, permettendo lo sviluppo di applicazioni in grado di sopportare un uso intensivo in ambienti commerciali o industriali. Il *framework* consente di usare la tecnica della *dependency injection* permettendo la creazione di applicazioni modulari con componenti debolmente legati, ideali per micro-servizi e applicazioni distribuite in rete.

3.1.2 *Dependency injection*

La *dependency injection* [2] è un modello di programmazione che permette di delegare la risoluzione delle dipendenze di classi e oggetti di un progetto a un componente esterno chiamato “iniettore”.

Secondo il funzionamento del *pattern* la prima volta che si tenta di risolvere una dipendenza, successivamente alla risoluzione di essa, l’iniettore si prenderà carico di salvare il componente dipendente in un contenitore di istanze, rendendolo così facilmente accessibile a tutti gli altri componenti del progetto che ne faranno richiesta successivamente.

L’intento della tecnica della *dependency injection* è quello di attuare una separazione delle competenze (*SoC*) tra la costruzione e l’uso di un oggetto, aumentando la leggibilità e la riutilizzabilità del codice.

3.1.3 *Spring Boot*

Java Spring Boot [3] è una soluzione del tipo *convention over configuration* offerta dal *framework Spring* che permette di facilitare lo sviluppo di applicazioni e micro-servizi mediante tre funzionalità di base:

- **Autoconfigurazione**: le applicazioni vengono inizializzate con una serie di dipendenze prestabilite che non necessitano di essere configurate manualmente. Questa caratteristica permette di velocizzare lo sviluppo di applicazioni riducendo la possibilità di errore umano.
- **Approccio "opinionated"**: questo tipo di approccio definito "saccente" viene usato da *Spring Boot* per scegliere quali dipendenze iniziali necessita un progetto. Alla creazione del progetto, tramite lo strumento *Spring Initializr*, viene chiesto all'utente di compilare un *form* scegliendo da una lista le funzionalità aggiuntive desiderate, chiamate *Spring Starters*, occupandosi successivamente di integrarle senza che sia richiesto all'utente di installare manualmente nessun *package*.
- **Applicazioni autonome**: tramite l'integrazione di un *web server* come *Tomcat*, *Spring Boot* permette la creazione di applicazioni autonome che possono essere eseguite senza fare affidamento a un *web server* esterno.

3.1.4 *Java* 8

Java [4] è un linguaggio di programmazione *object-oriented* ad alto livello pensato per la portabilità. Il codice *Java*, anziché essere direttamente compilato nel linguaggio macchina specifico per l'architettura sul quale deve essere eseguito, viene tradotto in un linguaggio intermedio chiamato *Java bytecode*. Le istruzioni in *Java bytecode* sono pensate per essere eseguite dalla *Java Virtual Machine (JVM)* scritta specificamente per l'*hardware* su cui deve essere eseguito il codice.

Questo livello di astrazione permette lo sviluppo di codice potenzialmente eseguibile su tutte le piattaforme che supportano *Java* senza bisogno di ricompilare il codice. Il beneficio della portabilità porta con sé un *overhead* che incide sul costo di esecuzione del codice, specialmente se si compara *Java* a un altro linguaggio di programmazione compilato direttamente in linguaggio macchina.

3.1.5 *RabbitMQ*

RabbitMQ [5] è un *broker* di messaggistica che implementa il protocollo *AMQP* scritto nel linguaggio *Erlang*, le sue librerie sono disponibili per diversi linguaggi di programmazione.

Un *broker* di messaggistica (*message broker*) è un programma intermedio (*middleware*) usato per mediare la comunicazione tra applicazioni. Agisce riducendo al minimo la consapevolezza reciproca tra i *client*, traducendo i messaggi dal protocollo di messaggistica formale del mittente al protocollo di messaggistica formale usato dal ricevitore.

RabbitMQ [6] offre un modello architetturale per la convalida, la trasformazione e l'instradamento di messaggi, integra nativamente il protocollo *AMQP* e offre un supporto per altri protocolli di comunicazione, tra i quali compaiono *MQTT, HTTP, STOMP e websocket*.

3.1.6 *AMQP 0-9-1*

AMQP 0-9-1 [7] (*Advanced Message Queuing Protocol*) è un protocollo di messaggistica che permette la comunicazione tra applicazioni *client* e *broke*r. *AMQP* è un protocollo di rete, i *publisher*, i *conusmer* e i *broker* possono risiedere su macchine differenti.

Secondo le specifiche, i messaggi inoltrati dai processi chiamati *publisher* vengono in primo luogo recapitati negli *exchange*, componenti del *broker* di messaggistica comparabili a degli uffici postali. Ricevuti i messaggi, gli *exchange* a loro volta distribuiscono una copia dei messaggi su una o più delle strutture

chiamate *queues* (code), usando delle precise regole di instradamento chiamate *binding*. Infine, i *broker* di messaggistica si occupano di inoltrare i messaggi ai processi *consumer* che si sono sottoscritti a una determinata coda in attesa di consumare il messaggio.

Le reti sono inaffidabili e le applicazioni possono fallire nel processare i messaggi, per ovviare a questo problema il modello *AMQP 0-9-1* implementa una funzionalità di *message acknowledgements (ACK)*. Quando questa tecnica è in uso, un *broker* rimuoverà completamente un messaggio da una coda solo quando riceverà una notifica, da parte del consumer, che ne indica la completa e corretta ricezione. Nel caso in cui risulti impossibile inoltrare un messaggio, quest'ultimo può essere restituito al mittente, eliminato oppure inserito nelle cosiddette "*dead letter queue*", permettendone una successiva lettura.

AMQP 0-9-1 è un protocollo programmabile, le varie entità e gli schemi di instradamento solo definiti direttamente dalle applicazioni e non dagli amministratori dei *broker*.

3.1.7 *Queue*

Le *queue* (code) in AMQP 0-9-1 sono delle entità che si trovano all'interno di un *broker*, hanno lo scopo di contenere i messaggi che verranno consumati dalle applicazioni. Una coda può avere degli attributi come un nome, la durata e l'esclusività.

Una coda, prima di essere usata, deve essere dichiarata dall'applicazione che ne fa uso. La dichiarazione comporta la creazione di una coda nel *broker*, se viene dichiarata una coda già esistente la dichiarazione non avrà effetto.

3.1.8 *Routing key*

La *routing key* (chiave di instradamento) è un attributo usato da alcuni *exchange* come filtro per instradare i messaggi su determinate code. Viene impostata durante un'operazione di *binding* (legame) tra un *exchange* e una o più code.

3.1.9 *Exchange*

Gli *exchange* sono delle entità *AMQP 0-9-1* verso le quali sono inviati i messaggi. Un *exchange* riceve un messaggio e lo instrada verso zero o più code. L'algoritmo di instradamento dipende dal tipo di *exchange* e dalle regole, chiamate *binding*. AMQP 0-9-1 prevede quattro tipi di *exchange*:

- **Direct:** inoltra i messaggi alle code basandosi sulla *routing key* del messaggio. Questo tipo di *exchange* è ideale per un instradamento di messaggi del tipo *unicast*. Gli *exchange* di tipo *direct* vengono utilizzati per distribuire i compiti tra più processi collegati alla stessa coda, permettendo un bilanciamento del carico con un algoritmo *round robin*.
- **Default:** è un exchange di tipo diretto che non ha un nome ed è dichiarato preventivamente dal *broker*. La proprietà che lo rende particolare è quella per cui ogni coda è automaticamente legata ad esso con una *routing key* che equivale al nome della coda stessa.
- **Fanout:** inoltra i messaggi a tutte le code che sono collegate ad esso, ignorando la *routing key*. Questo tipo di *exchange* è ideale per un invio di messaggi del tipo *broadcast*.
- **Topic:** instrada i messaggi a una o più code basandosi su una corrispondenza tra la *routing key* e il *pattern* che è stato usato quando si è fatta un'operazione di *bind* tra la coda e *l'exchange*. Questo tipo di *exchange* viene usato per un instradamento *multicast.*
- **Header:** progettato per instradare basandosi su diversi attributi che sono facilmente espressi da un *header* anziché una *routing key*. Possono essere usati come *exchange* di tipo *direct* quando la *routing key* non è espressa come una stringa di testo ma come un valore di tipo intero oppure un oggetto di tipo dizionario.

3.1.10 *Eclipse Mosquitto*

Eclipse Mosquitto [8] è un *broker* di messaggistica *open source* che implementa il protocollo *MQTT*.

Mosquitto è pensato per essere leggero e adatto specialmente a dispositivi a bassa potenza. Oltre ad accettare le connessioni da *client MQTT*, *Mosquitto* ha una funzione di *bridge* [9] che permette la connessione tra più *server MQTT*. Questa funzione permette la creazione di reti di *server* MQTT consentendo ai messaggi di fluire da una parte all'altra della rete creata.

3.1.11 *MQTT*

MQTT [10] è un protocollo di messaggistica che implementa il paradigma *client/server*, è progettato per essere leggero, aperto e semplice da implementare. Queste caratteristiche lo rendono ideale per l'utilizzo nell'ambito dell'*IoT* e tutti i contesti nel quale è richiesto avere in impatto contenuto a livello di banda e di risorse. Il protocollo si basa su *TCP/IP* offrendo una connessione bidirezionale, ordinata e senza perdita di dati. Le funzionalità offerte dal protocollo sono le seguenti:

- Utilizzo del modello *publish/subscribe* che permette una distribuzione di messaggi del tipo "uno a molti" e permette di attuare il disaccoppiamento tra le applicazioni.
- Un trasporto di messaggi agnostico nei confronti del contenuto del *payload* del messaggio stesso
- Tre tipi di *QoS (Quality of Service)* [11] per la consegna di messaggi:
    - ***At most once*:** i messaggi sono inviati in base al *best effort* dell'ambiente operativo. Usando questo tipo di invio, è possibile riscontrare una perdita di messaggi; pertanto, può essere usata solo nei casi in cui la perdita di un messaggio non incide sulla funzionalità dell'applicazione.
    - ***At least one*:** la ricezione dei messaggi è garantita attraverso l'uso di ACK. Questo tipo di inoltro non esclude la possibilità di riscontrare dei duplicati nel caso in cui ci sia un ritardo nella

ricezione o una perdita del messaggio di conferma da parte dei processi subscriber.

    - ***Exactly one*****:** viene garantita la ricezione di un messaggio soltanto una volta, senza ripetizioni, attraverso un *handshake* a due vie. Questo tipo di invio può essere utilizzato nei casi in cui una mancata o una doppia ricezione possono alterare il corretto funzionamento del sistema.

- Un *overhead* ridotto nel trasporto dei messaggi e uno scambio minimo di messaggi di controllo.
- Un meccanismo di notifica delle parti interessate nel caso in cui avvenga una disconnessione non prevista da parte di un processo.

3.1.11 *Bootstrap*

Bootstrap [13] è un *framework CSS open source* che permette di creare pagine *web* reattive. Fornisce allo sviluppatore librerie *HTML*, *CSS* e *Javascript* per facilitare lo sviluppo del front end di un servizio web, permettendo di creare un'uniformità grafica tra le varie pagine dell'applicativo. Tramite il supporto del *responsive web design* [14], *Bootstrap* permette il regolamento automatico del *layout* delle pagine web tenendo conto delle caratteristiche e della dimensione del dispositivo utilizzato.

3.1.12 *JavaScript*

Javascript [15] è un linguaggio di programmazione multi-paradigma, conforme alle specifiche dettate da *ECMAScript*. È debolmente tipizzato, debolmente orientato agli oggetti, è un linguaggio interpretato e supporta il paradigma di programmazione guidata da eventi, funzionale e imperativa. Gli *script* vengono inseriti nei documenti *HTML* e interagiscono con il *DOM* della pagina. Tutti i *browser* principali sono dotati di un loro motore *Javascript* [16] che esegue il codice senza compilarlo direttamente sul *client*, questo approccio ha il vantaggio di non sovraccaricare il *server* con la presenza di script particolarmente complessi.

3.1.13 *HTTP*

HTTP [17] (*HyperText Transfer Protocol*) è un protocollo a livello applicativo che lavora con un'architettura tipo *client/server*, il *client* si occupa di eseguire una richiesta e il *server* si occupa dell'elaborazione e della restituzione di una risposta al chiamante. Esistono due tipi di messaggi *HTTP*: i messaggi di richiesta e i messaggi di risposta. Questo protocollo differisce da altri per il fatto che le connessioni vengono chiuse una volta che una particolare richiesta è stata soddisfatta.

3.1.14 *Docker*

Docker è una piattaforma *open-source* che consente di automatizzare il *deployment*, la scalabilità e la gestione di applicazioni all'interno di *container*. Un container è un'unità leggera e portatile che include tutto il necessario per eseguire un'applicazione, come codice, librerie, dipendenze e configurazioni.

Sfruttando questa tecnologia, le applicazioni possono girare in ambienti isolati, riducendo conflitti tra diverse versioni di software. Docker semplifica la creazione e la gestione di microservizi, permettendo di scalare facilmente le applicazioni. Poiché i container condividono il kernel del sistema operativo, risultano più leggeri e veloci rispetto alle macchine virtuali tradizionali.

3.2 Scelte progettuali

Per sviluppare il sistema ***CloudREINEU*** è stato scelto l'approccio a micro-servizi, la comunicazione tra servizi viene mediata da un *message broker*. Per la distribuzione e l'esecuzione del sistema si è la piattaforma *Docker*.

Lo sviluppo di un servizio monolitico prevede di strutturare la logica del progetto in modo che tutti i processi siano strettamente legati tra di loro ed eseguiti come un solo servizio, la progettazione di un servizio monolitico risulta quindi concettualmente più semplice ma comporta il rischio di avere un'applicazione meno stabile. Il fallimento di un singolo processo può avere un impatto molto esteso sul funzionamento del servizio, potenzialmente rendendolo inservibile nella sua totalità. Un server monolitico risulta poco scalabile e difficile da mantenere, la modifica o l'aggiunta di nuove funzionalità può creare conflitti con il codice già esistente e quindi obbligare lo sviluppatore a un riadattamento di tutto il processo.

Scomporre l'applicazione in elementi più piccoli e indipendenti ha permesso di ovviare ai problemi descritti precedentemente. Ciascun micro-servizio ha un ruolo ben definito e viene eseguito come un processo a sé stante, può essere sviluppato con una logica che non dipende dalle implementazioni degli altri servizi che si trovano in rete e può essere modificato o scalato senza che il funzionamento globale dell'applicativo ne risenta.

