UNIVERSITÀ degli STUDI di CATANIA

Dipartimento di Matematica e Informatica
Corso di Laurea in Informatica

PROGETTO SISTEMI CLOUD E LABORATORIO

Davide Scalisi

PROGETTAZIONE DI UN SISTEMA SOFTWARE PER LA REGISTRAZIONE E L'INVIO DI DATI TELEMETRICI DI CONSUMI ENERGETICI TRAMITE INTERNET

Documentazione e progetto sono stati realizzati da Davide Scalisi (1000038316) per poter conseguire l'esame di Sistemi Cloud e Laboratorio del corso di laurea magistrale in informatica dell'anno accademico 2022/2023 con i docenti Giuseppe Pappalardo e Andrea Fornaia.

ANNO ACCADEMICO 2022/2023

Sommario

Introduzione

Lo scopo di questo progetto è quello di utilizzare i dati messi a disposizione dal dispositivo PowerMonitor da me realizzato. Questi dati telemetrici di consumi energetici verranno quindi utilizzati dai diversi attori del sistema al fine di ottenere determinati obbiettivi.

Repository GitHub del Progetto

- Smart meter: PowerMonitor
- Documentazione: CloudProject-Docs
- Sezione “Edge”: CloudProject-Edge
- Sezione “Blockchain”: CloudProject-Edge
- Sezione “Cloud”: CloudProject-Cloud (non utilizzata nella dimostrazione finale)
- Sezione “Vagrant”: CloudProject-Vagrant-Cloud (in sostituzione della precedente)

