Progettazione e realizzazione di un sistema distribuito per il monitoraggio ambientale dei parchi comunali

Ludovico Di Iorio
0336019
ludovico.diiorio@students.uniroma2.eu

I. Abstract

Il presente lavoro descrive la progettazione e la realizzazione di un sistema distribuito, che sfrutta il pattern dei microservizi e dell'edge computing. Per la realizzazione sono state utilizzate le tecnologie di Kubernetes (Minikube e k3s) e i servizi offerti da AWS (EC2 e Dynamic IP), oltre che differenti linguaggi di programmazione (Go, Python, Js).

II. Introduction

Il monitoraggio delle condizioni ambientali dei parchi comunali è fondamentale per le amministrazioni locali che necessitano di dati aggiornati per ottimizzare la manutenzione e garantire la qualità degli spazi verdi urbani.

Il presente progetto sviluppa un sistema per il monitoraggio ambientale dei parchi comunali del comune di Aprilia (LT) basato su un'architettura edge-cloud. Il sistema è composto da dispositivi edge dotati di sensori per il rilevamento delle condizioni climatiche (temperatura, umidità, luminosità e qualità dell'aria) che raccolgono periodicamente dati ambientali e li trasmettono a un server centralizzato per l'elaborazione e consentirne la visualizzazione.

Per garantire realismo nella simulazione, i dispositivi edge sono implementati come macchine virtuali con k3s per la gestione dei microservizi in Python. I dati raccolti vengono sottoposti a controlli di validità prima della trasmissione, evitando l'invio di informazioni ridondanti o invalide. Il server è ospitato su un'istanza EC2 di AWS e utilizza Minikube per gestire un cluster Kubernetes che orchestrare i microservizi di back-end (sviluppati in Go) e front-end (HTML+JavaScript). La persistenza dei dati è gestita con un database relazione e la visualizzazione dei dati in tempo reale tramite un'interfaccia web.

L'architettura adottata dimostra l'integrazione efficace di edge computing, containerizzazione e servizi cloud, applicando i principi fondamentali dei sistemi distribuiti per creare una soluzione scalabile e affidabile per il monitoraggio ambientale urbano.

III. Background

Il progetto si colloca nell'ambito dei sistemi distribuiti e del cloud computing, due paradigmi fondamentali nello sviluppo di applicazioni moderne scalabili e modulari. L'obiettivo è integrare tecnologie edge, microservizi, container e orchestrazione, per simulare un ambiente realistico in cui dati ambientali vengono raccolti, pre-elaborati e visualizzati tramite un'infrastruttura distribuita.

Uno degli elementi chiave è il concetto di edge computing, che prevede l'elaborazione dei dati direttamente nei dispositivi periferici vicini alla fonte, riducendo la latenza e il traffico verso il server nel cloud. Nel contesto di questo progetto, i dispositivi edge sono simulati da macchine virtuali leggere e gestite tramite k3s, una distribuzione semplificata di Kubernetes adatta a dispositivi con risorse limitate. Ogni dispositivo simula la raccolta dati da sensori ambientali e invia i risultati, in modo intelligente, al server centrale.

La comunicazione tra dispositivi edge e backend avviene tramite gRPC, un moderno sistema di Remote Procedure Call sviluppato da Google, che consente la trasmissione efficiente di messaggi binari su HTTP/2. gRPC è stato scelto per la sua leggerezza e per il supporto nativo alla serializzazione tramite Protocol Buffers, ideale per scenari IoT e real-time.

L'infrastruttura centrale è basata su Kubernetes, eseguito tramite Minikube all'interno di una macchina virtuale EC2 di AWS. Il cluster ospita i microservizi per il backend e per il frontend, entrambi containerizzati con Docker, garantendo isolamento, scalabilità e semplicità di deployment. Il backend è sviluppato in linguaggio Go per sfruttare le sue capacità di concorrenza, mentre il frontend è un'applicazione web statica servita tramite un container Nginx.