La scelta di *Java* come linguaggio di programmazione è motivata dalla familiarità acquisita durante il percorso di studio e per la vastità delle librerie e *framework open source* disponibili. L'utilizzo del *framework Spring Boot* ha facilitato la creazione e l'integrazione dei servizi di rete, consentendomi di lavorare in un ambiente di sviluppo professionale.

La scelta riguardo l'utilizzo della piattaforma *Docker* ha reso il sistema facilmente distribuibile e scalabile, permettendomi di prendere scelte progettuali indipendenti dal sistema operativo e dall'infrastruttura che avrebbe ospitato il servizio sviluppato.

3.3 Architettura di rete

Il sistema ***CloudREINEU*** è composto da più micro-servizi eseguiti come *container* indipendenti all'interno dell'ambiente di esecuzione offerto da *Docker*.

I micro-servizi che compongono l'architettura sono:

- ***CloudGatewayServer*:** è il punto di ingresso al sistema, espone delle *API RESTful* con le quali è possibile interagire con i componenti del sistema e ospita una *web page* con la quale l'utente può utilizzare il sistema.
- ***CloudArchiver*:** si occupa di memorizzare in un archivio i *logfile* relativi ai dati rilevati durante le sessioni di *test* delle performance eseguiti sulle celle coibentate.
- ***CloudKmeterWrapper*:** permette di gestire l'esecuzione di più istanze contemporanee del micro-servizio *CloudKmeter*.
- ***CloudKmeter*:** si occupa di eseguire da remoto le sessioni di *test* della cella, calcolando la stima del coefficiente di dispersione termica e gestendo il funzionamento degli attuatori collegati alla centralina di controllo.

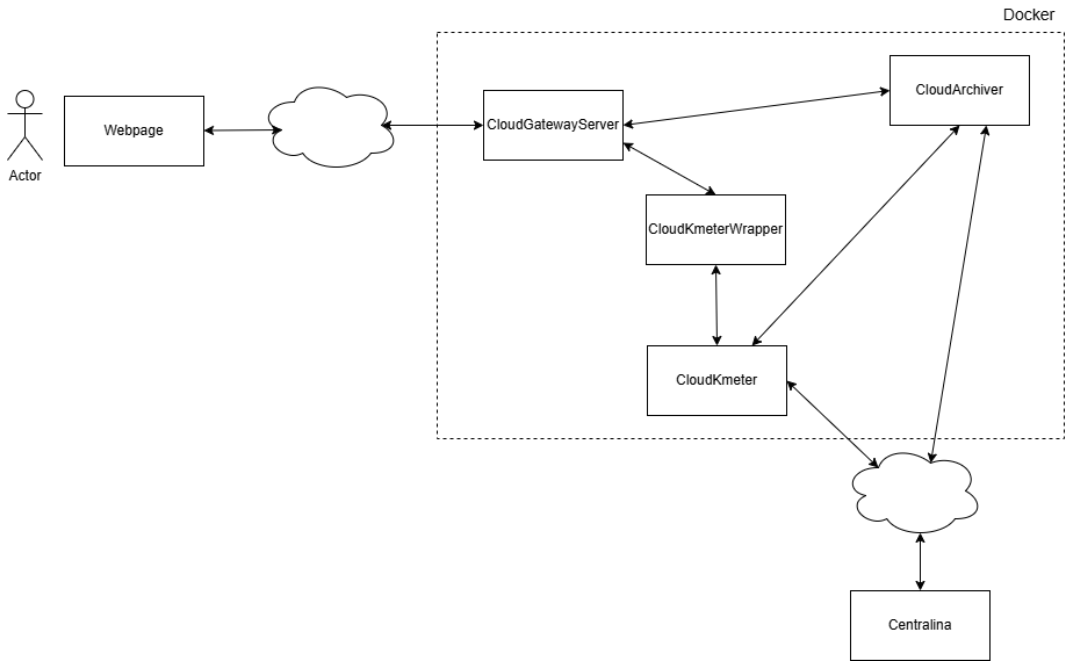


Figura 2: Schema ad alto livello della comunicazione tra moduli

3.3.1 Comunicazione tra moduli

Per consentire la comunicazione tra i micro-servizi sviluppati viene impiegato l'uso di un *message broker*.

L'utilizzo di *RabbitMQ* come *message broker* permette di sfruttare più protocolli di comunicazione contemporaneamente. Nello specifico vengono utilizzati: 1) il protocollo *MQTT* per avere una comunicazione asincrona tra i micro-servizi, sfruttando il paradigma *pubblish/subscribe*; 2) il protocollo *AMQP* per offrire una comunicazione più strutturata che in questo caso permette di sfruttare meccanismi che simulano la comunicazione sincrona offerta dal protocollo *HTTP*.

La scelta di utilizzare un *message broker* permette di sviluppare un'architettura con un'elevata tolleranza ai guasti. Nel caso in cui si provasse a comunicare con un servizio che risulta momentaneamente non raggiungibile, sarà il *broker* di messaggistica a occuparsi della consegna del messaggio quando il servizio ricevente tornerà ad essere operativo salvando il messaggio all'interno delle strutture dati utilizzate per comunicare con i servizi connessi.

Un *message broker* permette, inoltre, una maggiore scalabilità del sistema gestendo in modo dinamico un elevato numero di connessioni senza la necessità di particolari modifiche all'infrastruttura.

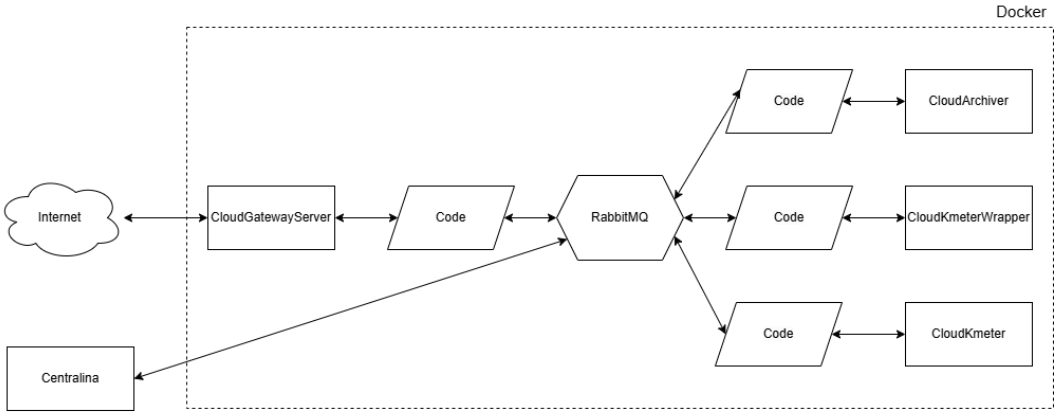


Figura 3: Schema a livello di message broker della comunicazione tra moduli

3.4 *CloudGatewayServer*

Il micro-servizio ***CloudGatewayServer*** è strutturato per essere l'unico punto di accesso al sistema. Espone una serie di *API REST* con le quali è possibile interagire con gli altri micro-servizi della piattaforma e ospita un'istanza del *web server Tomcat* che offre all'utente una pagina *web* con la quale accedere alla piattaforma *CloudREINEU*.

Nel momento in cui viene interrogata una determinata *API*, il micro-servizio ***CloudGatewayServer*** riceve la richiesta *HTTP* e la incapsula all'interno di un messaggio *AMQP*. Il messaggio creato viene inoltrato tramite il *message broker RabbitMQ* sulla coda del micro-servizio che dovrà gestire quella determinata richiesta.

Il codice del micro-servizio è strutturato per avere al suo interno una serie di *controller*, uno per ogni *API REST* esposta.

Un *controller RESTful* in *Spring Boot* è una componente che gestisce le richieste *HTTP* e risponde con dati o azioni, seguendo i principi dell'architettura *REST (Representational State Transfer)*. In *Spring Boot*, un controller *RESTful* è una classe annotata con `@RestController`.

```
@RestController
@RequestMapping(value = "api/params")
public class ParamsRestController {

	@GetMapping(value = "/**", produces=MediaType.APPLICATION_JSON_VALUE)
	public ResponseEntity<String> getHandler() {

		JsonObject response = new JsonObject();
		response.addProperty("brokerAddress", brokerAddress);
		response.addProperty("websocketPort", websocketPort);
		response.addProperty("brokerUsername", brokerUsername);
		response.addProperty("brokerPassword", brokerPassword);
		response.addProperty("serverPort", serverPort);

		String responseBody = gson.toJson(response);

		return new ResponseEntity<>(responseBody,HttpStatus.OK);
	}
}
```

Esempio di dichiarazione di un controller RESTful con il framework Spring Boot

3.4.1 Richiedere i parametri di connessione al *message broker*

Durante la fase di caricamento della *web page*, viene richiesto al micro-servizio ***CloudGatewayServer*** di comunicare i parametri necessari per stabilire una connessione con l'istanza di *RabbitMQ*.

L'API dedicata alla richiesta è la seguente:

```
hostname:ports/api/params
```

La classe *ParamsRestController* ha un *controller* che risponde allegando nel corpo della risposta HTTP un oggetto JSON che contiene: l'indirizzo *IP*, la *password* e lo *username* dedicati alla connessione della pagina *web*.

3.4.2 Richiedere l'avvio o lo spegnimento di una sessione di *test*

Quando una centralina di controllo è impostata come "passiva", è possibile avviare una sessione di *test* delle performance direttamente dal sistema *cloud*. Il micro-servizio che si occupa di eseguire il *test* in remoto è il servizio *CloudKmeter*. L'*API* dedicata alle richieste di avvio o di arresto di una istanza di *test* è la seguente:

```
hostname:port/api/kmeter/deviceDomain/DeviceSubdomain
```

La porzione dell'*API* formata dai campi `deviceDomain/DeviceSubdomain` rappresenta l'identificativo univoco della centralina di controllo connessa al sistema per la quale si vuole gestire la sessione di *test*.

La classe *KmeterRestController* ha al suo interno un *controller* che gestisce le richieste *POST*. L'intenzione di avviare o fermare una determinata istanza del micro-servizio *CloudKmeter* viene specificata inserendo nel corpo della richiesta *HTTP* il comando richiesto, sotto forma di oggetto JSON con la seguente sintassi:

```
{"command": COMMAND}
```

3.4.3 Navigare all’interno dell’archivio

Durante l’utilizzo della pagina *web* può essere necessario navigare all’interno dell’archivio delle sessioni di *test* per ottenere dati relativi a una sessione specifica. L’archivio viene gestito dal micro-servizio *CloudArchiver*, le richieste inoltrate dalla *web page* vengono incapsulate in un messaggio *AMQP*, attendendo la risposta del micro-servizio destinatario.

Per navigare all'interno dell'archivio, è necessario interrogare la seguente *API*:

```
hostname:port/api/archiver/foo/bar
```

La porzione dell’API composta da `/foo/bar` indica il percorso all'interno dell'archivio relativo al *file* o alla *directory* per la quale è richiesto l’accesso. Nel caso in cui la richiesta indica un *file*, la risposta sarà strutturata in modo da contenere nel *body* della risposta *HTTP* il contenuto del file richiesto. Nel caso di una *directory*, la risposta conterrà una lista degli elementi contenuti nella *directory* richiesta.

3.5 *CloudArchiver*

Il micro-servizio ***CloudArchiver*** è stato sviluppato per sostituire il funzionamento del modulo ***Archiver*** in esecuzione sulla centralina di controllo (vedi capitolo 2.1.2). Il vantaggio di avere un unico servizio di archiviazione in *cloud* è dato dalla possibilità di gestire contemporaneamente le sessioni di *test* di più centraline remote.

Nel momento in cui su una centralina di controllo viene avviata una nuova sessione di *test*, indipendentemente che questa sia impostata per essere “attiva” o “passiva”**,** viene inoltrato alla rete un messaggio di persistenza che ne notifica l'avvio. Il servizio ***CloudArchiver*** è strutturato per intercettare i messaggi di persistenza e registrare il traffico prodotto da tutte le centraline per le quali è attiva una sessione di *test*.

Nel caso di una centralina impostata come "attiva" vengono registrati i messaggi prodotti dai moduli ***meters*** e ***kmeters***, nel caso di una centralina "passiva" verranno registrati i messaggi prodotti dal modulo ***meters*** e dal micro-servizio ***CloudKmeter*** relativo a quella specifica centralina (vedi capitolo 3.7).

Per rendere il servizio ***CloudArchiver*** sovrapponibile al modulo ***archiver*** presente sulla centralina di controllo, il metodo con cui viene archiviato il traffico relativo a una sessione di *test* avviene attraverso la creazione di più *logfile*, uno per ogni sensore connesso alla centralina. Il contenuto dei *logfile* è strutturato in modo da contenere il corpo dei messaggi *MQTT* prodotti dai moduli in esecuzione, codificato nel formato *JSON*.

L'archivio che contiene i *logfile* relativi alle sessioni di monitoraggio viene creato dinamicamente sfruttando la struttura gerarchica dei *topic* relativi ai messaggi *MQTT* ricevuti. Questa caratteristica presenta due vantaggi sostanziali: 1) offre un alto grado di flessibilità e adattabilità del metodo di archiviazione nel momento in cui si operano delle modifiche sostanziali alla logica con la quale vengono strutturati i *topic*; 2) offre una corrispondenza diretta tra *topic* e *directory* dell'archivio.

Una seconda funzionalità offerta dal micro-servizio ***CloudArchiver*** è quella di fornire alla *webpage* (quindi all'utente che utilizza il servizio ***CloudREINEU***) gli strumenti necessari per navigare all'interno dell'archivio e consultare i *logfile* memorizzati.

Le comunicazioni tra la *webpage* e il micro-servizio ***CloudArchiver*** vengono mediate dal micro-servizio ***GatewayServer*** che è strutturato per funzionare come singolo punto di accesso al sistema. Per la comunicazione tra i due micro-servizi viene sfruttata una coda dedicata all'inoltro di messaggi *AMQP*.

