POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica
Sessione di Laurea Dicembre 2021

Tesi di Laurea Magistrale

Progetto e implementazione di un servizio di raccolta in streaming di dati IoT per produzione e logistica

Relatori
Prof. GIOVANNI MALNATI
Prof. FABIO FORNO

Candidato
BELTRANDO ALESSIO RANIERI

Anno Accademico 2020-2021

Alla mia famiglia e a tutte le persone che hanno creduto in me. Questo traguardo lo dedico a voi.

Sommario

Al giorno d'oggi l'ottimizzazione dei processi aziendali è una delle problematiche di maggiore interesse con cui le aziende si misurano. Tra questi, uno dei processi di maggiore criticità è sicuramente quello della logistica e in particolare della tracciabilità di un prodotto. Per tracciabilità di un prodotto, si intende la possibilità di reperire tutte le informazioni riguardanti i processi produttivi che hanno portato al prodotto finale. Nel mondo della filiera animale, dove la tracciabilità è di fondamentale importanza, sta entrando in gioco anche il mondo dell'industria 4.0, che utilizzando dispositivi IoT, aiuta a mantenere in piedi il meccanismo di monitoraggio tramite una continua raccolta di dati ed eventi. Il lavoro di tesi svolto rientra in un progetto più grande che ha come risultato quello della realizzazione di una piattaforma che prende il nome di iChain, avendo come scopo quello di aiutare le aziende, in un contesto di Supply Chain, con la raccolta, il processing e lo scambio dei dati raccolti, in maniera efficiente e veloce.

Il mio lavoro si basa sulla realizzazione di un backend in grado di ricevere e processare dati dai sensori IoT. Durante questo processo, si è dovuti essere conformi agli standard GS1, in particolar modo allo standard EPCIS che consente la condivisione di eventi, e quindi delle informazioni, tra i vari attori della Supply Chain. I dati provenienti dai sensori sono delle serie temporali, ovvero dati inseriti in ordine cronologico, identificati da un timestamp che ne indica il momento in cui vengono generati. Grazie a queste serie è possibile analizzare i dati precedenti e stimare delle variazioni future per rilevare guasti o generare avvisi se ce ne fosse il bisogno. Lavorare con serie temporali porta però con sé l'onere di una gestione molto elevata di dati, quindi è di fondamentale importanza usare tecnologie e pattern adeguati alla loro gestione.

La tesi parte con lo scopo di studiare una corretta configurazione per le tabelle in TimescaleDB. Quest'ultimo è un database creato come estensione di PostgreSQL appositamente per la gestione delle serie temporali. Le tabelle hanno il compito di memorizzare i dati raw provenienti dai sensori, e sono ingeriti nel sistema tramite un'interfaccia di capturing con il gateway dei dispositivi IoT. Queste tabelle di dati raw sono collegate ad una piattaforma per il data streaming, che prende il nome di Kafka, tramite un connector manuale che rileva i cambiamenti all'interno delle tabelle e li pubblica su dei topic per poi essere processati. Le regole per processare i dati sono decise dall'azienda stessa tramite delle API Web e possono consistere in operazioni di semplice monitoraggio, oppure nell'impostazione di allarmi per la salvaguardia del prodotto. Tutte le API con la quale il server espone i suoi servizi vengono autenticate tramite il protocollo JWT.

Con l'implementazione di una prima versione di un backend per il monitoraggio dei

sensori IoT, gli obiettivi principali del presente lavoro sono stati raggiunti. Per versioni future, sono possibili ancora delle migliorie dal punto di vista del monitoraggio aziendale, con l'aggiunta di altre regole e di altri tipi di allarme, nonché con l'introduzione di nuove API per successive funzioni disponibili ai client, come quella della visualizzazione in tempo reale dei dati acquisiti.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Al giorno d'oggi l'ottimizzazione dei processi aziendali è una delle problematiche di maggiore interesse con cui le aziende si misurano. Tra questi, uno dei processi di maggiore criticità è sicuramente quello della logistica ed in particolare della tracciabilità di un prodotto. Per tracciabilità di un prodotto, si intende la possibilità di reperire tutte le informazioni riguardanti i processi produttivi che hanno portato al prodotto finale.

