Relazione di Progetto: Architettura Microservizi Event-Driven per Monitoraggio IoT su Kubernetes

Studente: Enea Manzi
Corso: Cloud Computing Technologies
Anno Accademico: 2024/2025
Repository: GitHub Link

Indice

1. Introduzione e Scenario Applicativo

1.1 Obiettivi del Progetto

Questo progetto implementa un'architettura a microservizi su Kubernetes per il monitoraggio real-time di una rete di sensori IoT. Il sistema è progettato per soddisfare i requisiti di due tracce d'esame complementari:

- **Kafka (Project 1):** Implementazione di un message broker con proprietà di fault tolerance, alta disponibilità e sicurezza avanzata (TLS + autenticazione SASL).
- **API Gateway (Project 3):** Utilizzo di Kong per centralizzare autenticazione, load balancing e routing verso i microservizi backend.

1.2 Scenario Applicativo

Lo scenario di riferimento simula un ambiente industriale in cui dispositivi eterogenei inviano flussi continui di dati. Sebbene le tipologie di eventi gestiti siano quattro — **Telemetria** (dati ambientali), **Boot** (avvio dispositivo), **Firmware Updates** (log di manutenzione) e **Alerts** (allarmi critici) — il sistema è stato progettato per classificarli in due macro-categorie logiche in base ai requisiti di qualità del servizio (QoS):

1. **Flussi Operativi ad Alta Frequenza:** Dati massivi ma tolleranti a latenze minime (Telemetria, Boot, Update).

2. **Eventi Critici:** Dati sporadici che richiedono la massima garanzia di consegna e durabilità (Alerts).

L'obiettivo primario è stato soddisfare i requisiti architetturali tipici dei moderni sistemi distribuiti — **Scalabilità, Resilienza, Sicurezza e Osservabilità** — superando i limiti delle architetture monolitiche tradizionali. Per ottenere ciò, la soluzione combina due tecnologie pilastro dell'ecosistema Cloud: **Apache Kafka** per il disaccoppiamento asincrono e lo streaming degli eventi, e **Kong API Gateway** per la gestione centralizzata del traffico e della sicurezza all'edge (Ingress). L'intera infrastruttura è orchestrata su **Kubernetes**, garantendo portabilità e automazione del ciclo di vita dei componenti.

2. Architettura del Sistema

La soluzione adotta un pattern Event-Driven per disaccoppiare l'ingestione dati dal processamento, garantendo scalabilità e resilienza. I componenti comunicano in modo asincrono tramite messaggi, eliminando le dipendenze dirette (*tight coupling*) e permettendo ai servizi di ingestione e di elaborazione di scalare in modo indipendente.

2.1 Design dei Namespace e Isolamento

L'infrastruttura è stata segmentata in namespace logici per garantire la *Separation of Concerns* e una migliore gestione delle risorse, simulando un ambiente multi-tenant:

- `kong`: Dedicato esclusivamente al layer di Ingress e API Gateway, isolando il punto di ingresso dal resto del cluster.
- `kafka`: Contiene il core della business logic (Producer, Consumer), il message broker (cluster Kafka) e lo storage (MongoDB).
- `metrics`: Isola il servizio di analytics, permettendo di scalarlo indipendentemente dal flusso di ingestione e proteggendo le performance del database operativo da query analitiche pesanti.

2.2 Flusso dei Dati (Data Pipeline)

La pipeline di elaborazione è progettata per gestire picchi di carico senza perdita di dati (*Backpressure handling*):

1. **Ingress & Security:** Il traffico entra esclusivamente attraverso **Kong**, che agisce da Reverse Proxy intelligente validando l'identità del sensore tramite API Key prima ancora di contattare i servizi backend.
2. **Ingestion (Producer):** Un microservizio *stateless* sviluppato in Python (Flask) riceve il payload HTTP. Invece di scrivere direttamente su database (che creerebbe un collo di bottiglia), il Producer arricchisce il dato con metadati (UUID, timestamp) e lo instrada verso il topic Kafka appropriato in base alla tipologia di evento.
3. **Buffering & Streaming (Kafka):** Il broker persiste il messaggio su disco con garanzie di durabilità e ordine temporale. **In questa fase avviene il routing intelligente**: gli eventi operativi (Boot, Telemetry, Update) vengono instradati verso `sensor-telemetry`,

mentre gli allarmi critici confluiscono in `sensor-alerts`, permettendo politiche di retention e compressione differenziate.
4. **Processing (Consumer):** Un worker sottoscrive i topic, preleva i messaggi ed esegue la normalizzazione dei dati (es. conversione timestamp ISO-8601 in oggetti data nativi).
5. **Storage (MongoDB):** I dati elaborati vengono storicizzati in collezioni ottimizzate per serie temporali.
6. **Analytics:** Il *Metrics Service* interroga il database per calcolare aggregazioni on-demand (es. media mobile della temperatura) esposte via API REST.