Struttura richiesta dell’architettura

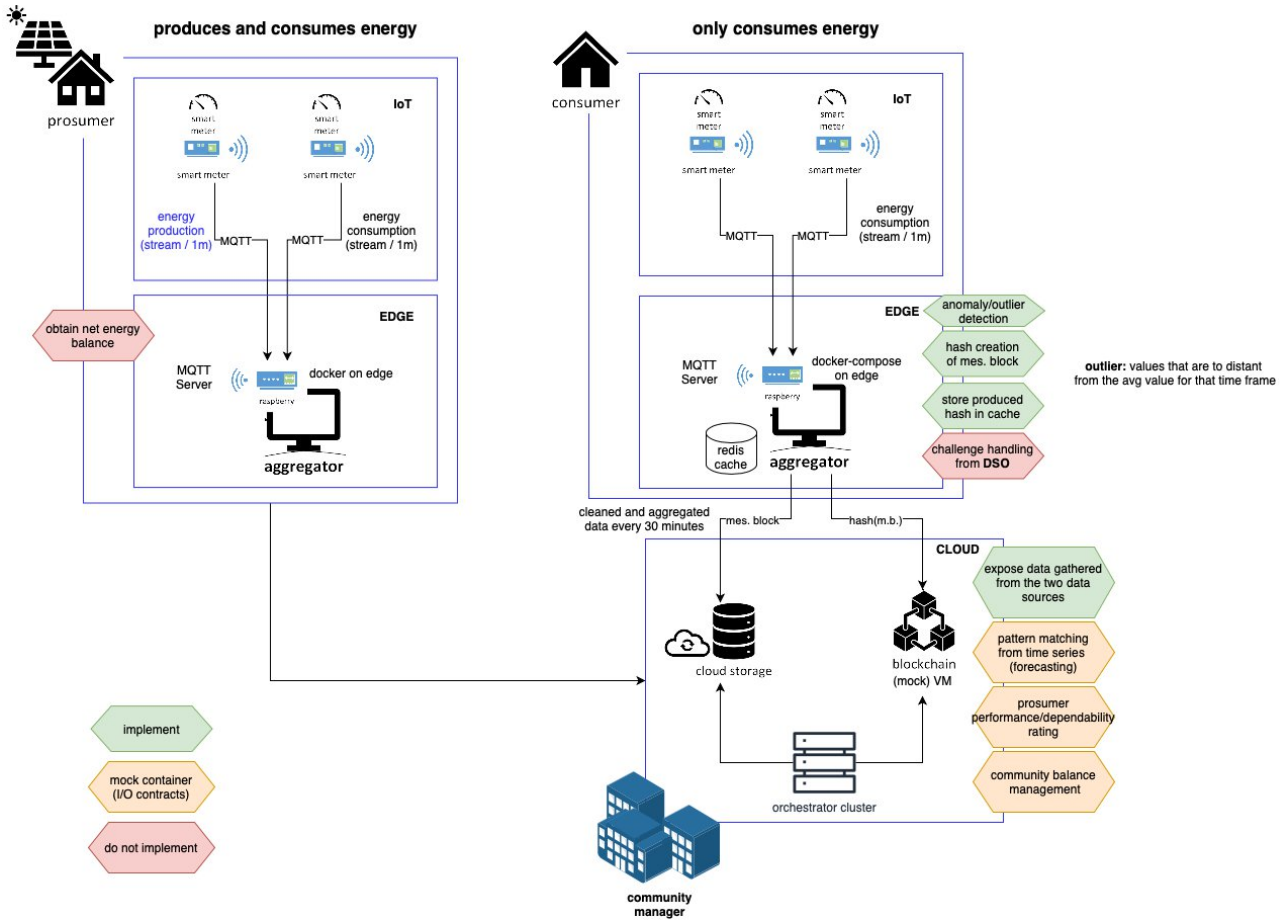

Figura 1: Schema assegnato dell'intera architettura software.

Tale struttura è stata appositamente descritta nel dettaglio dal punto di vista topologico. I vari servizi in gioco devono essere o implementati interamente (verde), implementati come dei mock (arancio) o non implementati, ma solamente descritti in tale architettura (rosso).

Sezione Edge

La sezione di elaborazione dati interna all'abitazione è denominata "Edge". Essendo gli elementi smart meter approfonditamente trattati nell'apposito documento referenziato nella precedente sezione, in questo capitolo ci si concentrerà principalmente nella descrizione dell'aggregatore e dei suoi compiti.