Il micro-servizio ***GatewayServer*** riceverà la richiesta *HTTP* inoltrata della pagina *web* nella quale viene richiesto l'accesso a una risorsa presente nell'archivio. La richiesta *HTTP* verrà quindi incapsulata all'interno di un messaggio *AMQP* e inoltrata al micro-servizio ***CloudArchiver***, il quale risponderà offrendo la risorsa richiesta. Il messaggio *AMQP* una volta consumato dal micro-servizio ***GatewayServer*** verrà trasformato in risposta *HTTP* e inoltrato alla *webpage*. La scelta del protocollo *AMQP* permette di sfruttare il meccanismo di richiesta/risposta tipico dell'*HTTP*.

L'accesso al contenuto dell'archivio è stato progettato ispirandosi al principio *Hypermedia as the Engine of Application State* (*HATEOAS*) tipico delle *API RESTful*. Nell'approccio *HATEOAS*, quando un *client* effettua una richiesta, la risposta non contiene solo i dati richiesti, ma anche informazioni aggiuntive sotto forma di *link* che indicano quali altre operazioni possono essere effettuate da quel punto in poi. Utilizzando questa logica, il *client* non ha bisogno di conoscere a priori la struttura completa dell'*API* o i suoi *endpoint*, ma può navigare l'*API* seguendo i link dinamicamente forniti nella risposta.

I parametri relativi alla connessione con il *message broker*, alle code utilizzate per la comunicazione e la posizione dell'archivio sono modificabili sfruttando un *file* di configurazione.

3.5.1 Connessione al *message broker RabbitMQ*

Successivamente all'avvio del micro-servizio ***CloudArchiver***, verrà stabilita una connessione con il *message broker* tramite la creazione di tre code distinte:

- ***MqttQueue*:** all'interno di questa coda, verrà instradato il traffico *MQTT* prodotto dalle centraline di controllo e dai componenti *cloud* che simulano determinati moduli eseguiti all'interno delle centraline.
- ***MqttLoggerQueue*:** all'interno di questa coda verranno instradati i messaggi di persistenza prodotti dalle centraline connesse al *message broker*.
- ***ArchiverQueue*:** all'interno di questa coda verranno instradate le richieste relative all'accesso dell'archivio prodotte dalla pagina *web*.

Le code che ospiteranno i messaggi *MQTT* vengono legate al *topic exchange* chiamato *"amq.topic"*. Si tratta di un *exchange* predefinito in *RabbitMQ* che consente l'uso di *pattern* (in questo caso i *topic*) per distribuire i messaggi a più code.

Per differenziare i messaggi tra le due code *MQTT*, viene eseguita un'operazione di *binding* tra la coda ***MqttLoggerQueue*** e il *topic* "`#.kmeter.logger`", rendendo così possibile l'instradamento dei soli messaggi di persistenza relativi allo stato della centralina. Analogamente, alla coda ***MqttQueue*** viene unito il *topic* "`#`", il quale consente la ricezione di tutto il traffico MQTT prodotto dalle centraline. Verrà successivamente eseguita un'operazione di filtraggio in modo da scartare i messaggi che non è necessario archiviare.

La coda relativa ai messaggi *AMQP* sfrutta un *direct exchange*, un tipo di *exchange* offerto da RabbitMQ che permette di inoltrare i messaggi a una o più code in base a una *routing key* specifica, in questo caso la *routing key* utilizzata è "`rpc.archiver`".

```
@Bean
public TopicExchange topicExchange() {
        return new TopicExchange("amq.topic");
}

@Bean
public Queue mqttLoggerQueue() {
        return new AnonymousQueue();
}

@Bean
public Binding mqttQueueBinding(TopicExchange topicExchange,Queue mqttQueue) {
        return BindingBuilder.bind(mqttQueue).to(topicExchange).with("#");
}
```

Esempio di dichiarazione di un *topic exchange*, di una coda e del relativo *binding* basandosi sul *topic MQTT* specificato.

3.5.2 Gestire la persistenza delle centraline connesse al sistema

Le centraline di controllo sono programmate per inoltrare a intervalli di tempo prefissati dei messaggi di persistenza per segnalare la loro presenza in rete. I messaggi di persistenza vengono chiamati *"logMessage"* (vedi appendice).

Il micro-servizio ***CloudArchiver*** è strutturato per avere un *listener*, ovvero una funzione che permette di ricevere e consumare i messaggi, dedicato ai messaggi di persistenza che si trovano sulla coda *MqttLoggerQueue*. Si trova nella classe *MqttListener*. I messaggi ricevuti possono contenere al loro interno i tre tipi di stato relativi alla centralina di controllo (vedi capitolo 2.1.1).

Alla ricezione di un messaggio di persistenza che contiene il valore "ON", il *listener* aggiunge l'identificativo della centralina a una tabella chiamata *activeKmeters*. Conoscere quali centraline sono attive e monitorate è necessario per evitare di registrare messaggi "spuri" di centraline che non sono attualmente monitorate.

La ricezione di un messaggio di persistenza che contiene il valore "OFF", significa che è stata interrotta la sessione di *test* che era attiva per quella centralina, l'identificativo viene quindi rimosso dalla tabella *activeKmeters* e viene avviata una procedura di chiusura del *logfile* relativo alla sessione, richiamando il metodo *close* della classe *Writer*.

I messaggi provenienti dalla centralina di controllo che contengono il valore "PASSIVE" vengono ignorati perché per ogni centralina fisica impostata come "passiva", viene lanciato un processo *cloud* che simula il modulo ***kmeter*** locale alla centralina stessa**.** Sarà il modulo ***CloudKmeter*** a inviare il messaggio di persistenza che contiene il valore "ON".

```
@RabbitListener(queues = "#{mqttLoggerQueue.name}")
public void listenLogMessage(Message message) {
        [...]
}
```

Esempio di dichiarazione di un *listener* utilizzando **Spring for RabbitMQ** offerto da **Springboot**

3.5.3 Gestire la ricezione di un messaggio

Per consumare il traffico *MQTT* inoltrato sulla coda *MqttQueue*, viene sfruttata una funzione *listener* inclusa nella classe *MqttListener* che estrae dal messaggio ricevuto il corpo e il relativo *topic*. Conoscere il *topic* permette di filtrare i messaggi indesiderati basandosi su una *blocklist*, in questo caso vengono ignorati i messaggi di persistenza o i messaggi relativi a componenti del sistema che non è necessario archiviare.

Successivamente alla fase di filtraggio dei messaggi, si sfrutta il *topic* per estrarre l'identificativo della centralina che ha prodotto il messaggio e, se la centralina compare tra quelle attive (è presente nella tabella *activeKmeters*), si inoltra il messaggio ricevuto al metodo *write* della classe *LogFileWriter*.

3.5.4 Creazione di un *logfile* per una sessione di *test*

Il metodo *write* offerto della classe *LogFileWriter* permette di archiviare il messaggio ricevuto nel *logfile* relativo alla sessione di monitoraggio in corso.

Quando il messaggio ricevuto viene memorizzato nell'archivio, possono verificarsi due eventi: 1) esiste già un *file* relativo alla sessione in corso, il nuovo messaggio ricevuto viene scritto in coda al *file*; 2) non esiste un *file* relativo alla sessione di monitoraggio in corso. In questo caso si tratta del primo messaggio

relativo a nuova sessione, è necessario creare un nuovo *logfile*. La logica sfruttata dal metodo *write* può essere illustrata con il seguente grafico:

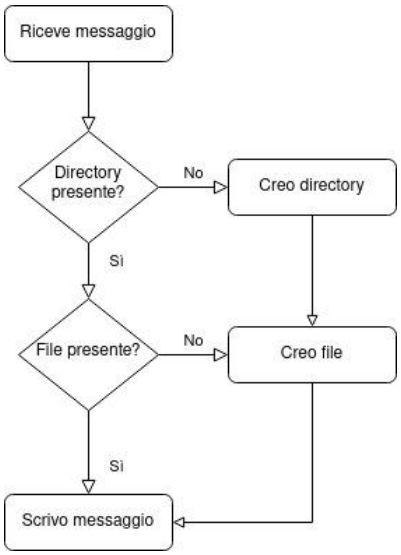


Figura 4: schema di funzionamento

Per mantenere una consistenza tra il funzionamento del modulo *archiver* attivo sulla centralina di controllo e il micro-servizio *CloudArchiver*, viene creato un *logfile* separato per ogni sensore collegato alla centralina.

Nel caso di una sessione di *test* attiva, viene creato un *file* chiamato `"data.log"`. Questo *file* conterrà il corpo dei messaggi MQTT relativi al singolo sensore, per la durata della sessione di *test*. Nel caso di una sessione di *test* conclusa, il file verrà rinominato con il *timestamp* di inizio e di fine sessione.

3.5.5 Chiusura di un *logfile* per una sessione di *test* conclusa

La ricezione di un messaggio di persistenza che contiene il valore "OFF", indica la conclusione di una sessione di *test* attiva per una determinata centralina di controllo. Viene avviata una procedura di chiusura del *logfile* relativo alla sessione, richiamando il metodo *close* della classe *Writer*.

Il metodo *close* è strutturato per cercare ricorsivamente all'interno dell'archivio tutti i *logfile* relativi alla sessione di *test* conclusa. Per ogni *logfile* trovato, avvierà una procedura di ridenominazione del file sfruttando la seguente

logica per il nome: “session_yyyyMMdd-HH:mm:ss_yyyyMMdd-HH:mm:ss”, dove la prima data è quella di inizio sessione e l’ultima è quella di fine sessione. Per determinare l'orario nel quale si è avviata la sessione di *test*, indipendentemente dal ciclo di vita del servizio *CloudArchiver*, si usa il *timestamp* contenuto nel primo messaggio ricevuto relativo a una determinata sessione.

```
public void close(String topic) {
        checkIfTopicIsEmpty(topic);
        String subdomain = getPathForSubdomainFromTopic(topic);
        ArrayList<String> list = findLogs(subdomain);
        long startTime = getStartTimeFromTable(list);
        long now = (System.currentTimeMillis());

        String fileName = generateFilename(startTime,now);
        for(String path : list) {
                renameLog(path,fileName);
        }
}
```

Porzione di codice utilizzata per gestire la chiusura dei *logfile* relativi a una sessione

3.5.6 Comunicazione con il micro-servizio *GatewayServer*

Le comunicazioni tra la *webpage* e il micro-servizio ***CloudArchiver*** vengono mediate dal micro-servizio ***GatewayServer.*** Il micro-servizio ***GatewayServer*** riceverà la richiesta *HTTP* inoltrata della pagina *web* nella quale viene richiesto l'accesso a una risorsa presente nell'archivio. La richiesta *HTTP* verrà quindi incapsulata all'interno di un messaggio *AMQP* e inoltrata al micro-servizio ***CloudArchiver***, il quale risponderà offrendo la risorsa richiesta. Il messaggio *AMQP* una volta consumato dal micro-servizio ***GatewayServer*** verrà trasformato in risposta HTTP e inoltrato alla *webpage*.

Per consumare questo tipo di richieste, è previsto un unico *listener* nella classe *AmqpListener* che consuma i messaggi *AMQP* inoltrati alla coda ***ArchiverQueue***. Le richieste ricevute possono essere di due tipi:

- **GET:** richiesta di accesso a una risorsa presente all'interno dell'archivio *cloud*.
- **DELETE:** richiesta di eliminazione di una risorsa presente all'interno dell'archivio *cloud*.

Alla ricezione di una richiesta "GET", il percorso specificato nel corpo dalla richiesta viene inoltrato al metodo *read* della classe *LogReader*, che risponderà con la risorsa richiesta o con un messaggio di errore nel caso la risorsa non fosse presente. Sebbene nel codice sia presente il metodo "DELETE", questo non è attualmente sfruttabile dall'utente e pertanto il suo funzionamento non verrà discusso.

3.5.7 Accedere al contenuto dell'archivio *cloud*

Quando viene richiamato il metodo *read* della classe *LogReader* è necessario specificare il precorso della risorsa richiesta dall'utente. Se questo percorso è valido, possono verificarsi due eventi: 1) il percorso richiesto è relativo a un *logfile*, la chiamata al metodo restituirà il contenuto del file di sessione; 2) Il percorso richiesto è una *directory*, il metodo restituirà il contenuto della *directory* sotto forma di *link* navigabili tramite *API*.

Nel caso in cui il percorso specificato non avesse una corrispondenza all'interno del repository, viene sollevata un'eccezione che permette di restituire un messaggio d'errore.

```
public String read(String requestPath) {
        if(requestPath.equals("")) throw new LogFileException("Void path");
        [...]
        if(itExists(updatedPath)) {
                if(isDirectory(updatedPath)) {
                        response = readDirectory(updatedPath,requestPath);
                } else {
                        response = readFile(updatedPath);
                }
        } else {
                throw new LogFileException("Unable to find: "+requestPath);
        }
        return response;
}
```

Porzione di codice relativo alla logica con la quale vengono gestite le letture.

3.6 *CloudKmeterWrapper*

L'approccio tipico dei micro-servizi nello sviluppo di applicazioni di rete porta gli sviluppatori a preferire una struttura basata su più servizi indipendenti, ciascuno con compiti limitati, rispetto a un singolo servizio monolitico. Il micro-servizio *CloudKmeter* è stato sviluppato per gestire il traffico di solo una centralina di controllo, per questo è prevista l'esecuzione simultanea di più istanze indipendenti.

Per ogni centralina di controllo impostata come "passiva" che si vuole monitorare tramite il sistema *CloudREINEU*, è necessario avviare una nuova istanza del micro-servizio *CloudKmeter***.** Il micro-servizio ***CloudKmeterWrapper*** è stato sviluppato con il compito di gestire l'esecuzione delle varie istanze del servizio *CloudKmeter***.**

All'avvio del servizio *CloudREINEU***,** il micro-servizio ***CloudKmeterWrapper*** crea una coda connettendosi con il *message broker* e si mette in ascolto del traffico *MQTT* prodotto dalle centraline connesse in rete.

Quando una centralina si annuncia come "passiva" tramite i messaggi di persistenza, significa che la centralina di controllo stessa si limita solamente a rilevare le temperature interne ed esterne alla cella e la potenza istantanea, lasciando che il calcolo del coefficiente di dispersione termica e l'archiviazione dei messaggi venga eseguito dal servizio *cloud*.

L'annuncio di una centralina impostata come "passiva" non coincide necessariamente con l'avvio di un'istanza del servizio *CloudKmeter*. Per avviare una sessione di *test* in *cloud* è necessario che l'operatore del servizio, tramite la pagina *web*, richieda esplicitamente che venga avviata un'istanza di *CloudKmeter* per una centralina tra quelle disponibili. Durante l'avvio del servizio ***CloudKmeterWrapper*** viene creata una seconda coda con lo scopo di poter comunicare con il micro-servizio ***GatewayServer*** in modo da poter ricevere le richieste di avvio.