Per la persistenza dei dati è stato scelto MariaDB, un database relazionale compatibile con MySQL, eseguito in un container Docker su un'istanza EC2 separata. Questa architettura distribuita consente una separazione delle responsabilità e una maggiore flessibilità gestionale.

IV. System Architecture

Il sistema è composto da cinque componenti:

A. *Dispositivi edge*

Sui dispositivi edge è presente un cluster k3s per la gestione di un microservizio, sviluppato in Python, che simula i dati ambientali raccolti da un sensore. Ogni edge effettua un pre-processamento locale ed invia i dati al backend tramite gRPC. Nel pre-processamento avviene anche un controllo di validità dei dati e rilevamento di variazioni significative rispetto agli ultimi valori raccolti.

B. *Backend*

Per il server cloud sono stati sviluppati 2 microservizi in Go, uno che consente di ricevere le misure raccolte dai dispositivi edge, tramite gRPC, mentre l'altro microservizio serve per esporre delle API REST per il frontend, attraverso cui vengono fornite le misure aggiornate e i dati storici all'utente.

C. *Frontend*

Il servizio per il frontend è stato realizzato a partire da un server Nginx, che restituisce le risorse web (HTML+Javascript). Comunica con il backend tramite richieste periodiche e mostra i dati in tempo reale, sia in forma testuale che tramite grafici.

D. *Database*

Il database si trova su un'istanza EC2 separata ed è basato su MariaDB che è un sistema di gestione di database relazionali.

E. *Admin*

L'amministratore del sistema avrà un programma dedicato con il quale può inserire, rimuovere parchi e sensori, oltre ad effettuare l'associazione e la dissociazione dei sensori dai parchi.

I componenti del backend e del frontend sono containerizzati e orchestrati all'interno di un cluster Minikube installato su un'istanza EC2, ed ogni microservizio è contenuto in un pod distinto.

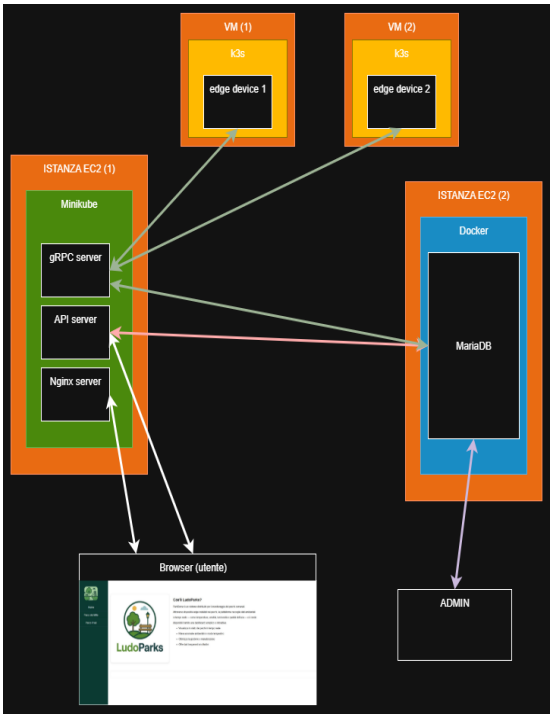


diagramma dell'architettura del sistema

V. Implementation

A. *Dispositivi edge*

I dispositivi edge, installati nei parchi comunali, sono responsabili della raccolta, pre-elaborazione e invio dei dati ambientali. Sono realizzati come microservizi Python eseguiti su macchine virtuali e orchestrati con k3s.

