IOT Devices Simulation

Un applicazione con architettura a "microservizi" per la simulazione di dispositivi IoT.

Riccardo Maria Pesce

Introduzione

L'obiettivo dell'applicazione è simulare il monitoraggio, la gestione, il campionamento e l'analisi di dati provenienti da dispositivi IoT. Per fare ciò, l'applicazione è suddivisa in tre microservizi comunicanti in maniera asincrona, ognuno dei quale possiede un proprio database atto a memorizzare i dati di interesse per il singolo microservizio.

Scelte progettuali

Per realizzare i microservizi, si è voluto utilizzare il linguaggio Python con il framework **FastAPI** che ci permette di realizzare servizi REST in maniera molto veloce ed agevole.

Il simulatore comunica col microservizio di recording utilizzando il protocollo *MQTT*, sebbene i servizi comunichino attraverso *Kafka*. Questa scelta deriva dalla necessità di simulare l'ambiente IoT, e sebbene il simulatore sia stato implementato come una API call appartenente al servizio di Monitoring & Management, in realtà essa simula i dispositivi IoT stessi, indipendentemente dal microservizio in cui tale funzione è implementata.

Di consueto, i microservizi comunicano attraverso *Kafka* in maniera asincrona.

Di seguito sono presenti le descrizioni dei tre microservizi con le relative scelte progettuali in dettaglio.

Microservizi

Monitoring & Management Microservice

Tale microservizio implementa il sistema di monitoraggio e gestione dei dispositivi IoT. Esso di appoggia a MongoDB in modo da poter memorizzare i dispositivi e le misure gestite nel comodo ed agevole formato JSON (per cui MongoDB eccelle come database). Attraverso questo servizio è possibile usare le consuete operazioni **CRUD** sia sui dispositivi (collection *devices*) che sulle misure (collection *measures*).
Oltre tali operazioni, come già detto nell'introduzione, è possibile utilizzare un API per simulare il vero e proprio campionamento delle differenti misure da parte dei dispositivi

presenti, in modo che il valore campionato stesso venga inviato attraverso *MQTT* al microservizio *Recording Microservice* (di cui sotto).
Il topic alla quale tale microservizio è interessato è il topic *device_commands* con il quale, in caso di malfunzionamento di un dispositivo, quest'ultimo viene automaticamente spento (e questa sarà la nostra transazione distribuita, dato che coinvolge tutti e tre i microservizi).

Recording Microservice

Tale microservizio serve il compito di acquisire le misure dai dispositivi IoT, memorizzarle attraverso transazioni *ACID* in un database locale (PostgreSQL), ed inviarle al microservizio di analisi (*Analytics Microservice*). A tale servizio è delegato il compito di iniziare la transazione distribuita che ha il compito di portare allo stato di spento il dispositivo che ha causato l'allarme (che ha mostrato, in una misurazione, uno stato di salute negativo, opportunamente simulato con l'API). PostgreSQL è stata la scelta, sebbene si potesse pensare di utilizzare Timescale. Ma siccome il nostro fine in tale servizio non è l'analisi ma il puro e solo recording transazionale, allora abbiamo focalizzato la nostra scelta su questo database open source.