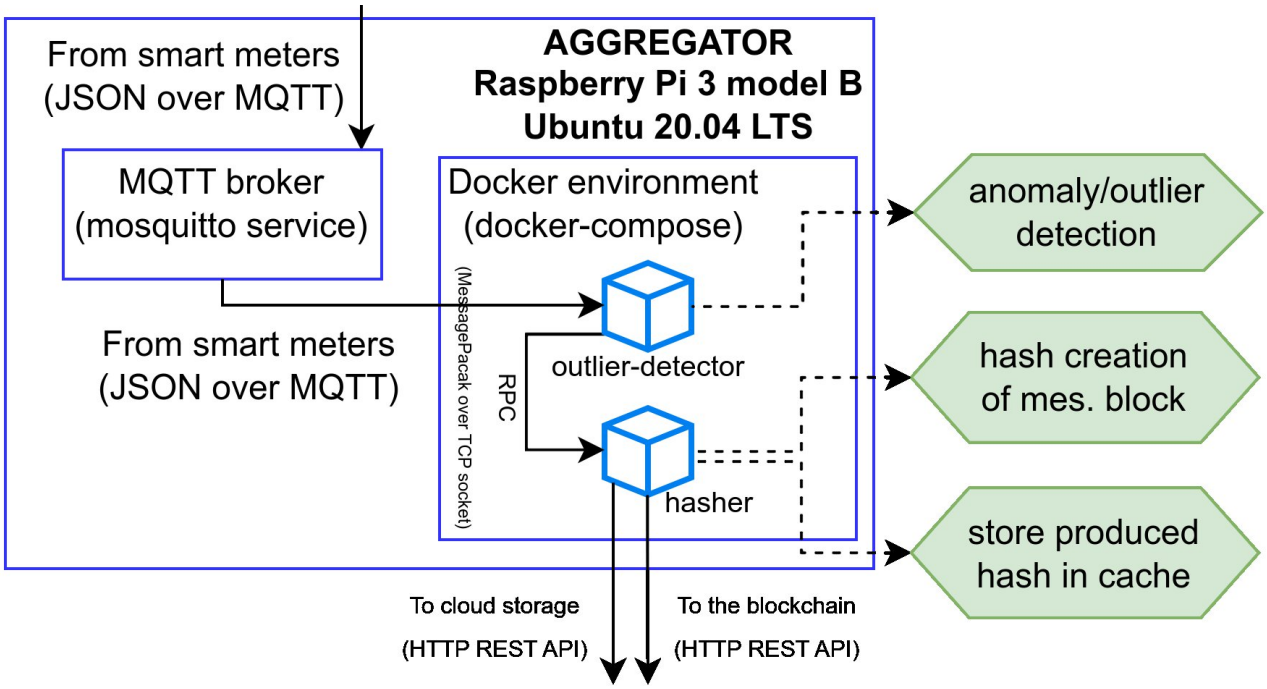

Figura 2: Implementazione sezione Edge.

L'implementazione dei microservizi richiesti avviene tramite la creazione di due immagini Docker chiamate outlier-detector ed hasher, le quali saranno utilizzate all'interno di un'ambiente Docker e tramite una descrizione architetturale realizzata tramite un docker-compose.

Connessione MQTT all'aggregatore

Nel dettaglio, i dati uscenti dal dispositivo smart meter, codificati tramite JSON, raggiungono l'aggregatore tramite il servizio di MQTT broker Mosquitto, il quale è installato ed in ascolto sulla porta 1883 come un servizio di systemctl.

Servizio outlier-detector

A questo punto, il servizio outlier-detector, il quale si è precedentemente sottoscritto al topic MQTT "/PowerMonitor/data", viene avvisato e riceve i suddetti dati JSON. Questi dati vengono in un primo momento filtrati, ricavando solamente i dati di potenza apparente (VA) e di potenza attiva (W) e successivamente viene eseguito un algoritmo di outlying detection, in modo tale da scartare un certo numero di campioni. I parametri dell'applicazione sono dinamicamente impostabili tramite la modifica delle variabili d'ambiente definite all'interno del file docker-compose.yml. Una volta raffinati i dati, questi vengono inviati ad un secondo thread all'interno dello stesso container outlier-detector (tramite una coda di messaggi protetta nativamente da sezione critica), il quale ha il compito di inviare a sua volta, tali dati raffinati al secondo container, hasher.

Servizio hasher

Sia il precedente microservizio che l'attuale, sono stati implementati utilizzando il linguaggio Python. Tramite quindi la libreria Python zerorpc (server RPC esposto in maniera predefinita sulla porta 4090), è stato possibile implementare la chiamata a procedura remota (RPC) con la quale i dati vengono ricevuti dal container outlyer-detector. Ricevuti questi dati, tale container è composto da due thread, il primo riceve i dati tramite RPC e li invia al secondo (utilizzando sempre una coda di messaggi protetta nativamente da sezione critica). Il secondo thread si occupa invece di molteplici compiti, tra cui:

1. Creazione di un particolare blocco dati JSON.
2. Salvataggio dell'hash (SHA-256) all'interno di un file locale di database.
3. Invio del blocco al servizio esterno di Blockchain (CloudProject-Blockchain).
4. Invio del blocco al servizio di Cloud storage (Openstack container)

Anche questo servizio è pienamente parametrizzato tramite il file docker-compose.yml. I dati escono dal microservizio codificati sottoforma di JSON e vengono trasportati tramite delle apposite chiamate ad API REST che il servizio di Blockchain ed il servizio di Cloud storage espongono. In particolare, per la connessione al Cloud storage gestito da Openstack, è stata utilizzata l'omonima libreria Python combinata con un apposito file "clouds.yaml" fornito dal Cloud provider (GARR). Questo file contiene i segreti necessari ad autorizzare le necessarie operazioni di scrittura che devono essere effettuate dal servizio hasher al container Openstack.