3.6.1 Connessione al *message broker RabbitMQ*

In seguito all'avvio del micro-servizio, verrà stabilita una connessione con il *message broker* creando tre code distinte:

- ***loggerQueue*:** all'interno di questa coda verranno instradati i messaggi di persistenza prodotti dalle centraline connesse al servizio.
- ***cloudLoggerQueue*:** all'interno di questa coda verranno instradati i messaggi di persistenza prodotti dalle istanze del micro-servizio *CloudKmeter* in esecuzione.
- ***cmdQueue*:** all'interno di questa coda verranno instradati i messaggi necessari per avviare o per terminare una sessione di *test cloud* prodotti dagli utenti finali che utilizzano la pagina web.

Le code che ospiteranno il traffico *MQTT* sfruttano il *topic exchange* chiamato *"amq.topic"*, un *exchange* predefinito in *RabbitMQ* che consente l'uso di pattern.

Per differenziare i messaggi tra le due code MQTT, viene eseguita un'operazione di *binding* tra la coda ***loggerQueue*** e il *topic* `"from.#.#.kmeter.logger"`, rendendo così possibile l'instradamento dei soli messaggi di persistenza relativi allo stato della centralina. Analogamente, alla coda ***cloudLoggerQueue*** viene unito il *topic* `"cloudFrom.#.#.kmeter.logger"`. Per differenziare il traffico prodotto dai servizi *cloud* rispetto il traffico prodotto dalle centraline viene aggiunto il suffisso *"cloud"* al *subtopic* che indica la direzione del messaggio (vedi appendice).

La coda relativa ai messaggi AMQP prodotti dal micro-servizio ***GatewayServer*** sfrutta un *Direct Exchange*, un tipo di *exchange* offerto da RabbitMQ che permette di inoltrare i messaggi a una o più code in base a una *routing key* specifica, in questo caso la *routing key* utilizzata è `"rpc.cmd"`.

3.6.2 Gestire i messaggi di persistenza

Il sistema è strutturato per gestire due differenti tipi di messaggi di persistenza su due code separate: 1) I messaggi di persistenza contenuti nella coda ***loggerQueue***, inviati dalle centraline di controllo per monitorare quali centraline vengono impostate come "passive"; 2) i messaggi di persistenza contenuti nella coda ***cloudLoggerQueue***, inviati dai processi *CloudKmeter* per tenere traccia di quali sessioni di *test cloud* sono attive.

Per gestire i messaggi di persistenza contenuti nella coda ***loggerQueue***, il micro-servizio ***CloudKmeterWrapper*** prevede una funzione *listener* dedicata, che prende il nome *"listenToDevicesLogMessages"*.

Lo stato delle centraline di controllo non è predeterminato e può cambiare nel tempo secondo le necessità di utilizzo. Risulta necessario tenere traccia delle centraline che entrano nello stato "passivo" e delle centraline che escono dallo stato "passivo".

Per modellare correttamente sia lo scenario di una centralina che diventa “passiva”, sia lo scenario di una centralina che smette di essere “passiva” risulta necessario mantenere in memoria due liste separate:

- **passiveDevices**: contiene l'identificativo di tutte le centraline che si sono annunciate come "passive" e il *timestamp* di ricezione dell'ultimo messaggio di persistenza.
- **cloudKmeters**: contiene l'identificativo di tutte le centraline per le quali è stata avviata una sessione di *test* remota e il *timestamp* relativo all'ultimo messaggio di persistenza ricevuto dall'istanza del micro-servizio *CloudKemeter* associato a quella centralina.

Quando ricevuto un nuovo messaggio di persistenza, viene aggiornato il *timestamp* relativo all’elemento della lista che rappresenta la centralina o il processo *cloud* che ha prodotto il messaggio. Il vantaggio di avere una lista aggiornata con gli ultimi *timestamp* di ricezione permette di poter stabilire se una centralina ha dei problemi o se si è disconnessa.

Quando una centralina passa dallo stato "passivo" a un altro stato, deve essere interrotta la sessione di *test* per quella determinata centralina. La funzione *listener* che gestisce i messaggi, oltre a rimuovere la *entry* relativa alla centralina dalla lista ***passiveDevices*** e dalla lista ***cloudKmeters***, invia anche un messaggio di spegnimento al micro-servizio *CloudKemeter* relativo alla centralina che ha cambiato stato, in modo da interromperne l'esecuzione.

Il sistema prevede anche la presenza di un secondo *listener*, che prende il nome di *"listenToCloudKmetersLogMessages"*. Il compito di questa funzione è quella di consumare i messaggi che si trovano sulla coda ***cloudLoggerQueue***, ovvero i messaggi di persistenza inviati dalle istanze del micro-servizio ***CloudKmeter*** attive in un determinato istante.

In seguito alla richiesta da parte dell'utente di avviare una sessione di *test* per una determinata centralina di controllo impostata come "passiva", il micro-servizio ***CloudKmeterWrapper*** avvierà un'istanza del micro-servizio ***CloudKemeter***.

Successivamente all'avvio, il micro-servizio si annuncerà sulla rete mandando un messaggio di persistenza che contiene il valore "ON". Alla ricezione del messaggio, la funzione *listener* inserirà l'identificativo della centralina e il relativo *timestamp* di ricezione del messaggio nella lista **cloudKmeters.**

Quando l'utente decide di terminare una sessione di monitoraggio, il processo ***CloudKmeter*** relativo alla centralina inoltra un messaggio di persistenza con il valore "OFF" e successivamente si spegne. La funzione *listener* rimuoverà la *entry* relativa alla centralina di controllo dalla lista **cloudKmeters.**

3.6.3 Avviare una nuova istanza del micro-servizio *CloudKmeter*

L'utente utilizzatore del servizio ***CloudREINEU*** accede a una pagina web dalla quale può consultare una lista di centraline passive connesse al sistema. Per ogni centralina disponibile può avviare o fermare una sessione di monitoraggio. Dalla pagina *web* viene inoltrata una richiesta *HTTP* al micro-servizio *GatewayServer*,

questo a sua volta incapsula la richiesta in un messaggio *AMQP* e lo inoltra al micro-servizio *CloudKmeterWrapper*. Per consentire la comunicazione, viene sfruttata la coda `"cmdQueue"`.

Quando l'utente richiede l'avvio di una nuova sessione di monitoraggio, il servizio *CloudKmeterWrapper* esegue due verifiche: 1) verifica che la centralina per la quale è stato richiesto un test cloud sia ancora disponibile; 2) che nel frattempo non sia già stata avviata un'istanza del servizio *CloudKmeter*. Nel caso queste due condizioni siano verificate, viene avviata una nuova sessione di test cloud, in caso contrario viene restituito all'utente un errore.

Per avviare una nuova istanza del micro-servizio *CloudKmeter* è necessario conoscere l'identificativo della centralina da monitorare, questa informazione viene comunicata dal servizio *GatewayServer* all'interno della richiesta inoltrata sulla coda `cmdQueue`.

Il micro-servizio *CloudKmeterWrapper* è progettato per essere in grado di avviare più istanze di un processo esterno, è quindi necessario conoscere a priori il percorso in cui si trova l'eseguibile del micro-servizio *CloudKmeter*. Questa informazione, insieme alle credenziali per la connessione al *message broker*, possono essere definite prima dell'avvio del servizio *CloudKmeterWrapper* tramite un file di configurazione.

```
processBuilder.command(
        "java",
        "-jar",
        kmeterJarName,
        "--h="+brokerAddress,
        "--p="+brokerPort,
        "--u="+brokerUsername,
        "--psw="+brokerPassword,
        "--domain="+domain,
        "--subdomain="+subdomain
        )
        .inheritIO();
processBuilder.start();
```

Esempio di codice necessario per avviare un processo esterno tramite *Java* tramite la classe *ProcessBuilder*

La classe *ProcessBuilder* in *Java* è una delle principali classi utilizzate per creare e gestire processi di sistema esterni, ovvero eseguire comandi o programmi esterni dal codice *Java*. Questa classe consente di avviare un processo figlio, specificarne

l'ambiente di esecuzione, gestirne l'input/output e gestire eventuali errori. Il comando da eseguire viene creato tramite il costruttore o il metodo *command().*

Questo tipo di implementazione permette di avere un comportamento analogo a un'operazione di *fork* tipica del linguaggio *C*. La generazione di un processo *child* rende quest'ultimo totalmente indipendente dal ciclo di vita del processo *parent* che lo ha generato.

Quando viene richiesto di terminare una sessione di *test cloud*, è necessario arrestare la relativa istanza del servizio *CloudKmeter*. Poiché Il servizio *CloudKmeterWrapper* non può operare direttamente sul ciclo di vita di altri processi in esecuzione sulla macchina, è necessario utilizzare un canale di comunicazione tra processi. Viene inviato un messaggio *MQTT* sul *topic* `"cloudTo.DEVICE_ID.kmeter.shutdown"`, dedicato allo spegnimento della macchina. Successivamente, la centralina viene rimossa dalla lista *CloudKmeters*, in modo che sia possibile avviare nel futuro una nuova istanza.

3.6.4 Rimozione delle centraline non raggiungibili

Il processo ***CloudKmeterWrapper*** è progettato per avere al suo interno una *routine* che ciclicamente scorre le liste *passiveDevices* e *cloudKmeter* per rimuovere le centraline il cui ultimo messaggio di persistenza è più vecchio di un determinato periodo di tempo preimpostato (*TTL*). L'esecuzione di questa *routine* assicura la possibilità di avviare processi *CloudKmeter* solo per centraline che sono connesse al sistema e funzionanti.

```
@Scheduled(fixedDelayString = "${remover.time}", initialDelayString =
"${remover.time}")
public void deadDeviceRoutine() {
        [...]
}
```

Esempio di dichiarazione di una routine sfruttando il *framework Spring Boot.*

L'annotazione `@Scheduled` viene utilizzata per programmare l'esecuzione automatica di un metodo a intervalli regolari o in base a un'espressione *cron*. Il campo `fixedDelayString` indica il ritardo fisso tra la fine di una esecuzione

del metodo e l'inizio della successiva. Qui, il valore è specificato come una variabile esterna (proveniente da un file di configurazione). Il campo `initialDelayString` imposta un ritardo iniziale prima della prima esecuzione del metodo. Anche questo valore è impostato tramite una variabile esterna.

3.7 *CloudKmeter*

Il micro-servizio ***CloudKmeter*** è stato progettato per replicare in *cloud* il funzionamento di due moduli presenti sulla centralina: ***kmeters*** e ***heater***.

Il modulo ***kmeters*** si occupa di calcolare il coefficiente di dispersione termica della cella coibentata sulla quale si sta eseguendo una sessione di *test*. Il modulo ***heater*** si occupa dell'accensione e dello spegnimento degli elementi riscaldanti in modo da garantire una temperatura costante all'interno della cella.

Ogni istanza del processo *CloudKmeter* attiva all'interno del sistema *CloudREINEU* è associata a una centralina di controllo. Quando viene avviata un'istanza del micro-servizio *CloudKmeter,* questa si metterà in contatto con la centralina di controllo e inizierà una fase di scambio di informazioni. La centralina di controllo invierà al processo *CloudKmeter* i parametri specifici della cella sulla quale si sta eseguendo il *test*. Successivamente a questa fase, il servizio *CloudKmeter* si metterà in ascolto del traffico prodotto dalla centralina e calcolerà i seguenti indici:

- **TI:** calcolo della media delle temperature interne.
- **TE:** calcolo della media delle temperature esterne.
- **P:** calcolo della potenza assorbita.
- **K:** calcolo del coefficiente di dispersione termica.

Conoscendo la media delle temperature interne (TI) e delle temperature esterne (TE) alla cella coibentata, il servizio *CloudKemeter* si occupa anche di regolare la temperatura interna della cella mantenendola costante, in base a un valore di temperatura impostabile dall'utente tramite una sezione specifica della pagina *web*.

3.7.1 Connessione al *message broker*

All'avvio del micro-servizio ***CloudKmeter*** verrà stabilita una connessione con il *message broker* RabbitMQ creando diverse code e relativi *binding* per instradare specifici messaggi *MQTT*.

Il servizio ***CloudKmeter*** gestisce una comunicazione bidirezionale con la centralina di controllo. I messaggi provenienti dalla centralina vengono inoltrati su un *topic* che, come primo elemento, ha la chiave "`form/…`". I messaggi diretti verso la centralina hanno il prefisso "`to/…`".

Per differenziare i messaggi prodotti dalla centralina di controllo e i messaggi prodotti dai micro-servizi *cloud*, vengono usati prefissi "`cloudFrom/…`" e "`cloudTo/…`".

La pagina *web* non può comunicare direttamente con la centralina di controllo. Quando sono richiesti dei parametri specifici, la *webpage* contatta il servizio *CloudKmeter* che funzionerà come *proxy* per la centralina di controllo e risponderà al suo posto.

I messaggi scambiati durante il funzionamento del micro-servizio possono essere suddivisi in tre categorie principali: rilevazioni effettuate dalla centralina, parametri di funzionamento della centralina e richieste provenienti dalla pagina *web*. Vengono gestiti da una serie di code separate:

- **Code utilizzate per ricevere le rilevazioni effettuate dalla centralina**
  - ***meterQueue***: riceve i messaggi relativi alle rilevazioni di temperatura esterna e interna alla cella coibentata prodotte dalla centralina da monitorare. Viene effettuato un *binding* al *topic* "`from.<domain>.<subdomain>.meter.#`"
  - ***powerMeasurementQueue*:** riceve i messaggi relativi alle misurazioni di potenza, con un *binding* che ascolta il *topic* "`from.<domain>.<subdomain>.meter.Power`".

- **Code utilizzate per ricevere i parametri e le descrizioni da centralina:**
  - ***descriptionQueue*:** riceve i messaggi che contengono una descrizione dei parametri calcolati dal servizio ***kmeters*** avviato sulla centralina. La descrizione contiene il nome dei parametri da monitorare e l'unità di misura utilizzata. Viene effettuato un *binding* con il *topic* "`from.<domain>.<subdomain>.#.description`".
  - ***kmeterParamsQueue*:** riceve i messaggi che contengono i parametri specifici della centralina, vengono instradati sulla coda tramite un *binding* con il *topic* "`from.<domain>.<subdomain>.kmeter.params`".