Tracciare un prodotto significa inoltre maggiori garanzie per il consumatore finale, e maggiore tutela per il produttore, che può garantire in questo modo la qualità del suo prodotto e dimostrare come nel processo di produzione sia rimasto conforme alle norme vigenti. In un mondo globalizzato come il nostro, dove la tracciabilità è di fondamentale importanza, sta entrando in gioco anche il mondo dell'industria 4.0, che utilizzando dispositivi IoT, aiuta a mantenere in piedi il meccanismo di monitoraggio tramite una continua raccolta di dati ed eventi. Nell'ambito industriale l'applicazione dell'Iot è noto con l'acronimo I-IoT (Industrial Internet of things). Collegando numerosi dispositivi intelligenti e impianti tramite Internet, le aziende possono automatizzare tutta una serie di processi, risultando più competitive a costi molto minori. L'I-IoT è quindi una tecnologia chiave nell'industria 4.0, quelli che molti definisconno come una nuova rivoluzione industriale.

Il meccanismo di monitoraggio, documentando ogni passaggio produttivo della filiera, permette un controllo radicale sulla produzione garantendo che le norme e leggi vigenti siano rispettate. Le leggi hanno lo scopo di tutelare il consumatore finale, e per farlo necessitano di tenere traccia non soltanto dei processi dell'azienda che ha gestito la produzione, ma anche di quelli derivanti dalle ditte che hanno partecipato in maniera indiretta al prodotto, un esempio può essere un azienda che fornisce un mangime ad un allevatore, anche non rientrando direttamente nel processo produttivo della carne o del latte, contribuisce con una sua percentuale alla qualità del prodotto finale. Quindi, in un contesto di Supply Chain come quello descritto, è importante gestire in maniera efficiente tutto il coordinamento delle varie fasi che concorrono alla realizzazione del prodotto, a partire dalla gestione delle materie prime, fino ad arrivare all'immagazinamento finale e la vendita vera e propria della merce al consumatore. Quest'ultimo ovviamente, è il "giudice finale" del processo di produzione, ed avere una Customer Experience adeguata costituisce un modo per poter reperire in maniera rapida e veloce le informazioni sul prodoto che

sta acquistando, permettendo di dare prima di tutto garanzie di qualità sulla merce, e in secondo luogo dimostrare come si siano rispettatte tutte le normi vigenti nel processo produttivo.

Avere una buona tracciabilià non è solamente obbligo legislativo o Customer Experience, bensì consente di migliorare significativamente gli step delle fasi produttive, individuando in maniera precisa come e dove agire per abbattere i costi, e quali aspetti nella produzione rappresentino un collo di bottiglia per la stessa. Un aspetto che rende tutto ciò possibile è la comunicazione tra gli attori nella Supply Chian, attualmente molte imprese si affidano ancora a del materiale cartaceo per la condivisione delle informazioni, oppure lavorano in un contesto ancora poco digitalizzato, risultando essere inadeguate e molto poco efficienti nei confronti della concorrenza con una buona gestione della Supply Chain. Un ultimo aspetto critico è la rintracciabilità di un prodotto, che rappresenta l'altro lato della stessa medaglia della tracciabilità. Essa infatti è il processo opposto: dal prodotto finale torna indietro per rintracciare un preciso evento nella catena produttiva.

Grazie a questi due aspetti un'impresa può avere una storia completa della propria produzione, prendendo di volta in volta decisioni adeguate alle circostanze, come ad esempio nel caso di un lotto alimentare andato a male: rintracciando il lotto è possibile con facilità ritirare dal mercato ogni elemento di quel lotto, e grazie ad una buona tracciabilià è possibile capire cosa sia successo e perché. Per realizzare quindi una buona gestione della Supply Chain in questo ambito, c'è bisogno di una buona raccolta ed elaborazione dati, nonchè di standard tra partner per una chiara e veloce comunicazione tra essi. Nello specifico:

- La raccolta dati è intesa come un'efficiente data ingestion dai sensori IoT;
- L'elaborazione dati rappresenta la parte di validazione, storing e processing dei dati raccolti dai sensori;
- Gli standard per la comunicazione devono essere chiari ed efficienti a tutti gli attori;

Oltre a questi punti, di fondamentale importanza è la visualizzazione dei dati, per il monitoraggio e il processo decisionale che avviene a monte della catena produttiva.
Il lavoro di questa tesi riguarda proprio la realizzazione di un backend in grado di ricevere e processare dati dai sensori IoT, restando conformi agli standard GS1 ed in particolare ad EPCIS.I dati provenienti dai sensori sono delle serie temporali, ovvero dati inseriti in ordine cronologico, identificati da un timestamp che ne indica il momento in cui vengono generati. Grazie a queste serie è possibile analizzare i dati precedenti e stimare delle variazioni future per rilevare guasti o generare avvisi se ce ne fosse il bisogno. Lavorare con serie temporali porta però con sé l'onere di una gestione molto elevata di dati, quindi è di fondamentale importanza usare tecnologie e pattern adeguati alla loro gestione. In particolare emerge la necessità di lavorare con dei stream di dati, che hanno la necessità di essere elaborati in un tempo ragionevole, quindi con una latenza contenuta, e senza bloccare la raccolta e il processamento di altri dati. Il paradigma del Data Streaming, ed in particolare la piattaforma Apache Kafka (per questo lavoro di tesi), ci viene incontro per far fronte a questa problematica.

Il lavoro di questa tesi rientra in un progetto più grande che ha come risultato quello della realizzazione di una piattaforma che prende il nome di iChain, avendo come scopo quello di aiutare le aziende, in un contesto di Supply Chain, con la raccolta, il processing e lo scambio dei dati raccolti, in maniera efficiente e veloce. Nei prossimi capitoli parleremo nel dettaglio degli standard GS1, della struttura del back-end realizzato, nonchè delle tecnologie utilizzate per raggiungere l'obiettivo.

Capitolo 2

Stato dell'arte

Realizzare una piattaforma che soddisfi gli obiettivi descritti nel capitolo precedente, è una sfida per molte aziende al giorno d'oggi, e come risposta molte di esse si servono di servizi online all'avanguardia e in continuo miglioramento, sia open-source che proprietari. Molte di queste piattaforme vengono fornite come *SaaS* (Software as a Service) da cui è possibile accedere anche tramite un interfaccia Web in maniera semplice e immediata, rendendo minimi i tempi di configurazione, oppure come*PaaS* (Platform as a Service). Inoltre bisogna aggiungere come le sopramenzionate piattaforme stiano aumentando sempre di più in numero a causa dell'enorme domanda di sistemi di monitoring IoT all'interno del mercato, creandone un'abbondanza che spinge sempre più a migliorare il proprio prodotto per tenerlo competitivo rispetto alla concorrenza.

Ci sono dei processi che accomunano tutte queste piattaforme, e sono il data collecting dei dati, il processing e la devices management, ma differiscono per il modo con cui si interfacciano con l'utente finale in fase di visualizzazione. Molte di esse sono inoltre soluzioni Cloud che si basano sull'infrastruttura dello stesso creatore della piattaforma, come ad esempio Amazon o Google. Dal punto di vista dell'azienda che ne usufruisce, questa abbondanza di soluzioni amplifica le possibilità di scelta, dando l'opportunità di selezionare quella che risulti essere più adatta alle proprie esigenze. Andiamo dunque a vedere i più popolari tra questi servizi:

- **Thingsboard**: Piattaforma Open-Source che fornisce tutto il lato server per i propri progetti IoT. Questo servizio è progettato per essere scalabile orizzontalmente, tollerante ai guasti, robusto, efficiente ed estremamente personalizzabile: grazie ad una serie di widget è possibile costruire la propria rules chain in maniera molto semplice e immediata. Fornisce il proprio servizio tramite un'applicazione Desktop;