3. Apache Kafka: il Core Event Streaming

La scelta di **Apache Kafka** come spina dorsale del sistema risponde alla necessità di gestire alti throughput di scrittura e garantire la persistenza dei dati anche in caso di disservizi dei consumer.

3.1 Orchestrazione Cloud-Native (Strimzi Operator)

L'implementazione su Kubernetes è stata realizzata tramite **Strimzi**, che adotta il pattern *Operator*. Questo permette di gestire il cluster Kafka come una risorsa nativa Kubernetes (`Kind: Kafka`), automatizzando compiti complessi come il rolling update dei nodi, la gestione dei certificati TLS e la configurazione dei listener.

Un aspetto innovativo di questa implementazione è l'uso della modalità **KRaft** (Kafka Raft Metadata), definita esplicitamente nella configurazione del cluster tramite `KafkaNodePool` distinti per ruoli di *Controller* e *Broker*. Questa architettura sostituisce la storica dipendenza da **ZooKeeper**, internalizzando la gestione dei metadati che in precedenza richiedeva un complesso coordinatore esterno.

- **Vantaggio Architetturale:** L'assenza di ZooKeeper semplifica drasticamente il deployment e riduce il footprint di risorse (CPU/RAM), eliminando la necessità di gestire, mettere in sicurezza e monitorare due sistemi distribuiti paralleli (Kafka + ZK) in favore di un'unica piattaforma unificata.

3.2 Strategia dei Topic e Trade-off

La configurazione dei topic riflette una precisa analisi dei requisiti non funzionali per le diverse tipologie di dati, implementando strategie differenziate:

- **Topic `sensor-telemetry` (Alta Efficienza):** Destinato a Telemetria, Boot e Update.
    - **Compressione Strategica (LZ4):** È stata abilitata la compressione `lz4` a livello di topic. Questa scelta implementa la prima fase di una **strategia di compressione ibrida**:
        - *Transport Layer (Kafka):* LZ4 è stato selezionato per la sua velocità di decompressione estremamente elevata e il bassissimo overhead sulla CPU. Per i flussi IoT in tempo reale, minimizza la latenza e riduce l'occupazione di

banda (fino al 60% per payload JSON ripetitivi), garantendo che il throughput del Producer non diventi un collo di bottiglia.
        - *Integrazione Architetturale:* L'uso di LZ4 per il trasporto "veloce" si integra perfettamente con il layer di persistenza successivo, dove MongoDB (tramite le Time Series collections) applica nativamente la compressione **Zstd** (più aggressiva ma computazionalmente più onerosa) per massimizzare l'efficienza dello storage a lungo termine.
    - **Partizionamento:** Configurato con 3 partizioni per massimizzare il parallelismo, permettendo fino a 3 istanze del Consumer di leggere contemporaneamente senza sovrapporsi.
    - **Retention:** Limitata a 7 giorni, sufficiente per mantenere disponibili i dati operativi "caldi" per eventuali replay o analisi immediate, delegando lo storico profondo a MongoDB.

- **Topic `sensor-alerts` (Alta Affidabilità):** Destinato esclusivamente agli allarmi.
    - **Durabilità (Min.ISR):** Configurato con `min.insync.replicas: 2` e `replicas: 2`. Questo garantisce che **nessun allarme venga perso** anche in caso di crash improvviso di un broker, ma richiede che entrambe le repliche siano online per accettare scritture (trade-off: disponibilità ridotta durante manutenzioni).
    - **Retention:** Estesa a 30 giorni per permettere analisi post-incidente e audit.

4. Kong API Gateway: Edge Computing & Security

L'esposizione diretta dei microservizi è considerata un anti-pattern in ambienti distribuiti. Si è scelto di utilizzare **Kong** come Ingress Controller per implementare il pattern *Gateway Offloading*: le responsabilità trasversali (autenticazione, rate limiting, routing) sono spostate dal codice applicativo all'infrastruttura.