Sezione Blockchain

Il servizio di Blockchain, implementato tramite un mock, punta a simulare un comportamento tipico che potrebbe avere una Blockchain in vista di futuri potenziali interfacciamenti con reali servizi esterni.

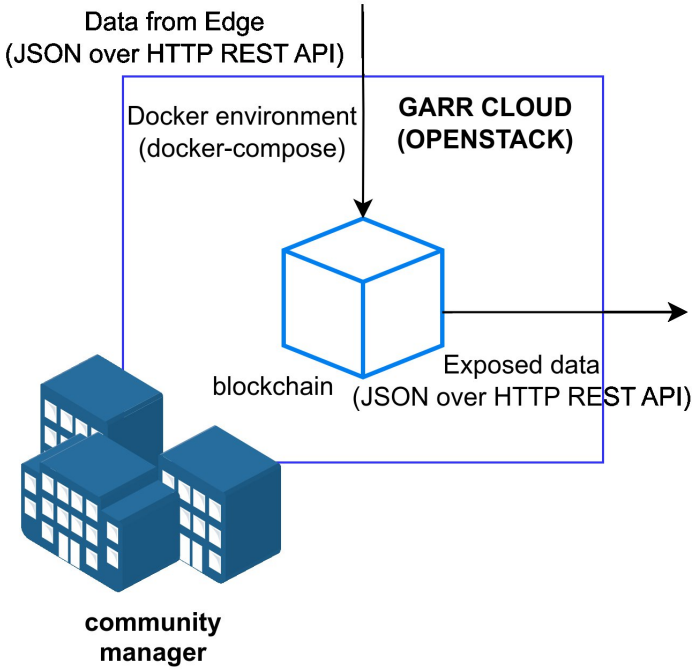

Figura 3: Implementazione sezione Blockchain (mock).

Il servizio Blockchain è stato scritto in Node.js, in quanto una delle librerie utilizzate è stata Express: una semplice libreria pensata per poter permettere in maniera semplice la scrittura di codice per la gestione di server HTTP utilizzando API REST e JSON. Una volta ricevuti i dati tramite l'endpoint esposto "/block/add" (server HTTP esposto in maniera predefinita sulla porta 5090), il microservizio containerizzato (infrastruttura descritta e parametrizzata allo stesso modo tramite un docker-compose.yml) ha la responsabilità di salvare in un file di database locale, l'hash comunicato del blocco insieme al suo timestamp. Il file locale è un semplice file JSON contenente un'array di dati e, per ogni dato, sono presenti il timestamp e l'hash del blocco corrispondente. Infine, il servizio di Blockchain espone l'endpoint "/block/get", il quale permette di ottenere tutti i dati che fino a quel momento sono stati salvati in locale dal servizio di Blockchain.

Servizio di Cloud storage

Come servizio di Cloud storage è stato utilizzato un container di Openstack, offerto come PaaS dall'infrastruttura cloud del GARR.