Analytics Microservice

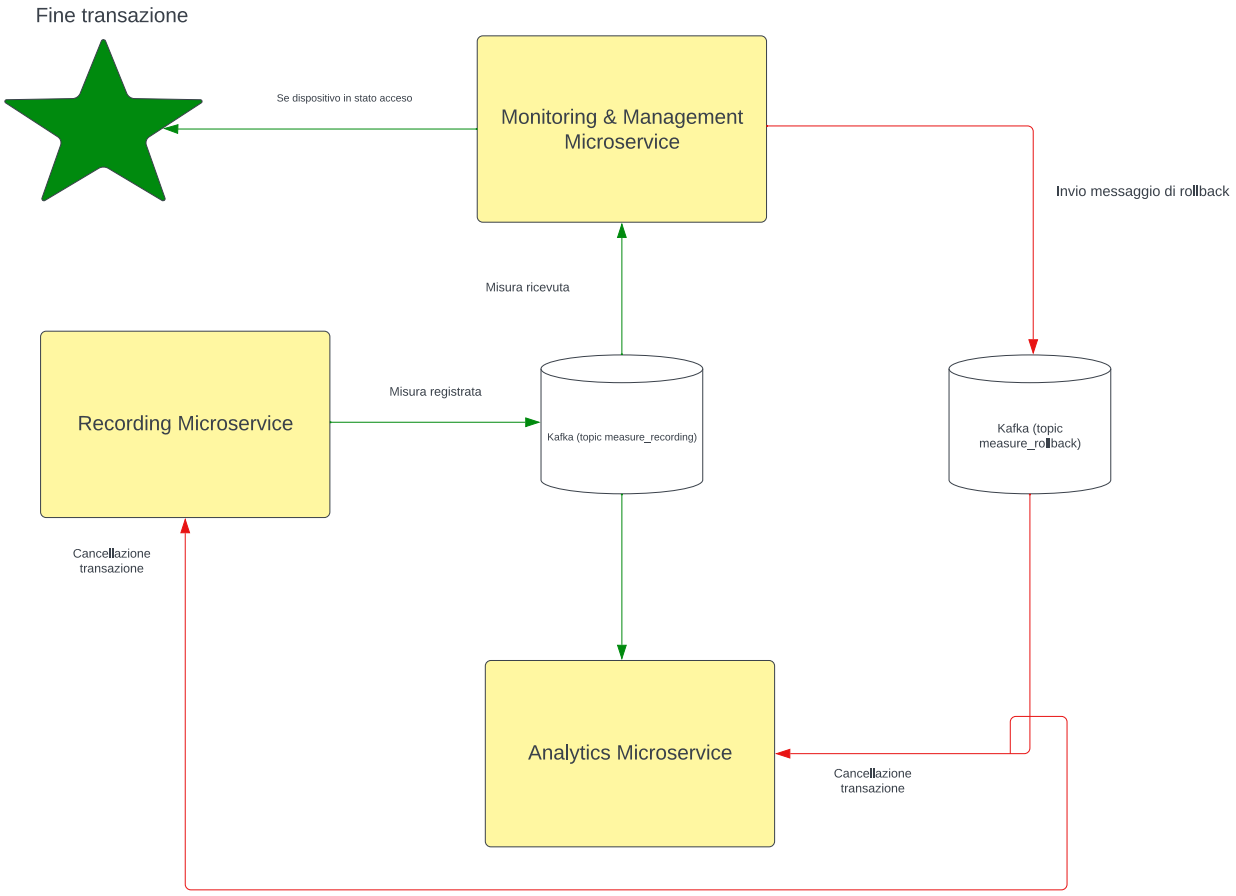
Con tale microservizio, si vuole fornire un sistema per l'analisi in tempo reale dei dati acquisiti a strumenti di terze parti per compiti come la Business Intelligence. Per far ciò si è utilizzato un database di tipo relazionale ma orientato alle analisi in tempo reale (*OLAP*): **ClickHouse**.
In modo particolare, con ClickHouse è stato possibile implementare le *Kafka Tables*, ossia delle tavole che permettono di consumare dati direttamente dai topic di Kafka, e successivamente aggregare tali dati nei successivi layer della nostra data warehouse.
La warehouse è composta dai seguenti layers:

1. **Stage.** Layer dove i dati vengono acquisiti dalle fonti esterne. In questo nostro caso, in tale layer è presente la Kafka Table per l'acquisizione delle transazioni dall'opportuno topic.
2. **Dwh.** Tale è il layer di stoccaggio, e rappresenta il primo strato di aggregazione dei dati. I dati non sono più "grezzi", ma sono stati opportunamente trasformati/puliti per gli scopi alla quali devono servire.
3. **Marts.** In tale layer, i dati sono aggregati in modo tale da fornire, per ciascuna singola osservazione (che nel nostro caso è una coppia misura-dispositivo) l'ultimo stato in cui si trovano. Per un dispositivo, in tale layer troviamo l'ultima misura effettuata, e l'ultima modifica. Per una misura, in tale strato troviamo l'ultimo valore.

Non è stata ignorata l'ipotesi di collegare Prometheus/Grafana a tale servizio (al Database) date le infinite possibilità che ClickHouse ci offre. Ma per semplicità, abbiamo deciso di affidarci ai servizi di database forniti da Prometheus e Grafana built in.