Per ogni dispositivo edge è stata usata un'istanza EC2, sulla quale è stato appunto installato k3s, il quale gestisce il microservizio come un pod del cluster. L'utilizzo di k3s, nonostante ci sia un solo microservizio attivo comporta diversi vantaggi:

- Simulazione più realistica: nei dispositivi edge reali molto spesso è utilizzato k3s;
- Monitoraggio e logging integrato: i controlli sullo stato di salute del sistema sono supportati e automatizzati da k3s;
- Resilienza: riavvio automatico in caso di crash del container;

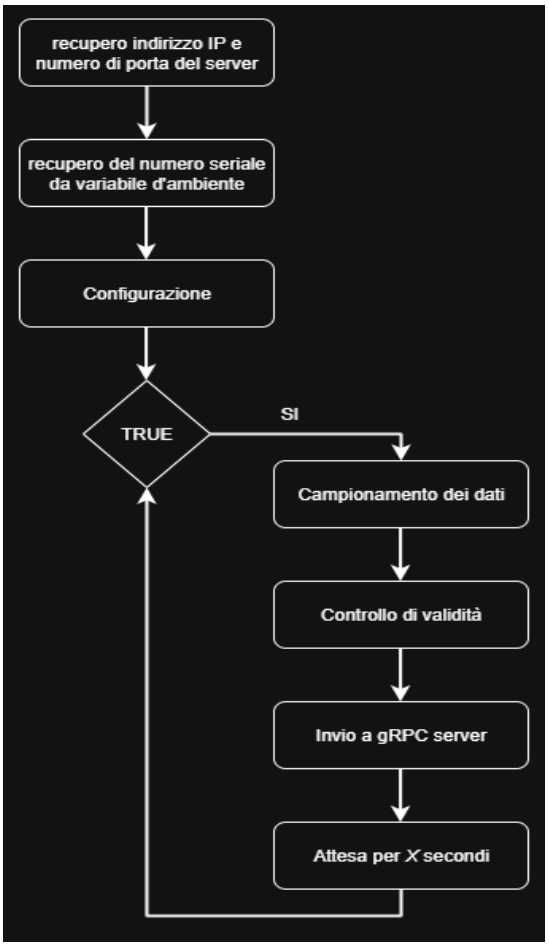
- Scalabilità futura: k3s facilita l'aggiunta di nuovi microservizi sullo stesso edge e la replicazione del servizio;
- Isolamento: il microservizio è in esecuzione in uno spazio virtualizzato nell'host;
- Portabilità: il deployment funziona identicamente su diversi dispositivi edge e ciò rende facilitata la migrazione e la replica dell'ambiente.

Nella fase di avvio, il dispositivo recupera dal file di configurazione le informazioni necessarie per potersi collegare al server, ovvero l'indirizzo IP e il numero di porta; poi recupera dalla variabile d'ambiente SERIAL_NUMBER il numero seriale univoco associato al dispositivo, il quale è definito con un componente ConfigMap (che serve per dichiarare le variabili d'ambiente) del cluster k3s.

Fatto ciò, viene creata un'istanza della classe *Sensor*, nel cui costruttore avviene l'inizializzazione dei parametri e una prima configurazione: il dispositivo contatta il server cloud (tramite gRPC) inviandogli il proprio numero seriale, per segnalarsi come attivo e recuperare l'id del parco a cui è associato, il proprio id (i quali serviranno successivamente per identificare le misure raccolte) e l'intervallo di tempo che intercorre tra una misurazione e la successiva.

Dopodiché si entra all'interno di un loop infinito, nel quale si ripetono le seguenti operazioni:

- *Recupero delle misure*: vengono generate delle misurazioni (tramite AdvancedSensorSimulator.py) e impacchettate, pronte per essere inviate.
- *Controllo validità*: i dati prima di essere inviati passano un controllo di validità, ovvero si verifica che i dati raccolti siano validi (non ci siano valori insensati), e che non siano ridondanti (la nuova misurazione deve esprimere un cambiamento significativo rispetto all'ultima misurazione), perché nel caso non fossero validi non verrebbero inviati, e ciò permetterebbe di risparmiare risorse del server.
- *Invio dei dati*: i dati sono validi per cui sono inviati al server tramite una comunicazione asincrona gRPC.



Edge flowchart

Il servizio, per poter girare su k3s, è stato prima containerizzato tramite Docker, per poi essere pushato su una repository di Docker Hub, perchè ciò facilita il recupero dell'immagine. L'ambiente su k3s è stato settato tramite un file di deployment (edge.yaml) che serve per generare il container dentro il cluster di k3s, e un file per il Config Map (edge-cm.yaml) che è servito per definire le variabili d'ambiente dentro il container.