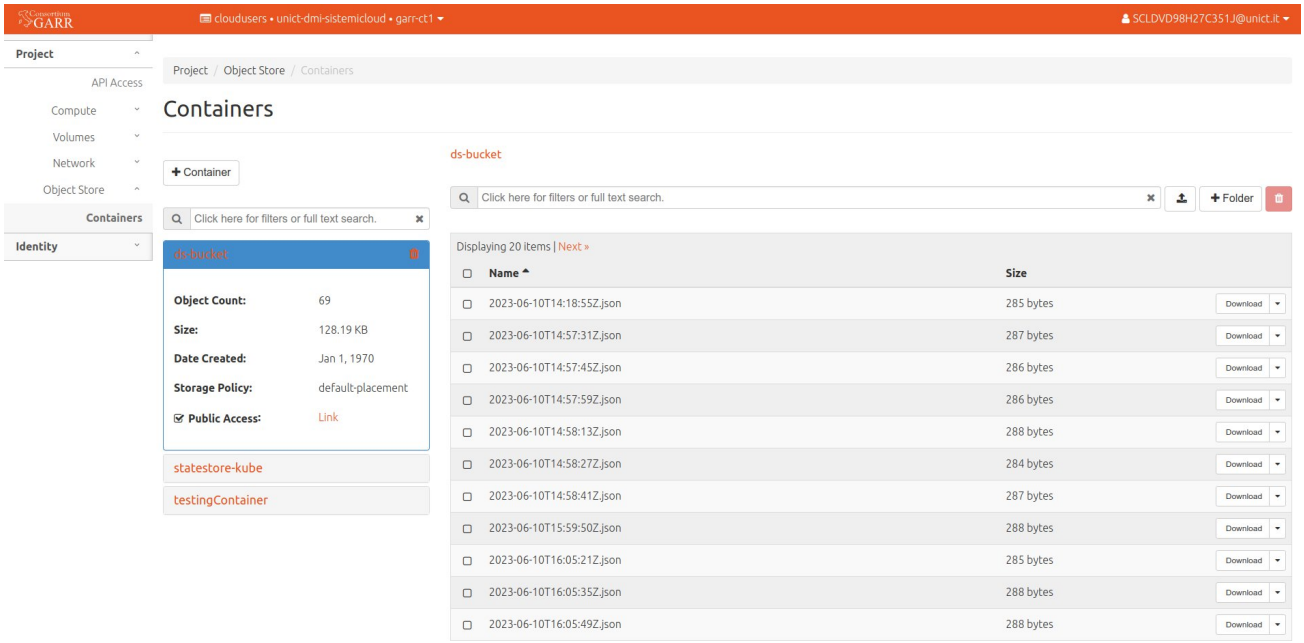

Figura 4: Portale di gestione del cloud GARR (Openstack).

Come detto precedentemente, l'autenticazione al cloud avviene tramite il file "clouds.yaml" scaricabile dal centro di controllo web offerto nel GARR Cloud. In dettaglio, il servizio utilizzato è denominato "Container Openstack" il quale non è altro che un file storage dove vengono inviati dai files provenienti dalla sezione Edge. I dati contenuti in questi files non sono altro che la codifica JSON precedentemente elaborata e munita di hash, calcolata precedentemente dalla sezione Edge. Non offrendo l'interfaccia web un ordinamento dei files per data, si è deciso di nominare i vari file JSON di telemetria con il loro corrispondente timestamp. Tali dati verranno successivamente recuperati dai servizi all'interno di un cluster Kubernetes per essere sottoposti ad ulteriori elaborazioni ed, infine, esposti tramite un front-end all'utente finale che ha la necessità di consultarli.

Sezione di cluster di elaborazione

Al fine di raffinare e di presentare i dati all'utente finale, è stato predisposto un cluster Kubernetes che provvede a rendere disponibili quattro microservizi per la gestione del front-end con l'utente finale. Per la scelta implementativa di tali microservizi (sempre sotto forma di immagini Docker) si è ancora una volta scelto di utilizzare Node.js in combinazione con la libreria Express. Lo sviluppo dell'infrastruttura di questa sezione è stata inizialmente la stessa della sezione di Blockchain, cioè una semplice IaaS su Cloud GARR. Tale tentativo è stato un successo fino all'esecuzione dei vari microservizi: sono infatti presenti dei problemi di configurazione delle macchine a livello di routing che non sembrano apparentemente essere risolvibili in maniera banale. Per tali motivi, il cluster ed anche il successivo sviluppo dell'applicazione sono stati spostati in locale, utilizzando appositamente un cluster Kubernetes formato da tre macchine virtuali descritte tramite il software Vagrant. Una macchina virtuale ha assunto il ruolo di nodo master del cluster, mentre le altre due hanno assunto i ruoli di due worker del cluster.