Transazione distribuita (SAGA)

In caso di misura proveniente da un dispositivo segnato come spento nel Monitoring & Management Microservice, questa misura deve essere invalidata, e quindi occorre eliminarla sia dal database transazionale che dal database analitico (rispettivamente gestiti dai corrispondenti microservizi Recording e Analytics).



Sopra il diagramma a stati. Da notare come l'utilizzo del Broker Kafka ci permetta che tale transazione avvenga in maniera asincrona nel nostro sistema distribuito, in modo da risultare non bloccante.

La misura viene inviata al topic opportuno, dove arriva ai due microservizi di analisi e di monitoraggio. Quest'ultimo provvederà a controllare lo stato del dispositivo che ha effettuato la misura: se il dispositivo è spento, allora verrà generato un messaggio di rollback nell'opportuno topic (measure_rollback), altrimenti non verrà fatto nulla. Nel caso di messaggio di rollback, i microservizi avranno implementata una logica per gestire tale tipo di messaggio in modo opportuno (grazie al recording_id, è abbastanza triviale rimuovere il messaggio dai database, sia ClickHouse che PostgreSQL).

La saga è stata gestita tramite orchestrator dato che un microservizio gestisce l'invio del messaggio di rollback, ossia il Monitoring & Management microservice.

Monitoraggio metriche sensori

Utilizziamo Prometheus (che esegue lo scraping delle metriche attraverso il servizio PushGateway) ed impostiamo ai fini di visualizzare le metriche dei sensori stessi le dashboard con Grafana.
Utilizziamo il servizio PushGateway dato che le metriche che vogliamo collezionare derivano dalle simulazioni, quindi verranno opportunamente raccolte e calcolate indipendentemente dal campionamento periodico che esegue Prometheus di default: in particolare, Prometheus eseguirà il suo campionamento su un servizio intermedio, il servizio di PushGateway appunto.
Le metriche raccolte sono il numero di simulazioni per sensore, la latenza del task di raccolta delle misurazioni stesse e la salute dei sensori.

La libreria ufficiale che implementa il client per python, sebbene nativamente non sia asincrona, è possibile utilizzarla in tal modo. Le misure vengono raccolte attraverso un Gauge (per i valori attuali delle misure) e attraverso un Counter per contare le osservazioni. Le misure fornite sono etichettate dal device_id visto che la misura stessa è strettamente legata ad esso. Per accedere a tali misure andare sul localhost:300, andare su Dashboards, cartella General, e li si trova tale schermata.

Implementazione API

Monitoring & Management Microservice

Per tale microservizio, si è deciso di implementare un'API composta dalle chiamate:

- GET - /devices: per ritornare una lista di tutti i devices
- GET - /device/{device_id}: per ritornare i dettagli relativi ad un device di dato device_id.
- PUT - /device: per aggiungere un device di dati parametri definiti nel corpo
- DELETE - /device: per rimuovere un device di dato device_id
- PATCH - /device: per modificare un device di dato device_id
- PUT - /simulate: per simulare il funzionamento dei dispositivi stessi.

Recording Microservice

Per tale microservizio, si è deciso di implementare un'API composta dalle chiamate:

- GET - /all: per ritornare una lista di tutti i recordings
- GET - /{device_id}: per ritornare una lista di tutti i recordings di un device di dato device_id
- GET – /{measure}: per ritornare una lista di tutti i recordings di una data misura

Analytics Microservice

Per tale microservizio, si è deciso di implementare un'API composta dalle chiamate:

- GET - /all: per ritornare una lista di tutti i recordings
- GET - /{device_id}: per ritornare una lista di tutti i recordings di un device di dato device_id
- GET – /{measure}: per ritornare una lista di tutti i recordings di una data misura

Kafka

Si è usato Kafka come Broker di messaggi per la comunicazione fra i diversi microservizi (ad eccezione del servizio di simulazione che fa uso di MQTT per simulare la comunicazione fra dispositivi IoT).
I topic di Kafka sono "measure_recordings", "device_commands", "measure_rollback": rispettivamente per la comunicazione delle misure effettuate, dei comandi di spegnimento del dispositivo in caso di guasto e di rollback nel caso di misurazione proveniente da un dispositivo identificato come spento.