    I parametri ricevuti contengono le informazioni necessarie per calcolare correttamente il coefficiente di dispersione termica della cella sulla quale si sta eseguendo il *test* (superficie, temperatura massima, temperatura minima, dimensione della finestra per eseguire il calcolo con media mobile, …) e altri parametri come la durata dell'intervallo per l'invio dei messaggi di persistenza (vedi appendice).
  - ***deviceStatusListenerQueue*:** riceve i messaggi che contengono una descrizione dei sensori connessi alla centralina di controllo. La descrizione contiene il nome dei sensori e l'unità di misura utilizzata. Viene effettuato un *binding* con il *topic* "`from.<domain>.<subdomain>.dev.#`".

- **Code utilizzate per ricevere le richieste ricevute da pagina *web*:**
  - ***requestToAllQueue*:** riceve le richieste relative ai parametri di funzionamento prodotte dalla pagina *web* instradate sul *topic* "`cloudTo.<domain>.<subdomain>.all`".
  - **requestToKmeterQueue:** riceve le richieste relative ai parametri relativi al servizio *kmeters* prodotte dalla pagina *web* e instradate

sulla coda tramite un bindign con il *topic* “`cloudTo.<domain>.<subdomain>.kmeter.request`”.

- **shutdownRequestQueue:** gestisce le richieste di spegnimento della centralina, con il topic “`cloudTo.<domain>.<subdomain>.kmeter.shutdown`”.

Le code che gestiscono il traffico *MQTT* utilizzano il topic exchange *"amq.topic"*, permettendo l'instradamento dei messaggi in base ai *pattern* specificati nei *binding*.

3.7.2 Gestione dei parametri di configurazione della centralina

All'avvio del micro-servizio seguono una serie di richieste verso la centralina di controllo per ottenere i dati operativi necessari per il funzionamento del servizio ***CloudKmeter***.

La classe *FetchParameters* si occupa di effettuare la richiesta per una descrizione dei sensori connessi alla centralina e una descrizione degli indici necessari per calcolare il coefficiente di dispersione termica. Inoltre, viene richiesta la lista completa delle impostazioni della centralina di controllo, in modo da allineare il servizio ***CloudKmeter*** con le misure che verrebbero effettuate dal processo *kmeters* sulla centralina. La descrizione dei sensori viene gestita dalla classe *DevicesStatusListener*, la descrizione degli indici e dei parametri relativi al modulo *kmeter* viene gestita dalla classe *FetchParameters.*

I parametri necessari al funzionamento del micro-servizio *CloudKmeter* vengono salvati come proprietà del sistema a livello *runtime*. Il metodo `System.setProperty(String key, String value)` permette di impostare una coppia chiave-valore che sarà conservata nella mappa delle proprietà di sistema. Queste proprietà possono essere poi recuperate da qualsiasi parte del codice usando `System.getProperty(String key).`

Quando la *web page* richiede i parametri di funzionamento del servizio *CloudKmeter*, in modo da mostrarli all'utente renderli disponibili alla modifica, la

richiesta viene gestita dalla classe *DescriptionRequestListener* che risponderà con i parametri ottenuti dalla centralina, funzionando come *proxy*. Se si riceve una modifica dei parametri, il servizio *CloudKmeter* inoltrerà i nuovi parametri alla centralina operando un *override* dei parametri.

3.7.3 Gestione dei messaggi prodotti dalla centralina di controllo

Il micro-servizio ***CloudKmeter*** si mette in ascolto di tutti i messaggi prodotti dalla centralina che sta monitorando. Quando viene ricevuto un messaggio riguardo la rilevazione di temperatura da parte di un sensore, questo viene salvato in una tabella chiamata *lastTemperatureValues.* Dopo un lasso di tempo prefissato dall'avvio dei micro-servizio, viene eseguita una *routine* che ciclicamente calcolerà la media dei valori interni e esterni.

Il valore della media delle temperature esterne e interne calcolato viene memorizzato in una lista (*averageTIList* e *averageTEList*) mettendolo in relazione con il *timestamp* relativo al momento in cui è stato calcolato, per renderlo disponibile al resto dei moduli in esecuzione all'interno del micro-servizio. Le rilevazioni riguardo la potenza istantanea calcolata dalla centralina vengono salvate nella lista *powerMeasurements*, che contiene tutte le rilevazioni di potenza e il *timestamp* di ricezione del messaggio.

3.7.4 Calcolo coefficiente di dispersione termica

Il coefficiente di dispersione termica è un parametro che misura la capacità di una struttura, come una cella coibentata, di trattenere il calore o di disperderlo verso l'esterno. Quando si parla di testare le prestazioni di una cella coibentata, il coefficiente di dispersione termica è cruciale perché fornisce indicazioni sulla sua efficienza termica. Il calcolo può essere eseguito con la seguente formula:

$$K = \frac{W}{S \cdot \Delta T}\ W/m^2$$

Dove:

- W: rappresenta l'energia utilizzata dal sistema di stufe per mantenere la temperatura costante, espresso in Wh
- $S = \sqrt{So \cdot Se}$ rappresenta la superficie della cella coibentata, espressa in m^2
- $\Delta T = TI - TE$: rappresenta la differenza tra le temperature interne e le temperature esterne alla cella coibentata, espresso in °C

Il micro-servizio ***CloudKmeter*** calcola il coefficiente di dispersione termica sfruttando i metodi della classe *Kmeter*.

Quando viene avviato il micro-servizio, viene eseguita una *routine* che permette di: 1) aggiornare il tempo del sistema con il *timestamp* corrente; 2) spostare la finestra che viene usata per il calcolo della media mobile; 3) calcolare il coefficiente di dispersione termica; 4) Inoltrare al *message broker* i messaggi relativi agli indici calcolati durante l'esecuzione della routine.

Le rilevazioni relative a temperature interne, esterne e potenza vengono estratte dalle liste condivise con le classi che si occupano di consumare i messaggi ricevuti dalla centralina. Dato che le liste hanno i dati espressi come una copia <*data-valore*> è facile estrarre solo le rilevazioni di temperatura relative alla finestra temporale nella quale si sta operando.

La dimensione della finestra temporale è un parametro che può essere impostato dall'utente tramite il *file* di configurazione del micro-servizio. Calcolare

il coefficiente di dispersione termica all'interno di una finestra temporale prefissata ha il vantaggio di smussare eventuali *spike* registrati all'accensione degli elementi riscaldanti.

```
@Scheduled(fixedDelayString = "${kmeter.time}", initialDelayString =
"${kmeter.time}")
private void kmeterRoutine() {
        try {
                updateTimeValues();
                updateMovingWindowStartTime();
                calculateK();
                sendMessages();
        } catch (KmeterException e) {
                logger.error("Kmeter: {}",e.getMessage());
        }
}
```

Esempio di dichiarazione di una routine con il *framework Spring Boot* e pseudo-codice relativo al calcolo del coefficiente di dispersione termica.

La classe *Kmeter* si occupa anche dell'invio in rete dei messaggi contenenti il valore del coefficiente di dispersione termica (K) calcolato e il valore della potenza totale assorbita dal sistema. Per distinguere i messaggi prodotti dal micro-servizio ***CloudKmeter***, il prefisso di questi messaggi sarà `"CloudFrom/…"`.

3.7.4 Regolazione della temperatura interna alla cella coibentata

Il micro-servizio ***CloudKmeter*** copre anche le funzionalità del modulo *heater* presente sulla centralina. Ottenuti i parametri relativi alla temperatura massima e minima di esercizio del sistema, ottenuto il parametro relativo alla tolleranza minima da usare durante i calcoli e la lista delle medie delle temperature interne del sistema, viene avviata una *routine* che permette di portare il sistema alla temperatura desiderata e mantenerlo costante.

Durante la prima esecuzione della routine è prevista una fase di inizializzazione nella quale si accendono le ventole del sistema e si impostano le variabili di sistema. L'algoritmo usato per la regolazione della temperatura interna alla cella sfrutta un sistema di "soglie mobili". Durante l'esecuzione del micro-servizio il valore di queste soglie verrà aumentato e diminuito dinamicamente in

base all'andamento della temperatura interna. Il superamento di una soglia prevede l'accensione di una stufa.

L'utilizzo di questo approccio permette di ridurre il numero di accensioni e spegnimenti consecutivi degli elementi riscaldanti.

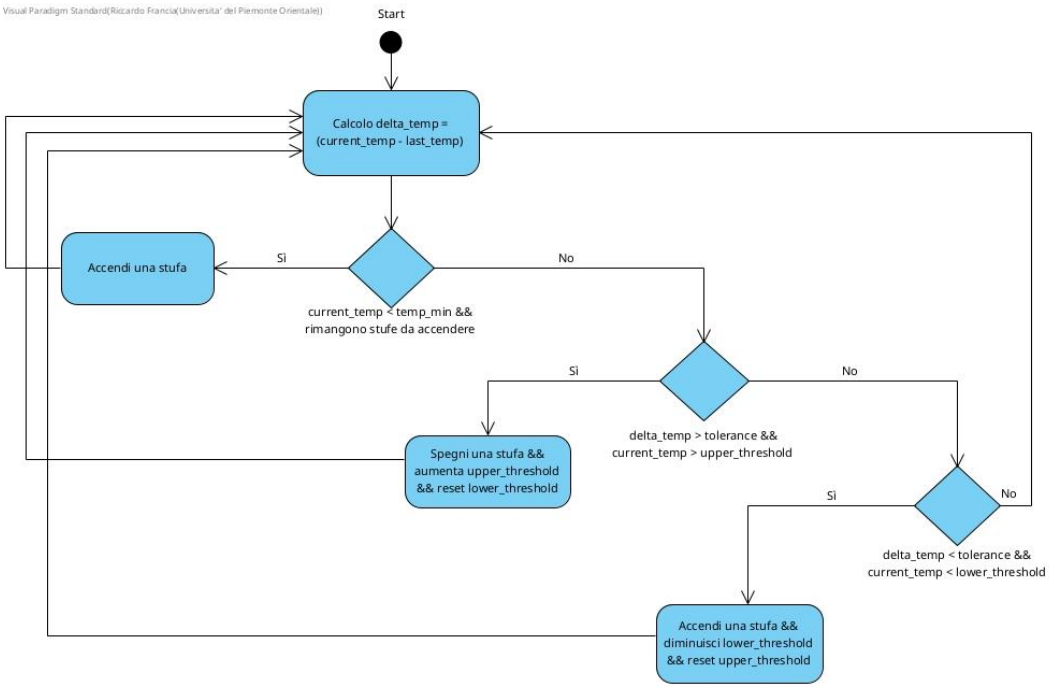


Figura 5: flowchart dell'algoritmo di regolazione della temperatura

Dove:

- Temperatura massima = $temp\_max$
- Temperatura minima = $temp\_min$
- Valore di tolleranza = $tolerance$
- Soglia massima = $upper\_threshold$
- Soglia minima = $lower\_threshold$

Il sistema inizialmente porrà la soglia massima e la soglia minima come:

$$\frac{(temp\_max + temp\_\min)}{2}$$

3.7.5 Invio di messaggi di persistenza

Il micro-servizio ***CloudKmeter*** è sviluppato in modo da comunicare ciclicamente la propria presenza all'interno della rete.

L'invio di messaggi di persistenza a intervalli regolari è utile per: 1) scoprire quali istanze sono avviate; 2) conoscere tempestivamente se un'istanza non è raggiungibile.

3.7.6 Arresto di un’istanza del servizio *CloudKmeter*

Il micro-servizio ***CloudKmeter*** predispone una coda per la ricezione dei messaggi di spegnimento. Un messaggio di spegnimento indica che l'utente ha richiesto lo stop della sessione di monitoraggio o che la centralina è stata spenta o non è più raggiungibile.

La routine di spegnimento prevede lo spegnimento di tutti gli elementi riscaldanti, lo spegnimento delle ventole e infine l'invio di un messaggio di persistenza nel quale si segnala alla rete lo spegnimento dell'istanza

3.8 *Web page*

Per rendere accessibili i servizi tramite un'interfaccia grafica, è necessaria l'implementazione di una pagina *web*. L'approccio scelto è quello della *single page application (SPA)* eseguita direttamente sul *browser* dell'utente (rendering *client-side*).

La scelta di una *single page application* è motivata dal voler strutturare un'interfaccia moderna e reattiva che permetta di eseguire gli *script* e il rendering degli elementi grafici direttamente nel *browser* dell'utente, sollevando il costo computazionale dal *server*. Tutti gli elementi della pagina vengono scaricati nel *browser* dell'utente solo durante la fase iniziale di caricamento del servizio, le successive interazioni con il *server* permettono di richiedere solo i dati necessari senza il bisogno di scaricare nuovamente tutta l'interfaccia grafica e gli *script*.

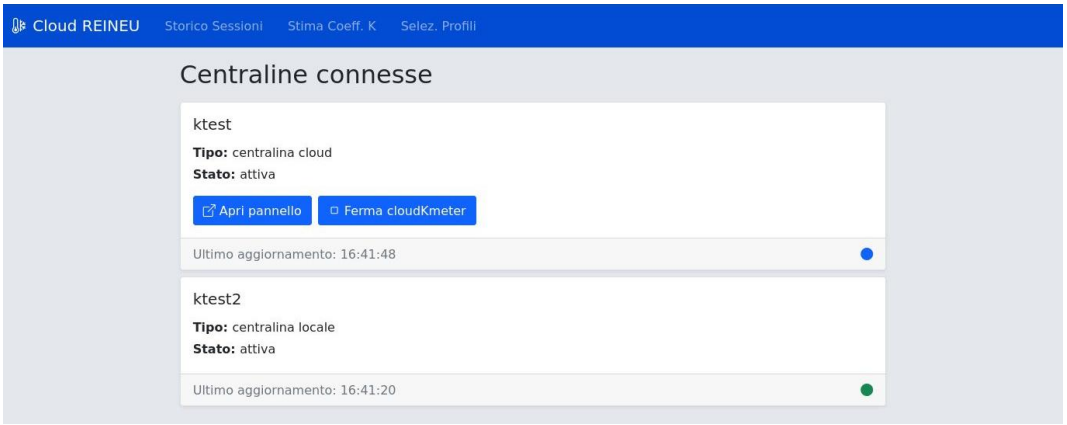
La pagina comunica con il servizio ***CloudREINEU*** interrogando le *API RESTful* offerte dal servizio *CloudGatewayServer* e riceve le misurazioni delle centraline connesse tramite un *websocket MQTT*.