B. Backend

Il backend rappresenta il nucleo logico dell'intero sistema distribuito. È stato sviluppato in linguaggio Go per la sua efficienza nella gestione concorrente e per la facilità di integrazione con servizi gRPC e RESTful. Le sue principali responsabilità sono la ricezione e gestione dei dati inviati dai dispositivi edge, il salvataggio delle misurazioni nel database relazionale e l'esposizione di API HTTP per l'interfaccia web.

Il backend è stato implementato con 2 microservizi distinti: uno per le comunicazioni con gli edge (gRPC server) ed uno per le

comunicazioni con i browser (API server). Tale scelta è stata veicolata dalla volontà di rendere i due servizi indipendenti, così da aumentarne la disponibilità; infatti, se uno dei due microservizi fallisse, non avrebbe ripercussioni sull'altro. Ognuno dei due microservizi è eseguito all'interno di un container e successivamente integrato come pod nel cluster Minikube e sono resi disponibili all'esterno tramite un componente Service del cluster, di tipo NodePort (serve per associare le porte dei pod a quelle del container di Minikube).

gRPC server: questo microservizio si occupa di mettersi in ascolto sulla porta 50051, in attesa di messaggi da parte dei dispositivi edge, quindi sia messaggi per la configurazione, sia messaggi contenenti le misurazioni raccolte.

Se riceve un messaggio di configurazione, il server dovrà recuperare dal database l'id del sensore e del parco a cui è stato associato, ma prima di inviarglieli viene modificato lo status del sensore e del parco, i quali sono rispettivamente impostati come 'attivo' e 'osservato'.

Se invece, il messaggio ricevuto è l'insieme delle misurazioni raccolte dal dispositivo edge, alla ricezione viene avviata una goroutine per eseguire l'inserimento dei dati nel database: in questo modo il server non resta bloccato durante le operazioni di inserimento dei dati e riesce a gestire simultaneamente più dispositivi edge attivi, rendendo il sistema scalabile e performante.

API server: l'altro microservizio si occupa di esporre un set di API REST utilizzate dal browser per recuperare i dati in formato JSON.

Precisamente sono esposte due API sulla porta 8080:

- */api/hello:* per ogni parco viene recuperata la misurazione più recente di ogni metrica, e sono anche recuperate le informazioni di temperatura massima e minima dei sette giorni precedenti
- */api/getData:* serve per recuperare le misurazioni avvenute durante la giornata odierna per una specifica metrica, le quali saranno poi usate per realizzare un grafico

La logica interna delle API consiste nella lettura dei parametri di query, nell'esecuzione di interrogazioni SQL sul database e nella serializzazione dei risultati in formato JSON.

Uno degli aspetti fondamentali del backend è la capacità di gestire molteplici richieste contemporaneamente, provenienti da dispositivi edge o dal frontend. Questo è reso possibile dall'uso delle goroutine di Go, che permettono un'esecuzione concorrente e leggera. Ogni richiesta gRPC ricevuta avvia una nuova goroutine per il salvataggio dei dati, così come ogni richiesta REST viene servita in modo indipendente. Questo modello consente al backend di mantenere alte prestazioni e tempi di risposta ridotti anche in condizioni di carico.

Per quanto riguarda le operazioni CRUD, per evitare problemi di inconsistenza causati da fallimenti, esse sono state implementate come transizioni; per cui se una funzionalità che comprende più operazioni CRUD dovesse fallire durante la sua esecuzione, non si avrebbero problemi di inconsistenza.

C. Frontend

Il frontend dell'applicazione costituisce l'interfaccia grafica accessibile agli utenti e rappresenta il punto di visualizzazione dei dati ambientali raccolti dai dispositivi edge. È stato progettato per essere semplice, chiaro e responsive, offrendo una panoramica in tempo reale dello stato dei parchi comunali monitorati.