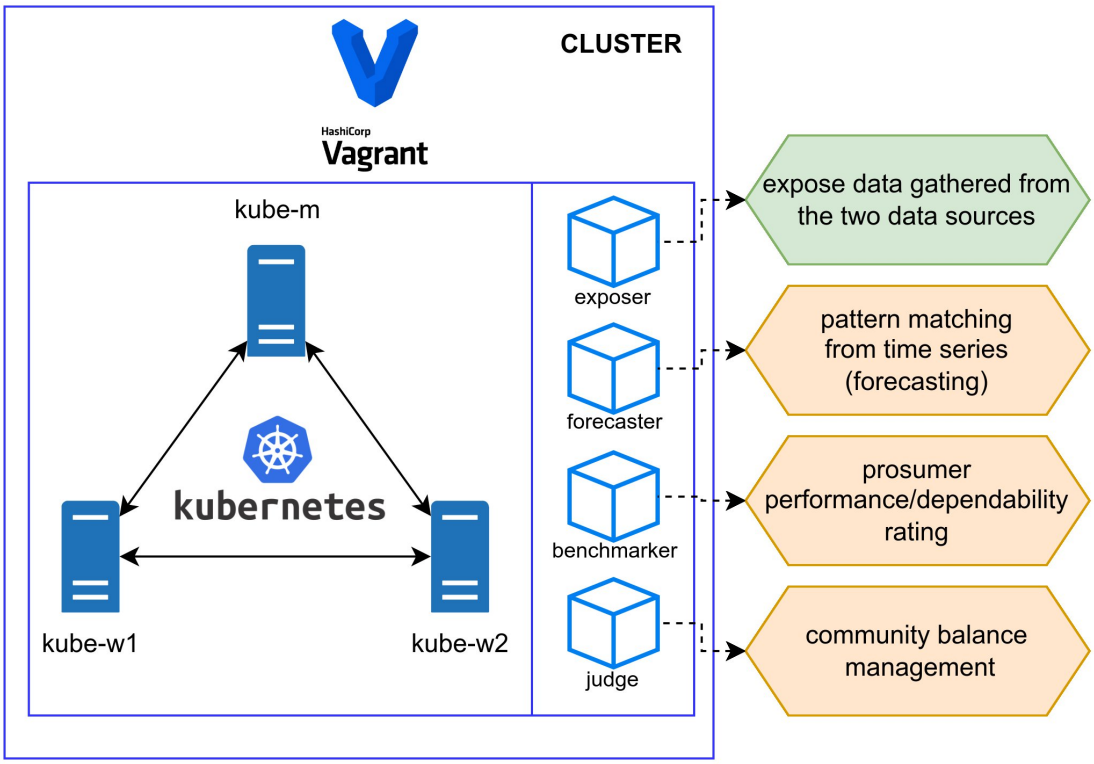

Figura 5: Implementazione sezione Cluster.

Caratteristiche dell'ambiente Kubernetes

Kubernetes espone i vari servizi tramite dei servizi di tipo LoadBalancers:

```
vagrant@kube-m:~/CloudProject-Vagrant-Cloud$ kubectl get services
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)          AGE
benchmarker-service  LoadBalancer   172.96.104.51    192.168.1.71   8081:30903/TCP   4d21h
exposer-service      LoadBalancer   172.96.205.246   192.168.1.71   80:31841/TCP     4d21h
forecaster-service   LoadBalancer   172.96.41.9      192.168.1.71   8080:30144/TCP   4d21h
judge-service        LoadBalancer   172.96.48.160    192.168.1.71   8082:30363/TCP   4d21h
```

Figura 6: Servizi Kubernetes esposti all'esterno del cluster.

Mentre ogni deployment è parametrizzabile per poter essere replicato un numero arbitrario di volte:

```
vagrant@kube-m:~/CloudProject-Vagrant-Cloud$ kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
benchmarker   4/4     4            4           4d21h
exposer       4/4     4            4           4d21h
forecaster    4/4     4            4           4d21h
judge         4/4     4            4           4d21h
```

Figura 7: Deployments in esecuzione all'interno del cluster.