La composizione della pagina è strutturata in modo da adattarsi alla dimensione del dispositivo in uso, ottimizzando così anche l'utilizzo su dispositivi mobili.

Per lo sviluppo della logica della pagina è stato usato *Javascript* come linguaggio di *scripting*, per la composizione grafica sono stati usati i *template HTML* e *CSS* offerti dal *framework Bootstrap*. L'approccio utilizzato è stato quello di separare il codice eseguibile dal codice *HTML*, preferendo la logica a classi rispetto alla logica di singoli *script* indipendenti.

3.8.1 Schermata iniziale

Quando l'utente si connette al sistema *CloudREINEU*, la pagina principale mostra una panoramica sulle centraline di controllo connesse.



Quando si connette al sistema una centralina locale, viene mostrato un pannello con l'identificativo della centralina, un riepilogo del suo stato e l'ultimo aggiornamento ricevuto.

Nel caso di una centralina "passiva", oltre a mostrare le stesse informazioni che vengono visualizzate per una centralina locale, il pannello permette eseguire due azioni: 1) accedere al pannello di controllo della centralina; 2) avviare o arrestare una sessione di *test*, gestita dal micro-servizio *CloudKmeter*.

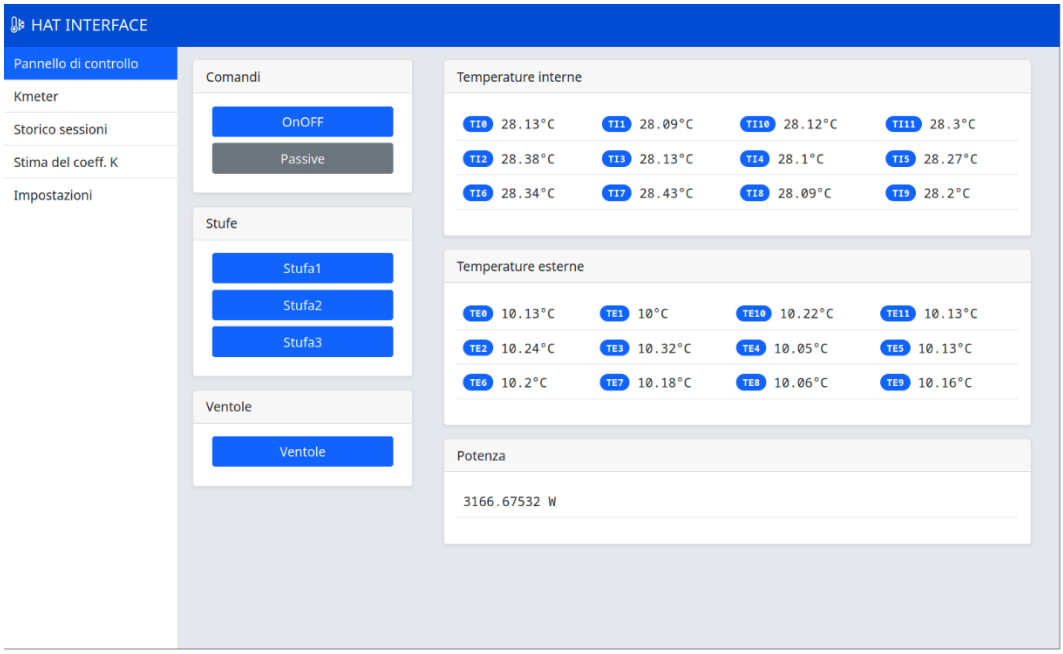
Lo stato delle centraline viene anche codificato con un codice colore, mostrato nel lato destro di ogni pannello:

- **blu**: centralina passiva con sessione di *test* in corso
- **verde**: centralina locale con sessione di *test* in corso
- **giallo**: centralina senza sessione di *test* in corso
- **rosso**: centralina non più raggiungibile

Tramite la barra di navigazione è possibile accedere allo storico delle sessioni e la schermata che permette di calcolare la stima del coefficiente di dispersione termica, gestite dal micro-servizio *CloudArchiver*.

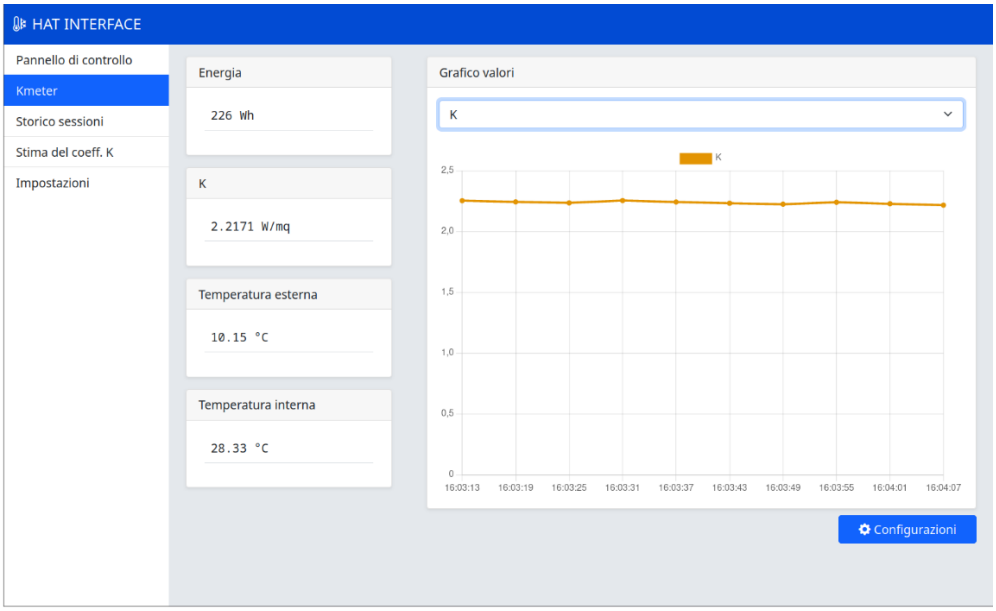
3.8.2 Pannello di controllo

Questa sezione permette di monitorare rapidamente lo stato della centralina. Dal pannello **“Comandi”** è possibile avviare o arrestare una sessione di *test* oppure selezionare la modalità “passiva” nel caso si volesse monitorare tramite il sistema *cloud*. Dal pannello **“Stufe”** è possibile monitorare il numero di elementi riscaldanti attivi all’interno della centralina e dal pannello **“Ventole”** è possibile attivare o disattivare le ventole. Nei pannelli posti a destra è possibile monitorare in tempo reale i valori di temperatura interna, esterna e potenza rilevati dai sensori collegati alla centralina.

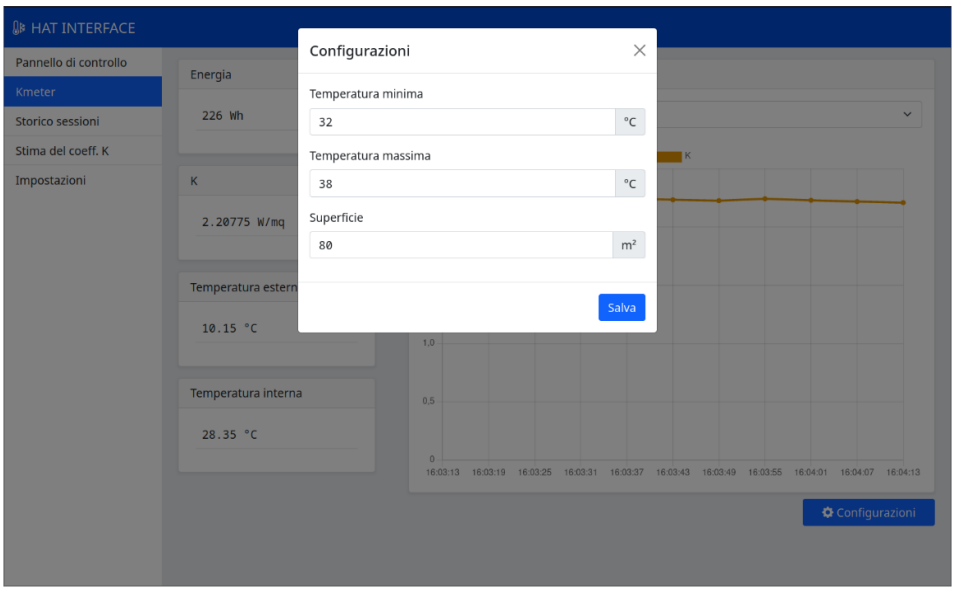


3.8.3 Sezione “kmeter”

Questa sezione permette di monitorare i valori misurati dal processo *kmeter* che è stato avviato sulla centralina o il corrispettivo processo *CloudKmeter*. Precisamente è possibile tenere traccia dell’energia consumata dal sistema, del valore del coefficiente di dispersione termica e delle medie delle temperature interne e esterne. Il grafico posto a destra dell’interfaccia permette di avere una visione più ampia dei valori registrati dal prototipo.



Premendo sul pulsante **"Configurazioni"** posto nell'angolo destro della schermata, è possibile accedere al pannello che permette di impostare la temperatura massima e minima che si desidera avere all'interno della cella. Questi valori saranno usati dal processo *heater* che, regolando l'accensione e lo spegnimento delle stufe, manterrà la temperatura interna alla carrozzeria pari a un valore attorno alla media tra temperatura massima e temperatura minima impostate. È inoltre possibile specificare la dimensione della superficie dei pannelli della cella. Verrà usato dal processo *kmeter* per stimare accuratamente il valore del coefficiente di dispersione termica (K).



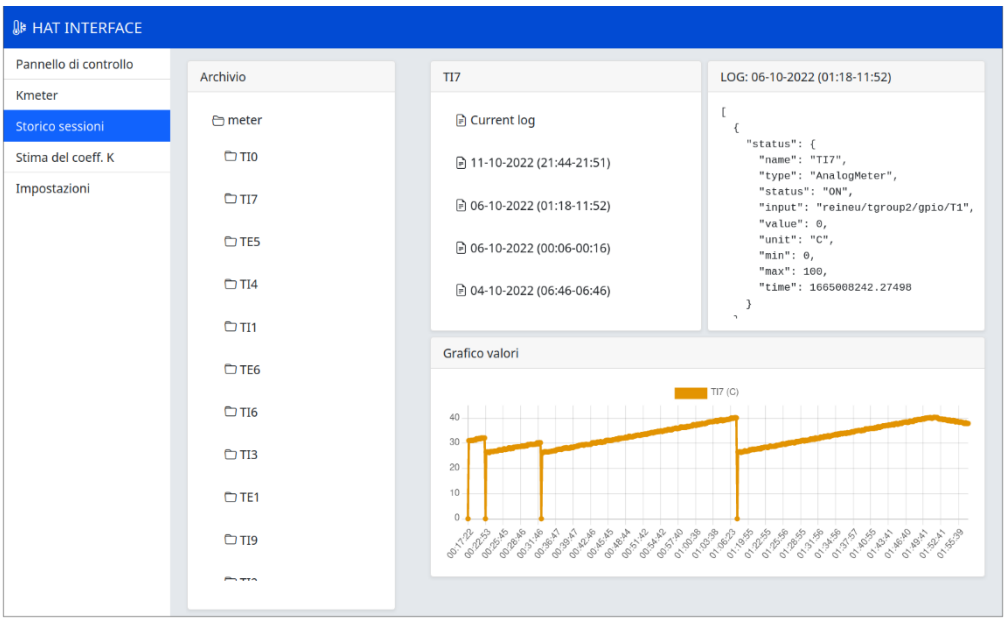
3.8.4 Storico sessioni

Questa sezione permette di navigare all'interno dell'archivio della centralina. Nell'archivio della centralina vengono salvati tutti i valori rilevati durante una sessione di monitoraggio. La gerarchia è *classeSensore/nomeSensore/fileSessione*.

Questa sezione permette di accedere alle sessioni delle seguenti classi di sensori:

- ***meter***: contiene le *directory* relative a tutti i sensori di temperatura e di energia collegati alla centralina
- ***kmeter***: contiene le *directory* relative a tutti i valori calcolati dal processo omonimo
- ***dev***: contiene tutti i messaggi scambiati con i dispositivi fisici della centralina, come i comandi di accensione e spegnimento delle stufe e delle ventole.

L'interfaccia grafica permette di navigare all'interno dell'archivio, la sezione più a sinistra mostra una lista dei sensori disponibili. Una volta selezionato un sensore, la sezione più centrale permette di navigare all'interno di tutte le sessioni salvate per quel determinato sensore. La parte più a destra mostra il contenuto testuale del *file* di sessione, mentre il grafico sottostante ne crea la sua rappresentazione grafica.

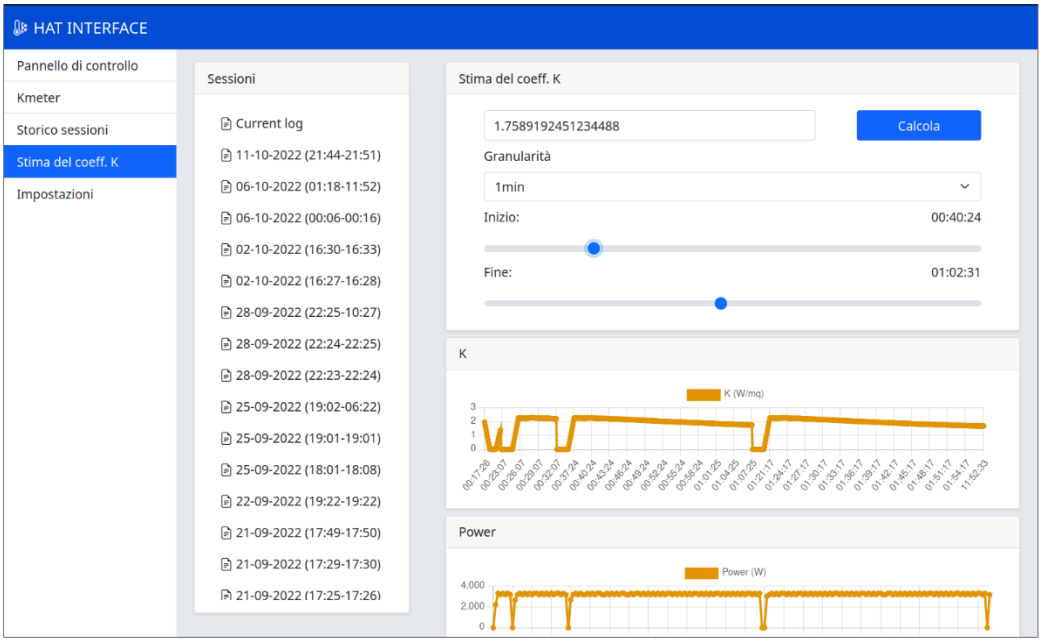


3.8.5 Stima del coefficiente di dispersione termica

Questa sezione permette di eseguire una stima più precisa del valore del coefficiente di dispersione termica (K), data una specifica sessione di *test*.