4.1 Sicurezza Dichiarativa "As Code"

La sicurezza non è implementata nel codice Python, ma definita tramite risorse Kubernetes (CRD - Custom Resource Definitions).

- **Autenticazione (Key-Auth):** Tramite il plugin `KongPlugin` di tipo `key-auth`, Kong intercetta ogni richiesta in ingresso. Verifica la presenza dell'header `apikey` e lo confronta con i *Secret* Kubernetes (`iot-devices-apikey`). Solo se la chiave è valida, la richiesta viene inoltrata al backend. Questo approccio permette di revocare l'accesso a dispositivi compromessi semplicemente aggiornando il Secret, senza dover ridistribuire o riavviare i microservizi.
- **Protezione (Rate Limiting):** L'architettura supporta l'applicazione dinamica di policy di Rate Limiting per mitigare attacchi Denial of Service (DoS), proteggendo i servizi backend da picchi anomali di traffico.

4.2 Routing e Load Balancing

Kong gestisce il routing basato su Host (es. `producer.nip.io`) e instrada il traffico verso i *Service* Kubernetes sottostanti. Il bilanciamento del carico tra le varie repliche dei pod applicativi avviene automaticamente sfruttando il meccanismo nativo dei Service, garantendo una distribuzione uniforme delle richieste.

5. MongoDB: Persistenza e Ottimizzazione Dati

Per lo storage è stato selezionato **MongoDB**, configurato per gestire efficacemente carichi di lavoro IoT caratterizzati da scritture massive ("write-heavy") e letture analitiche.

5.1 MongoDB Time Series Collections

Invece di utilizzare collezioni documentali standard, il sistema sfrutta le funzionalità native delle **Time Series Collections** (introdotte in MongoDB 5.0).

- **Ottimizzazione Storage e Compressione:** I dati vengono organizzati fisicamente in "bucket" compressi, raggruppati per intervallo temporale e `metaField` ("device_id"). Questa struttura colonnare permette a MongoDB di applicare algoritmi di compressione ad alta efficienza (come **Zstd**, standard per le Time Series) in modo molto più efficace rispetto ai documenti JSON tradizionali. Il risultato è una drastica riduzione dello spazio occupato su disco.
- **Efficienza I/O:** Grazie a questa struttura, le query su range temporali (es. *"dammi la temperatura media dell'ultima ora"*) richiedono la lettura di un numero significativamente inferiore di blocchi disco rispetto a una collezione documentale sparsa, migliorando sia il throughput di lettura che l'efficienza.
- **Performance Analitiche:** Le query di aggregazione implementate nel *Metrics Service* (es. media temperatura per zona) beneficiano di indici `clustered` creati automaticamente sul campo temporale, garantendo tempi di risposta rapidi anche nell'analisi di storici profondi.

5.2 Separazione Configurazione

La configurazione rispetta rigorosamente i principi delle *Cloud Native Apps*:

- **Configurazione esternalizzata:** Endpoint, porte e nomi dei database sono iniettati nei pod tramite `ConfigMap`.
- **Segreti protetti:** Le credenziali sensibili (utente/password Mongo, credenziali SASL Kafka) sono gestite esclusivamente tramite `Secrets` e montate come variabili d'ambiente. Nessuna password è presente in chiaro nel codice sorgente o nei file di configurazione non cifrati.

6. Analisi delle Proprietà Non Funzionali (NFP)

L'architettura è stata validata rispetto a tre pilastri fondamentali per sistemi cloud-native mission-critical, verificati tramite scenari di test specifici.

6.1 Security & Secrets Management

La sicurezza è stata implementata seguendo il principio della *Defense in Depth*, proteggendo ogni livello dello stack:

1. **Protezione del Canale (Data in Transit):** La comunicazione interna tra i microservizi (Producer/Consumer) e il cluster Kafka avviene esclusivamente su canale cifrato **TLS** (porta 9093). Questo previene attacchi di tipo *man-in-the-middle* all'interno del cluster.
    1. **Autenticazione dei Servizi:** L'accesso al broker non è anonimo ma protetto dal protocollo **SASL/SCRAM-SHA-512**. Ogni componente possiede un'utenza dedicata (`producer-user`, `consumer-user`) definita tramite risorse `KafkaUser`.
2. **Autenticazione all'Edge:** Kong agisce come *Policy Enforcement Point*. Nessuna richiesta raggiunge i microservizi se non possiede un header `apikey` valido, verificato contro i Secret Kubernetes. Questo centralizza la gestione degli accessi, evitando di disperdere logica di auth nel codice applicativo.
3. **Gestione dei Segreti:** Nessuna credenziale è hardcodata. Le password di MongoDB e Kafka sono iniettate nei Pod esclusivamente tramite **Kubernetes Secrets**, separandole nettamente dalla configurazione non sensibile gestita tramite **ConfigMaps**.
4. **Protezione da DoS (Rate Limiting):** È stata verificata sperimentalmente la capacità dell'infrastruttura di mitigare attacchi di tipo *Flood*. Applicando una policy di Rate Limiting (tramite risorsa `KongPlugin` configurata a 5 req/sec), un test di iniezione massiva di richieste ha confermato che il traffico eccedente viene respinto direttamente all'edge con codice HTTP `429 Too Many Requests`. Questo garantisce che le risorse dei microservizi backend non vengano saturate da traffico malevolo o anomalo.

6.2 Resilience, Fault Tolerance & High Availability

Il sistema è progettato per sopravvivere a guasti parziali senza perdita di dati o interruzione del servizio:

- **Fault Tolerance (disaccoppiamento e buffering):** La natura asincrona di Kafka garantisce che un crash del *Consumer* non impatti il *Producer*. Durante un disservizio del worker, i messaggi si accumulano nei topic Kafka (che agiscono da buffer persistente) e vengono elaborati (*drained*) non appena il Consumer torna online, garantendo zero perdita di dati (*Zero Data Loss*).
- **Self-Healing & HA (Kubernetes):** I `Deployment` configurati assicurano che il numero desiderato di repliche sia sempre attivo. In caso di crash del processo Python, il Kubelet riavvia automaticamente il container.

6.3 Scalabilità & Load Balancing

La validazione delle capacità di scaling è stata condotta in due fasi distinte: una verifica manuale per confermare i meccanismi di distribuzione del carico e una verifica automatica per testare

l'elasticità del sistema.

1. **Verifica dei Meccanismi (Scaling Manuale):**
   Prima di attivare l'automazione, è stato simulato uno scenario di carico incrementando manualmente le repliche dei microservizi per validare il comportamento architetturale:

   - **Ingress Load Balancing (HTTP Layer):** Scalando il *Producer* a 2 repliche e iniettando un carico di richieste rapide (Burst), è stato verificato tramite l'analisi dei log che il traffico viene distribuito alternativamente tra i diversi Pod. Questo conferma che il **Kong Ingress** e il Service Kubernetes bilanciano correttamente le richieste in ingresso.
   - **Consumer Parallelism (Stream Layer):** Scalando il *Consumer* a 3 repliche (corrispondenti alle 3 partizioni del topic `sensor-telemetry`), è stato verificato che tutti i Pod processano messaggi contemporaneamente. Questo dimostra che il protocollo di *Consumer Group* di Kafka assegna correttamente le partizioni esclusive ai nuovi worker, massimizzando il throughput di lettura.

2. **Elasticità Automatica (HPA):**
   Validati i meccanismi sottostanti, è stato attivato l'**Horizontal Pod Autoscaler (HPA)** configurato su una soglia di CPU del 50%. Sottoponendo il sistema a uno stress test prolungato, l'architettura ha dimostrato la capacità di:

   - **Scale Out:** Rilevare la saturazione della CPU e avviare automaticamente nuove repliche (fino al limite configurato di 4) per assorbire il picco.
   - **Scale Down:** Rilasciare le risorse terminando i Pod in eccesso al termine del carico, riportando il cluster allo stato operativo standard e garantendo l'efficienza dei costi.

7. Conclusioni

Il progetto ha dimostrato come l'orchestrazione tramite Kubernetes permetta di integrare tecnologie complesse come Kafka e Kong in un'architettura coesa, gestibile e scalabile.
L'adozione di pattern moderni come **KRaft** per Kafka (eliminando Zookeeper) e **Time Series** per MongoDB evidenzia un'attenzione specifica all'efficienza delle risorse, fondamentale in ambienti Cloud.
L'uso estensivo di Operatori (Strimzi) e CRD (Kong) ha permesso di spostare la complessità operativa dal codice applicativo alla configurazione dell'infrastruttura (*Infrastructure as Code*), risultando in un sistema non solo funzionale, ma progettato per evolvere e resistere ai guasti in scenari di produzione reali.