Il frontend è containerizzato all'interno di un'immagine Docker basata su Nginx, che ha il compito di servire i file statici HTML, CSS e JavaScript. Questa immagine è poi eseguita come pod all'interno del cluster Minikube, esattamente come per i microservizi del backend.

Per rendere il frontend accessibile anche dall'esterno (es. per accedervi da browser su internet), è stato configurato un Service di tipo NodePort che instrada il traffico HTTP entrante verso il pod contenente Nginx.

L'inclusione del server Nginx nel cluster si basa su una logica di coerenza architetturale: tutte le principali componenti dell'applicazione sono orchestrate all'interno dello stesso ambiente controllato, favorendo una gestione centralizzata e semplificata dell'infrastruttura.

Inoltre, mantenere il frontend all'interno del cluster rende più agevole il deployment dell'intera applicazione: il sistema può essere avviato tramite pochi comandi (kubectl apply) utilizzando file YAML, che definiscono le immagini per i pod, il loro numero e i componenti Service. Questa modalità semplifica l'automazione del rilascio e facilita

la replicabilità dell'ambiente, sia in fase di sviluppo che in ambienti di test o produzione.

Dal punto di vista della scalabilità, anche il servizio web può beneficiare delle funzionalità offerte da Minikube. Qualora l'applicazione venisse utilizzata in un contesto reale con numerosi accessi simultanei, sarebbe possibile scalare automaticamente i pod che eseguono Nginx, garantendo una distribuzione efficiente del carico.

D. Database

Il sistema fa uso di un database relazionale per la persistenza dei dati raccolti dai dispositivi edge e delle informazioni relative ai parchi e ai sensori. Nello specifico, è stato adottato MariaDB, una scelta motivata dalla sua leggerezza, compatibilità con MySQL e semplicità di gestione. Il database è stato posizionato su una macchina virtuale EC2 dedicata, separata da quella che ospita Minikube.

All'interno di quest'istanza EC2, il database viene eseguito come contenitore Docker. Tale scelta è stata fatta per garantire maggiore isolamento, portabilità e facilità di deploy, sfruttando i vantaggi offerti dalla containerizzazione. Il backend, che si trova all'interno del cluster Minikube, comunica con il database esterno tramite il suo indirizzo IP pubblico (Elastic IP) e una porta esplicitamente esposta dal container Docker.

Il modello relazionale è stato progettato per essere semplice ma efficace. Le tabelle sono:

- ***Parks:*** contiene le informazioni anagrafiche sui parchi monitorati (id, name, location, is_observed).
- ***Sensors:*** rappresenta i sensori installati nei parchi, con un riferimento al parco associato, lo stato di attivazione e un identificativo univoco (id, serial_number, is_active, park_id).
- ***Measures:*** registra tutte le rilevazioni dei sensori, con i valori delle metriche raccolte e il relativo timestamp (id, sensor_id, park_id, temperature, humidity, brightness, air_quality, timestamp).

Le operazioni di scrittura nel database sono effettuate dal backend in modo concorrente, ma sicuro, sfruttando le goroutine di Go. MariaDB, da parte sua, gestisce correttamente la concorrenza in lettura e scrittura tramite transazioni e lock automatici, garantendo la coerenza dei dati.

E. Admin

Per l'amministratore del sistema è disponibile un programma separato che gli consente di eseguire le seguenti operazioni:

- Mostrare i sensori registrati
- Mostrare i parchi registrati
- Aggiungere un sensore
- Aggiungere un parco
- Rimuovere un sensore
- Rimuovere un parco
- Associare un sensore ad un parco
- Disassociare un sensore da un parco

Le operazioni CRUD eseguite dall'amministratore sono eseguite come transizioni per evitare conflitti e problemi di inconsistenza.

VI. DEPLOYMENT

Il processo di deployment dell'intero sistema è stato progettato per riflettere un'architettura cloud-native, sfruttando container, orchestrazione tramite Kubernetes e macchine virtuali fornite da AWS. L'obiettivo era distribuire i diversi componenti dell'applicazione in modo modulare, scalabile e isolato.