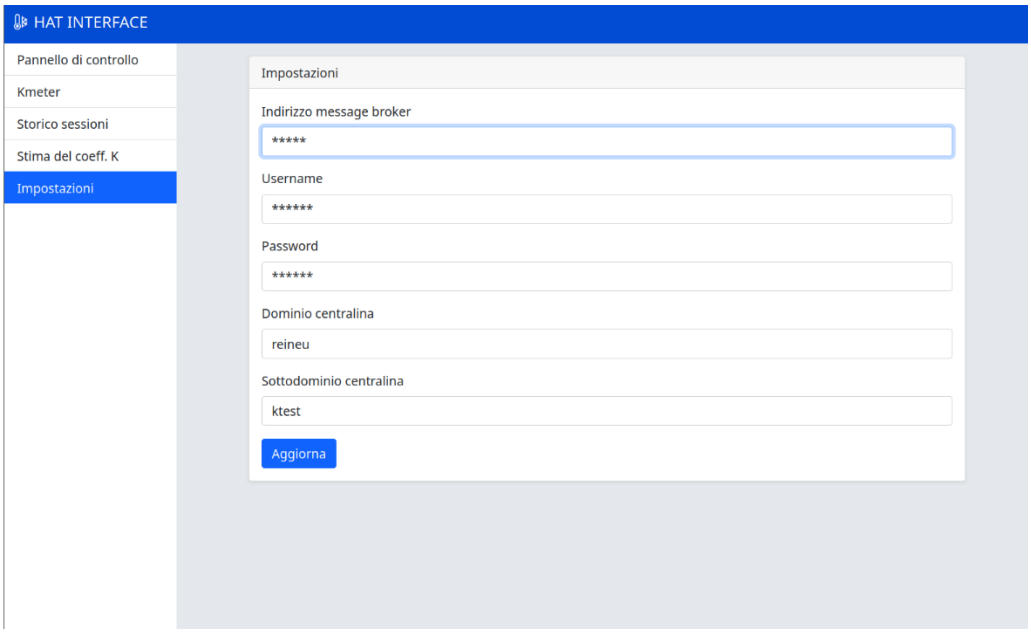
Il pannello a sinistra dell'interfaccia grafica permette di navigare tra le sessioni salvate all'interno dell'archivio della centralina. La restante parte dell'interfaccia grafica permette di selezionare una porzione di tempo specifica nella quale calcolare il valore K. Il selettore di granularità permette di avere più o meno precisione nella scelta del periodo temporale e i grafici sottostanti permettono di dare un'indicazione grafica del periodo di tempo selezionato.

La stima del coefficiente di dispersione termica eseguita da questa sezione viene calcolata tenendo conto del valore della superficie della carrozzeria che è impostano nelle configurazioni del modulo *kmeter*, per eseguire un calcolo più accurato vengono ignorati i valori del coefficiente di dispersione termica calcolati precedentemente dal micro-servizio e viene applicata nuovamente la formula prendendo in considerazione solo i valori contenuti nella finestra selezionata dall'utente.



3.8.6 Impostazioni

Questa sezione è disponibile solo per le centraline locali e permette di modificare rapidamente i dettagli della connessione verso il *message broker*, offrendo la possibilità di cambiare *server*, *account*, dominio e il sottodominio della centralina, che andranno a identificarla nel sistema *cloud*.



3.9 Esecuzione del sistema *CloudREINEU* tramite *Docker*

Docker è una piattaforma *open-source* che permette agli sviluppatori di creare, distribuire, eseguire, aggiornare e gestire *container* [18]. Docker permette di separare le applicazioni dall'infrastruttura fornendo la possibilità di eseguire le applicazioni in un ambiente isolato.

3.9.1 *Docker container*

Un *container* è un componente standardizzato e eseguibile che combina il codice sorgente dell'applicazione con le librerie del sistema operativo e le relative dipendenze richieste per eseguire il codice in qualsiasi ambiente. L'utilizzo dei *container* permette di sfruttare le capacità di isolamento dei processi e di virtualizzazione offerte dal *kernel* Linux, permettendo di condividere le risorse di una singola istanza del sistema operativo a più applicazioni.

La condivisione di risorse effettuata da Docker può essere paragonata alla condivisione di CPU, memoria e risorse di un hardware singolo effettuata da un *hypervisor* a più macchine virtuali [19]. A differenza di una tradizionale macchina virtuale, un container contiene solamente i processi e le dipendenze del sistema operativo necessarie per eseguire il codice. Questa caratteristica permette di ridurre la dimensione del *container* nell'ordine dei megabyte rispetto ai gigabyte di una macchina virtuale tradizionale, poiché non è richiesto che un *container* contenga l'intero sistema operativo e il relativo *hypervisor*.

Nel 2024 Docker ha una *market share* dell'83% rendendolo lo strumento di containerizzazione più diffuso [20].

La scelta di questo strumento è stata presa in visione di rendere il servizio ***CloudREINEU*** facilmente distribuibile e scalabile, permettendomi di prendere scelte progettuali indipendenti dal sistema operativo e dall'infrastruttura che avrebbe ospitato il servizio sviluppato. La facilità nella gestione dei *container* rende Docker uno strumento versatile anche nell'ipotesi che il sistema debba

essere mantenuto da figure esterne allo sviluppo del progetto, senza che abbiano conoscenze strettamente legate al funzionamento dei singoli servizi.

3.9.2 Creazione di un *container*

Avendo disponibile un *file* eseguibile relativo a un determinato servizio, per creare un *container Docker* è necessario eseguire i seguenti passi:

1. Creare una *directory* contenente l'eseguibile e il relativo *dockerfile*
2. Costruire un'immagine *Docker* basandosi sul *dockerfile*
3. Avvio del *container* basato sull'immagine creata nel passo precedente

1) **Creare una *directory* contenente l'eseguibile e il relativo *dockerfile***: Un *dockerfile* è un file di testo che contiene al suo interno le istruzioni necessarie per automatizzare la creazione di un'immagine *Docker* garantendo consistenza indipendentemente dal luogo nel quale l'immagine viene creata. Il *dockerfile* permette di specificare: l'immagine di base da cui partire, quali file aggiungere per il corretto funzionamento dell'eseguibile, quali comandi sono necessari per avviare l'eseguibile e quali porte esporre per permettere la comunicazione di rete.

```
FROM openjdk:17
EXPOSE 5672
COPY CloudArchiver-* CloudArchiver.jar
ENV USERNAME="test"
ENV PASSWORD="test"
ENTRYPOINT java -jar /CloudArchiver.jar --u=${USERNAME} --psw=${PASSWORD} --
h=localhost --p=5672
```

Esempio di dichiarazione di un *dockerfile* relativo al micro-servizio *CloudArchiver*.

- `FROM openjdk:17`: Per creare un *container* è necessario basarsi su un'immagine base tra quelle che vengono offerte da *Docker*. Dal momento che i micro-servizi del progetto *CloudREINEU* sono stati sviluppati in Java, ho scelto di basare il *container* su un'immagine che offre una distribuzione di *Debian* contenente le librerie necessarie per eseguire il codice *Java*, in questo caso viene sfruttata la libreria *OpenJDK 17*.

- `EXPOSE 5672:` Questa istruzione permette di dichiarare quali porte del *container* devono essere esposte in modo da poter comunicare con il mondo esterno. In questo caso viene esposta la porta `5672`, utilizzata dal protocollo *AMQP* e quindi necessaria per rendere possibile la comunicazione tra il container e il *message broker RabbitMQ*.

- `COPY CloudArchiver-* CloudArchiver.jar:` Questa istruzione permette di copiare l'eseguibile che si trova nella stessa *directory* del *dockerfile* all'interno del *container*. L'utilizzo del carattere `"*"` permette di copiare qualsiasi versione dell'eseguibile si trovi posizionata nella stessa cartella, automatizzando ulteriormente la creazione di un nuovo container in vista di un potenziale aggiornamento del servizio.

- `ENV USERNAME="test", ENV PASSWORD="test":` Questi comandi permettono di specificare quali variabili d'ambiente verranno utilizzate all'interno del *container*. Per il servizio *CloudArchiver* le variabili d'ambiente sono le credenziali usate per autenticare il micro-servizio sul *message broker*. Vanno viste come dei "segnaposto", al lancio effettivo del *container* queste verranno sovrascritte con i valori specificati dall'utente.

- `ENTRYPOINT java -jar /CloudArchiver.jar --u=${USERNAME} --psw=${PASSWORD} --h=localhost --p=5672`: La seguente istruzione permette di specificare il comando che verrà eseguito all'avvio del *container*. In questo caso si può notare l'utilizzo delle variabili d'ambiente.

2) **Costruire un'immagine Docker basandosi sul *dockerfile:*** Generalmente, per creare un'immagine *Docker* partendo da un *dockerfile*, è necessario aprire una finestra del terminale nella cartella in cui si trova il *dockerfile* e eseguire il seguente comando:

```
docker build -t nome-immagine .
```

Dove, nello specifico:

- `-t nome-immagine:` permette di assegnare un nome specifico all'immagine creata
- `"."`: specifica che il *dockerfile* si trova nella *directory* corrente, in caso contrario va specificato il percorso.

Il sistema *CloudREINEU* è formato da più micro-servizi che operano in maniera indipendente, a ogni micro-servizio corrisponde un relativo *container*. Per automatizzare ulteriormente la creazione delle immagini *Docker*, evitando la possibilità di errori umani durante la creazione delle relative immagini, ho sfruttato lo strumento ***Docker compose***.

Docker compose è uno strumento offerto da *Docker* che permette di definire applicazioni *multi-container* in modo strutturato. A tal proposito viene utilizzato un file di configurazione in formato *YAML* per descrivere i servizi, le reti e i volumi necessari per generare le immagini di un'applicazione composta da più container.

Di seguito riporto una porzione del file *docker-compose.yaml* che permette di creare l'immagine Docker relativa al servizio **CloudArchiver**. Nel file completo sono specificate anche le istruzioni che servono per creare le immagini relative agli altri servizi.

```
version: '3.4'
services:
    archiver:
        build:
            dockerfile: dockerfile
            context: ./archiver
        environment:
            USERNAME: "username"
            PASSWORD: "password"
        volumes:
            - ~/reineuCloud:/root/reineuCloud
        network_mode: "host"
        restart: always
[...]
```

- **“version: '3.4'”**: Definisce la versione dello schema di *Docker compose* utilizzato.
- **“Services”**: Definisce i servizi che saranno eseguiti in container separati
- **“Build”**: Questa sezione indica che *Docker compose* deve costruire un'immagine Docker per il servizio *CloudArchiver* utilizzando un *Dockerfile*.
- **“Dockerfile”**: Specifica il nome del *dockerfile* che *Docker* utilizzerà per costruire l'immagine.
- **“context”**: Definisce il contesto di *build*, che è la *directory* in cui *Docker* cercherà i file per costruire l'immagine.
- **“Environment”**: Le variabili d'ambiente **USERNAME** e **PASSWORD** vengono passate al *container*.
- **“Volumes”**: Definisce un volume che monta una *directory* dell'*host* (`~/reineuCloud`) all'interno del *container* (`/root/reineuCloud`), consente la condivisione persistente di dati tra il sistema host e il container.
- **“network_mode: "host"”**: Permette al *container* di condividere l'interfaccia di rete dell'*host*. Il *container* non avrà un proprio indirizzo *IP*, ma utilizzerà quello dell'*host*, utile quando il servizio deve comunicare direttamente con dispositivi o servizi sulla rete locale dell'*host* senza il livello di isolamento offerto da *Docker*.
- **“restart: always”**: Specifica che il *container* deve essere sempre riavviato automaticamente se si arresta per qualsiasi motivo, garantendo un'elevata disponibilità del servizio.

3.9.3 Gestione del ciclo di vita di un *container*

Successivamente alla definizione del *file docker-compose.yml*, è possibile avviare il processo di creazione delle immagini relative ai *container* specificati eseguendo il seguente comando:

```
docker-compose up
```

L'esecuzione di questo comando avvierà tutti i servizi definiti *nel file docker-compose.yml*, scaricherà le immagini *Docker* specificate se non sono già presenti sul sistema e, infine, avvierà i container e li connetterà alle reti e ai volumi definiti. Per fermare i *container* eseguiti con Docker Compose:

```
docker-compose down
```

Questo fermerà e rimuoverà i *container*, le reti e i volumi definiti nel *file*.

3.9.4 Avviare un'istanza di *RabbitMQ* come *container Docker*

Per avviare un *container* Docker che rende disponibile un'istanza di RabbitMQ in grado di supportare anche il protocollo MQTT, è necessario eseguire il seguente comando:

```
sudo docker run -d
--restart unless-stopped --hostname rabbit --name rabbitmq
-e RABBITMQ_DEFAULT_USER=user
-e RABBITMQ_DEFAULT_PASS=password
-p 5672:5672 -p 1883:1883 -p 15672:15672 -p 15675:15675 rabbitmq:3-management
```

I parametri più importanti sono i seguenti:

- **"--restart unless-stopped":** Questa opzione permette di riavviare automaticamente il *container* nel caso in cui si arrestasse inaspettatamente, a meno che non venga esplicitamente fermato dall'utente.

- **“-e RABBITMQ_DEFAULT_USER=user, -e RABBITMQ_DEFAULT_PASS=password”:** Queste opzioni impostano delle variabili d'ambiente all'interno del *container*. Nello specifico, configurano le credenziali per l'utente di default di RabbitMQ.