Per ogni replica del deployment, Kubernetes crea automaticamente un nuovo pod, il quale viene automaticamente orchestrato su un nodo diverso del cluster, in modo da garantire fault tollerance ed HA:

```
vagrant@kube-m:~/CloudProject-Vagrant-Cloud$ kubectl get pods
NAME                           READY   STATUS    RESTARTS        AGE
benchmarker-665dcc54d9-87tvq   1/1     Running   1 (3h14m ago)   4d21h
benchmarker-665dcc54d9-jksxt   1/1     Running   1 (3h13m ago)   4d21h
benchmarker-665dcc54d9-vmw6x   1/1     Running   1 (3h13m ago)   4d21h
benchmarker-665dcc54d9-xtrs4   1/1     Running   1 (3h14m ago)   4d21h
exposer-5565fb9556-btxcs       1/1     Running   1 (3h14m ago)   4d21h
exposer-5565fb9556-fqmx5       1/1     Running   1 (3h14m ago)   4d21h
exposer-5565fb9556-m4z8q       1/1     Running   1 (3h13m ago)   4d21h
exposer-5565fb9556-szvxr       1/1     Running   1 (3h13m ago)   4d21h
forecaster-6c4d479849-97cvj    1/1     Running   1 (3h14m ago)   4d21h
forecaster-6c4d479849-dmhxz    1/1     Running   1 (3h13m ago)   4d21h
forecaster-6c4d479849-trjdb    1/1     Running   1 (3h14m ago)   4d21h
forecaster-6c4d479849-vh7qn    1/1     Running   1 (3h13m ago)   4d21h
judge-6b6bc9d4d6-4lj27         1/1     Running   1 (3h14m ago)   4d21h
judge-6b6bc9d4d6-f2xwq         1/1     Running   1 (3h13m ago)   4d21h
judge-6b6bc9d4d6-gjhnf         1/1     Running   1 (3h13m ago)   4d21h
judge-6b6bc9d4d6-r84xj         1/1     Running   1 (3h14m ago)   4d21h
```

Figura 8: Con quattro repliche per ognuno dei quattro deployment, si ottengono 16 pods.

Servizio exposer

Questo servizio si limita ad aggregare in un array JSON, i dati del Cloud storage tramite l'endpoint HTTP "/container/block/get" (porta 8080), per poi esporli successivamente all'utente finale. Il secondo compito di questo servizio è quello di prelevare il database della Blockchain ed esporlo sotto l'endpoint "/blockchain/block/get".

Servizio forecaster

Questo servizio insieme ai successivi due, è implementato come un mock. Il suo compito è quello di esporre un server HTTP sulla porta 8081 che tramite l'interrogazione da parte di un client tramite un endpoint della forma "/forecast/nome_file.json", permette di ritornare al richiedente l'hash, il timestamp e la media dei due array di campioni appartenenti al file considerato. Tale file dovrà anche essere presente nel Cloud storage descritto precedentemente.

Servizio benchmarker

Il servizio espone sulla porta 8082, un server HTTP che risponde su un endpoint della forma "/blockchain/bench/get/user_id". Il parametro user_id è una stringa che rappresenta idealmente l'ID di un certo utente consumer/prosumer. Da tale parametro, il servizio ricava quindi un certo grado di rating corrispondente a tale ID utente. Il valore di ritorno finale di questa richiesta è quindi un numero intero da 1 a 10 (inclusi) che rappresenta il rating associato a quel rispettivo utente.

Servizio judge

Esponendo un ultimo web server HTTP sulla porta 8083, tale servizio mette a disposizione l'endpoint "/blockchain/decide/user_id" che, utilizzando il meccanismo di rating del servizio precedente, rifiuta o accetta l'utente nel sistema in base ad una logica applicativa interna.

Conclusioni

L'intero progetto è stato infine un successo nonostante le molteplici difficoltà incontrate nel corso dello sviluppo delle tre fasi. Il risultato finale è quindi quello desiderato inizialmente: un'architettura variegata e basata su sistemi Cloud espandibile sotto tutti i punti di vista e funzionante in rete locale e/o attraverso la rete internet.