Distribuzione delle componenti:

 Cluster **Minikube** (su EC2): il cuore dell'applicazione è un cluster Kubernetes basato su Minikube, eseguito all'interno di un'istanza EC2 di AWS. Al suo interno sono stati collocati tre microservizi:

- **API-server** (Go): si occupa della gestione delle API REST per il frontend e del recupero di informazioni dal database.
- **gRPC-server** (Go): si occupa della ricezione dei dati dai dispositivi edge, dell'accesso al database.
- **Frontend** (HTML, JS, CSS): è servito da un container con **Nginx** e offre l'interfaccia web per la visualizzazione dei dati.

 Dispositivi **Edge (su EC2):** sono eseguiti su macchine virtuali e orchestrati tramite **k3s**. Ogni edge device simula la raccolta dei dati da

sensori, li pre-processa e li invia periodicamente al server tramite gRPC.

 Database **(MariaDB):** risiede su un'istanza EC2, separata dal cluster Kubernetes, ed è eseguito all'interno di un container Docker. Questa scelta garantisce una separazione logica e operativa tra la persistenza dei dati e i componenti applicativi.

Tutti i microservizi (frontend, backend ed edge) sono stati prima containerizzati usando **Docker**, e poi le immagini sono state poi caricate su Docker Hub, ovvero un sistema di repository per immagini Docker.

Ogni servizio è descritto tramite un proprio Dockerfile, il quale definisce le istruzioni necessarie per costruire l'immagine del container. Le immagini sono poi dichiarate all'interno di cluster tramite un file Manifest (.yaml) per creare i pod corrispondenti.

Il cluster è stato gestito tramite file YAML che definiscono:

- i **Deployment**, per specificare il numero di repliche e l'immagine da usare;
- i **Service** (uno per ogni deployment), necessari per l'esposizione dei pod su internet;
- eventuali **ConfigMap**, per gestire parametri di configurazione.

Questa struttura consente di avviare, aggiornare o ri-deployare l'intera infrastruttura tramite semplici comandi (kubectl apply -f), favorendo l'automazione e la riproducibilità del sistema.

VII. DISCUSSION

Il progetto realizzato rispecchia quella che era la richiesta, ovvero realizzare un sistema che sfruttasse l'edge computing, i microservizi e la tecnologia di Kubernetes, con le sue varianti.

Durante la realizzazione del progetto sono emersi diversi aspetti significativi, sia positivi che critici, che meritano una riflessione.

Uno dei principali punti di forza del sistema è la **modularità**. L'adozione di un'architettura microservizi ha permesso di separare logicamente le responsabilità dei diversi componenti, semplificando il testing, la manutenzione e il deployment. Inoltre, la containerizzazione di ciascun modulo ha facilitato lo sviluppo locale e la successiva integrazione in un cluster Kubernetes, simulando uno scenario produttivo realistico.

La scelta di utilizzare **gRPC per la comunicazione tra edge e backend** si è rivelata vincente in termini di leggerezza, efficienza e chiarezza nella definizione delle interfacce. Tuttavia, ha richiesto una fase iniziale di apprendimento non banale, soprattutto per la corretta gestione dei file .proto e la sincronizzazione dei client/server in linguaggi differenti (Python e Go).

Uno dei principali aspetti affrontati è la **gestione della concorrenza nel backend**, specialmente nel contesto dell'inserimento asincrono dei dati nel database. Il linguaggio di programmazione Go ha semplificato molto tale problema, poichè grazie alle goroutine gli accessi avvengono in maniera sequenziale non causando problemi.

D'altro canto, il progetto può anche essere migliorato sotto alcuni aspetti. La **sicurezza** delle comunicazioni, poiché non sono state implementate accortezze per garantire una comunicazione criptata e sicura. Poi il sistema potrebbe essere migrato su un vero **cluster Kubernetes** che consentirebbe di sfruttare a pieno i vantaggi offerti dalla distribuzione del sistema in più nodi.