- **“-p 5672:5672 -p 1883:1883 -p 15672:15672 -p 15675:15675”:** Questa parte permette di configurare il *port mapping* tra il sistema *host* e il *container*. Nello specifico, permette di esporre le porte del *container* verso l'esterno, rendendolo accessibile dal sistema *host*:
    - **5672:5672**: Porta di *default* per il protocollo AMQP, utilizzato dai *client* RabbitMQ.
    - **1883:1883**: Porta per MQTT.
    - **15672:15672**: Porta per l'interfaccia *web* di gestione di RabbitMQ.
    - **15675:15675**: Porta per STOMP.

3.9.5 Abilitare *RabbitMQ* per supportare il protocollo *MQTT* e l’utilizzo *websocket*

Una volta avviata un’istanza di *RabbitMQ* tramite *Docker*, è possibile accedere alla console del *container* in esecuzione tramite il seguente comando:

```
docker exec -it rabbitmq /bin/bash
```

Accedere alla console del *container* in esecuzione è necessario per abilitare i *plugin* che permettono di ampliare le funzionalità del *message broker*. Di seguito vengono elencati i *plugin* che sono necessari per rendere questa istanza di *RabbitMQ* compatibile con il sistema ***CloudREINEU***:

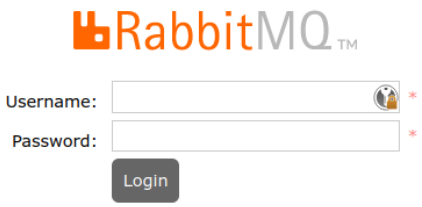
```
rabbitmq-plugins enable rabbitmq_management
rabbitmq-plugins enable rabbitmq_mqtt
rabbitmq-plugins enable rabbitmq_web_mqtt
rabbitmq-plugins enable rabbitmq_amqp1_0
```

- **“rabbitmq_management”:** Questo comando abilita il *plugin* di gestione di *RabbitMQ*, offre un'interfaccia *web* per monitorare e gestire *RabbitMQ*. Una volta abilitato, l'interfaccia web di gestione è solitamente accessibile alla porta **15672**.
- **“rabbitmq_mqtt”:** Abilitando questo plugin, *RabbitMQ* può ricevere e inviare messaggi utilizzando il protocollo *MQTT*, permettendo così a dispositivi o applicazioni che usano *MQTT* di interfacciarsi con *RabbitMQ*.
- **“rabbitmq_web_mqtt”:** Questo comando abilita il plugin Web MQTT, che fornisce un'interfaccia MQTT basata su WebSocket, utile soprattutto per applicazioni web.
- **“rabbitmq_amqp1_0”:** Abilita il supporto per il protocollo ***AMQP 1.0***. *RabbitMQ* nativamente utilizza il protocollo ***AMQP 0-9-1***, la versione 1.0 è più evoluta e ampiamente utilizzata in scenari *enterprise* e applicazioni *cloud* che necessitano di interoperabilità tra diversi sistemi di messaggistica.

3.9.6 Testare il corretto avvio del *container RabbitMQ*

Nel caso in cui l'ambiente nel quale è stato avviato il *container Docker* relativo a *RabbitMQ* è dotato di ambiente grafico, è possibile testare il corretto funzionamento del *container* collegandosi all'indirizzo `localhost:15672`. Nel caso non fosse possibile accedervi direttamente, è necessario aprire la porta `15672` del dispositivo nel quale è installato Docker e accedervi tramite un altro dispositivo specificando `IP_SERVER:15672`.

Nel caso in cui il *container* di *RabbitMQ* è stato avviato correttamente ci si dovrebbe trovare davanti a un portale di ingresso:



Inserendo le credenziali di *RabbitMQ* impostate all'avvio del *container*, è possibile accedere al pannello di controllo di *RabbitMQ*. Dal pannello principale (*overview*) è possibile accedere al menù "*ports and contexts*". Espandendo il menù è possibile controllare che le seguenti porte risultino aperte:

Ports and contexts

Listening ports

Protocol	Bound to	Port
amqp	::	5672
clustering	::	25672
http	::	15672
http/prometheus	::	15692
http/web-mqtt	::	15675
mqtt	::	1883

In questo caso l'istanza dei *message broker* è configurata correttamente e supporta anche lo standard *MQTT*, indispensabile per comunicare con alcune componenti del sistema **CloudREINEU** e le centraline di controllo.

4 Conclusioni

Il sistema sviluppato è risultato funzionante e stabile durante i test eseguiti. La scelta di sviluppare un sistema basato su micro-servizi distribuiti in rete e l'utilizzo di *message broker* per la comunicazione è risultata vantaggiosa in termini di sviluppo e adeguata al tipo di infrastruttura implementata. Queste scelte architetturali e implementative hanno contribuito allo sviluppo di software moderno, reattivo, stabile e facilmente scalabile.

La fase successiva di *test* prevede l'utilizzo in ambito industriale del sistema ***CloudREINEU***, la raccolta di *feedback* e l'eventuale aggiunta di servizi accessori secondo le necessità dell'azienda.

5 Bibliografia

[1]. **Spring Framework** https://www.ibm.com/cloud/learn/java-spring-boot

[2]. **Dependency Injection** https://it.wikipedia.org/wiki/Dependency_injection

[3]. **Spring Boot** https://www.ibm.com/cloud/learn/java-spring-boot

[4]. **Java** https://en.wikipedia.org/wiki/Java_(programming_language)

[5]. **RabbitMQ** https://it.wikipedia.org/wiki/RabbitMQ

[6]. **RabbitMQ…protocolli** https://www.rabbitmq.com/protocols.html

[7] **AMQP** https://www.rabbitmq.com/tutorials/amqp-concepts.html

[8]. **Eclipse** Mosquitto https://mosquitto.org

[9]. **Bridge** https://projects.eclipse.org/projects/iot.mosquitto

[10]. **MQTT** https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[11]. **QoS** https://en.wikipedia.org/wiki/MQTT

[12]. **Keycloak** https://www.keycloak.org/about

[13]. **Bootstrap** https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

[14]. **Responsive web design** https://it.wikipedia.org/wiki/Bootstrap_(framework)

[15]. **Javascript** https://en.wikipedia.org/wiki/JavaScript

[16]. **Motore Javascript** https://it.wikipedia.org/wiki/JavaScript

[17]. **HTTP** https://it.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[18]. https://www.ibm.com/topics/docker

[19]. ibidem

[20]. https://www.statista.com/statistics/1256245/containerization-technologies-software-market-share/

6 Appendice

6.1 Struttura dei *topic* utilizzati per i messaggi *MQTT*

Ogni modulo della centralina produce dei messaggi MQTT che contengono i valori letti dai sensori collegati o dei dati relativi al funzionamento del sistema. Il formato dei messaggi è standard, i topic sono strutturati gerarchicamente nel seguente modo:

```
direzione/nome_organizzazione/nome_centralina/servizio/id_sensore
```

Dove il campo “direzione” può assumere il valore “`from`” se il messaggio è stato prodotto da un determinato servizio o “`to`” se il messaggio è diretto verso un determinato servizio. Nel caso di servizi *cloud* viene usato “`cloudTo`” e “`cloudFrom`”

6.2 Struttura e contenuto dei messaggi scambiati

6.2.1 Modulo *meters*

```
from/reineu/ktest/meter/TI8

{"status":{"name":"TI8",  "type":"AnalogMeter",  "status":"ON",  "input":"reine
u/tgroup2/gpio/T2",  "value":29.20000,  "unit":"C",  "min":0,  "max":100,  "tim
e":1726083260.68615}}
```

Esempio di messaggio relativo a una rilevazione di temperatura eseguita da uno dei sensori collegati.

```
from/reineu/ktest/meter/Power

{"status":{"name":"Power",  "type":"ElectricMeter",  "status":"ON",  "input":"r
eineu/ktest/gpio/IN0",  "energy":0.00000,  "Ipower":0.00000,  "unit":"W",  "min
":0,  "max":3000,  "value":0.00000,  "time":1726083685.11890}}
```

Esempio di messaggio relativo a una rilevazione di potenza istantanea eseguita da uno dei sensori collegati.

6.2.2 Modulo *kmeters* e *heater*

```
from/reineu/ktest/kmeter/TI

{"status":{"name":"TI","min":-
50,"max":100,"unit":"C","value":33.34000,"time":1726083865.27966}}
```

Esempio di messaggio relativo al calcolo della temperatura media dei sensori interni alla cella.

```
from/reineu/ktest/kmeter/K

{"status":{"name":"K","min":0,"max":1,"unit":"W/mq","surface":80,"value":1.6461
0,"time":1726083871.27967}}
```

Esempio di messaggio relativo al calcolo del coefficiente di dispersione termica (K)

```
from/reineu/ktest/kmeter/logger

{"status":{ "name":"ktest", "mode":"ON", "time":"1726083871.29832"}
```

Esempio di messaggio di persistenza.

```
to/reineu/ktest/dev/Ventole {cmd:ON}
to/reineu/ktest/dev/Stufa1 {"cmd":"ON"}
to/reineu/ktest/dev/Stufa2 {"cmd":"OFF"}
to/reineu/ktest/dev/Stufa3 {"cmd":"OFF"}
```

Esempio di messaggi prodotti dal modulo *heater* e diretti al modulo che si occupa di gestire l'azionamento degli elementi riscaldanti durante l'esecuzione di una sessione di *test*.

6.2.3 Richiesta dei parametri

All'avvio di un'istanza del micro-servizio *CloudKmeter* vengono inviate una serie di richieste alla centralina alle quali viene risposto con i parametri operativi della centralina stessa.

Richiesta per descrizione degli elementi connessi alla centralina:

```
to/all

{request:description.json}
```

Risposte:

```
from/reineu/ktest/meter/description

{"reineu/ktest/meter":[
        {"name":"Power","type":"ElectricMeter","status":"ON","input":"reineu/kt
est/gpio/IN0","energy":303.00000,"Ipower":3022.52620,"Apower":358.13048,"unit":
"W","time":"1635264494.65476"},
        {"name":"TE0","type":"AnalogMeter","status":"ON","input":"reineu/tgroup
3/gpio/T0","value":10.13000,"unit":"C","time":"1635264510.64488"},
{"name":"TE1","type":"AnalogMeter","status":"ON","input":"reineu/tgroup3/gpio/T
1","value":10.00000,"unit":"C","time":"1635264511.74292"},

[…]

        {"name":"TI11","type":"AnalogMeter","status":"ON","input":"reineu/tgrou
p2/gpio/T3","value":28.71000,"unit":"C","time":"1635264506.44653"}]}
```

```
from/reineu/ktest/dev/description

{"reineu/ktest/dev": [
{"name":"OnOFF","type":"onoff","active":"ON","input":[],"output":[
        {"name":"reineu/ktest/gpio/OUT4","alias":"OnOff","status":"OFF"}]},
{"name":"Stufa1","type":"onoff","active":"ON","input":[],"output":[
        {"name":"reineu/ktest/gpio/OUT0","alias":"OUT0","status":"OFF"}]},
{"name":"Stufa2","type":"onoff","active":"ON","input":[],
  "output":[
        {"name":"reineu/ktest/gpio/OUT1","alias":"OUT1","status":"OFF"}]},
{"name":"Stufa3","type":"onoff","active":"ON","input":[],
 "output":[
        {"name":"reineu/ktest/gpio/OUT2","alias":"OUT2","status":"OFF"}]},
{"name":"Ventole","type":"onoff","active":"ON","input":[],
  "output":[
        {"name":"reineu/ktest/gpio/OUT3","alias":"OUT3","status":"OFF"}]}
]}
```

Richiesta per descrizione dei parametri per il funzionamento del servizio *CloudKmeter*:

```
to/reineu/ktest/kmeter/request {"request":"params"}
```

Risposta:

```
from/reineu/ktest/kmeter/params

{"reineu/ktest/kmeter": [
        {"name":"DT","category":"generic", "value":"6"},
        {"name":"MAXTEMP","category":"generic", "value":"40"},
        {"name":"MINTEMP","category":"generic", "value":"30"},
        {"name":"MINTIME","category":"generic", "value":"0"},
        {"name":"Stufa1","category":"heater", "value":"source"},
        {"name":"Stufa2","category":"heater", "value":"source"},
        {"name":"Stufa3","category":"heater", "value":"source"},
        {"name":"Ventole","category":"heater", "value":"fan"},
        {"name":"maxwindow","category":"generic", "value":"3600"},
        {"name":"persistency","category":"generic", "value":"120"},
        {"name":"pmax","category":"generic", "value":"400"},
        {"name":"pmin","category":"generic", "value":"0"},
        {"name":"starttime","category":"generic", "value":"1634748714.06601"},
        {"name":"status","category":"generic", "value":"OFF"},
        {"name":"surface","category":"generic", "value":"80"},
        {"name":"tmax","category":"generic", "value":"100"},
        {"name":"tmin","category":"generic", "value":"-50"},
        {"name":"tolerance","category":"generic", "value":"0.05"}
]}
```

Richiesta per descrizione dei parametri da monitorare durante sessione di *test*:

```
to/reineu/ktest/kmeter/request {"request":"description"}
```

Risposta:

```
from/reineu/ktest/kmeter/description

{"Kmeter/ktest/meter": [
        {"name":"power","type":"ElectricMeter","status":"ON", "value":0, "unit"
:"Wh", "time":"0"},
        {"name":"K","type":"AnalogMeter","status":"ON","value":0,"unit":"W/mq",
"time":"0"},
        {"name":"TE","type":"AnalogMeter", "status":"ON","value":0,"unit":"C","
time":"0"},
        {"name":"TI","type":"AnalogMeter","status":"ON","value":0.00,"unit":"C"
,"time":"0"}
]}
```

7 Ringraziamenti

Voglio ringraziare il Professor Giorgio Leonardi e il Professor Attilio Giordana per la disponibilità, il prezioso aiuto e la fiducia che hanno sempre riposto nel mio lavoro.

Ringrazio Gaia e Agnese per il continuo sostegno e per l'aiuto ricevuto. Ringrazio Martina, Angelo e Gloria per aver condiviso questo percorso e aver reso più leggero il tempo trascorso insieme.

Un ringraziamento speciale va alla mia famiglia che mi ha sempre sostenuto, con un pensiero al nonno: anche se non abbiamo fatto in tempo, sono certo che sei felice dei nostri traguardi.

Infine, un grazie di cuore a Francesca per essere sempre al mio fianco, per condividere i momenti più belli e starmi accanto nei momenti più difficili.