Kubernetes

Attraverso Kubernetes, è possibile eseguire il deployment dei nostri microservizi su un cluster di nodi computazionali astraendo i dettagli a basso livello e focalizzandoci sulla computazione stessa e sui dettagli inerenti alla replicabilità dei microservizi su più pods (unità computazionali).

La versione Kubernetes ha presentato alcuni problemi dovuti alla conversione diretta fra immagini docker a manifesti yaml per Kubernetes stesso.

La versione Kubernetes, tuttavia, non si rende estremamente essenziale se non per eseguire il deployment in uno scenario reale dove vi è la necessità di scalare i microservizi a più pod (per gestire moli di dati elevate).

In ogni caso, la lezione imparata da questo malfunzionamento bloccante è appunto quella di evitare soluzioni come la conversione con Kompose, ma invece prediligere gestori di pacchetti come Helm, che attraverso le charts ci permettono di utilizzare immagini nativamente compatibili con Kubernetes.

Mongo Express

Inizialmente per valutare il database mongo, ed avere un front-end per gestirlo, ho scelto Mongo Express per fare ciò, sebbene nelle versioni finali la sua entry nel docker-compose.yml sia stata commentata per questioni di efficienza (visto che già avevamo testato il funzionamento).

Kafdrop

Kafdrop è stato usato per testare Kafka, e vedere se effettivamente i messaggi arrivavano al broker, venivano consumati (grazie all'offset) e se il broker era appunto in un buon stato di salute.

Inizializzazione dei microservizi

Su docker (non necessario in K8S grazie alla presenza del CrashBackOff) è stato necessario utilizzare un timer prima di avviare i microservizi stessi in modo tale che non fallissero nel collegarsi con Kafka (visto che Kafka richiede un po' di secondi prima di essere inizializzato).

Istruzioni e note finali

Nella repository GitHub (https://github.com/RiccardoMPesce/IoT-Devices-Simulation) sono presenti le istruzioni. La versione su Kubernetes si trova al branch "k8s".

Vogliamo riportare qui degli errori/note che sono stati riscontrati nell'esecuzione del progetto.

- La versione K8S presenta dei problemi, dovuti alla traduzione diretta (attraverso il tool Kompose) del docker-compose file a manifesti Kubernetes. In modo particolare, sono stati riscontrati problemi con i vari Microservizi, con ClickHouse (che richiede a quanto pare il modulo Operator per poter essere deployato su Kubernetes). Per deployare su Kubernetes sarebbe stato opportuno quindi utilizzare degli Helm Charts, in quanto sono delle immagini funzionali per tale piattaforma di orchestrazione.
- ClickHouse ha client non ufficiale per Python. In particolare, il driver asincrono utilizzato sembra non essere un progetto completo e stabile. E questo può causare che durante l'execute della query per eseguire il rollback dello stato, l'entry dallo strato DWH non venga rimossa.
- L'utilizzo degli script "wait-for" risulta essere problematico nel momento in cui per esempio si deve eseguire il deployment to Kubernetes, dato che la logica di funzionamento di quest'ultimo per quanto concerne i file su disco fisso risulta differente. In ogni caso, il wait-for in Kubernetes risulta essere inutile dato che esso prevede dei meccanismi built-in per il riavvio.

In ultima nota si vuole osservare che la scelta di usare Python è dovuta alla rapidità di prototipizzazione di codice e quindi al cosiddetto time to market del codice. Tuttavia è stato notato che l'utilizzo di FastAPI risulta essere buono solo qualora si vogliano implementare API molto semplici, e si voglia interoperare con i più comuni Database.

Spring sarebbe stato più complicato da configurare solo all'inizio, ma esso fornisce più supporto al deployment su Docker/Kubernetes, così come fornisce driver per i più disparati Database (come per ClickHouse), e ci permette di utilizzare Maven per gestire le dipendenze in maniera più funzionale, automatica e soprattutto centralizzata. Inoltre, con Spring molti pattern per i microservizi sono più agevoli e documentati. FastAPI, essendo Python, non ha un gran supporto per qualsiasi pattern o stile architetturale.