- **Davra IoT**: Piattaforma proprietaria che eroga i suoi servizi tramite un inerfaccia Web, quindi accessibile da qualunque dispositivo. Le caratteristiche sono simili a quelle di Thingsboard, però in aggiunta forninsce una maggiore integrazione con prodotti di terze parti;

- **Rayven**: La caratteristica principale di questa soluzione è quella dell'integrazione tra AI e IoT, riducendo ancor più i tempi di intervento umano su aspetti di monitoraggio e prevenzione;
- **Kaa**: Anche qui stiamo parlando di una piattaforma proprietaria che eroga i suoi servizi basandosi sul suo Cloud sottostante, qui parliamo di PaaS ed oltre a trattare monitoring di IoT il suo range di impiego varia dal Big Data e Machine Learning fino ad arrivare proprio all'Embedded (come nel caso di dispositivi IoT);

Nell'ambito del lavoro di questa tesi, e più in generale dell'ambito applicativo di iChain, il problema non è relativo alle operazioni effetuate da questi servizi sul collecting dei dati, il processamento oppure la visualizzazione, bensì nel rendere conforme tutti gli stadi del monitoring agli standard GS1. Con la realizzazione di una prima versione di un back-end per scopi di monitoring IoT in iChain, il presente lavoro di tesi vuole gettare le basi proprio per una piattaforma in grado di essere conforme a GS1 ed in particolare ad EPCIS, standard presentato nel prossimo capitolo.

Capitolo 3

GS1

GS1 è un'organizzazione non-profit che sviluppa e mantiene standard globali per la comunicazione aziendale [19]. Il più noto tra questi standard è il codice a barre, un simbolo grafico che può essere scannerizzato elettronicamente e che viene applicato ai prodotti. I codici a barre, attraverso strumenti fisici e digitali, migliorano l'efficienza, la sicurezza, la rapidità e la visibilità delle supply chain. Un altro standard che sta rivoluzionando la gestione della Supply Chain è EPCIS, di cui parleremo approfonditamente in questo capitolo.

3.1 EPCIS

Ichain per il suo funzionamento si basa su questi standard GS1, in particolare sullo standard EPCIS (Electronic Product Code Information Services). L'obiettivo di EPCIS è quello di consentire a diverse applicazioni di creare e condividere dati sugli eventi che si vogliono mostrare, sia all'interno di un'impresa che tra le diverse aziende [8]. Lo standard EPCIS è stato originariamente concepito come parte di uno sforzo più ampio per migliorare la collaborazione tra partner commerciali condividendo informazioni dettagliate su oggetti fisici o digitali, che sono identificati a livello di classe o di istanza. Questi oggetti rappresentano l'attore principale di una fase del processo aziendale. EPCIS fornisce interfacce aperte e standardizzate che consentono una perfetta integrazione di servizi ben definiti in ambienti interaziendali e all'interno delle aziende. Fornendo solo interfacce standard di condivisione dei dati tra le applicazioni che acquisiscono i dati dei *visibility event* e quelle che devono accedervi, EPCIS non pone nessun vincolo su come le operazioni di servizio o i database stessi dovrebbero essere implementati.

Riassumendo, EPCIS definisce:

- Un data model che modella gli eventi che si verificano.
- Interfacce standard di condivisioni dei dati che non pongono vincoli sulle implementazioni delle tecnologie sottostanti

Esistono 2 tipi di interfacce:

- La *capture Interface*, che definisce la consegna di eventi EPCIS dall'applicazione di cattura ad altri attori che consumano i dati in tempo reale;
- La *query Interface* che definisce un mezzo per richiedere e inviare eventi EPCIS ad una applicazione, dopo l'acquisizione, in genere interagendo con una repository EPCIS

Complementare allo standard EPCIS abbiamo lo standard CBV (Core Business Vocabulary). Lo standard CBV fornisce definizioni di valori di dati che possono essere utilizzati per popolare le strutture dati definite nello standard EPCIS. Pertanto, le applicazioni dovrebbero utilizzare lo standard CBV nella massima misura possibile durante la costruzione degli eventi EPCIS [6].

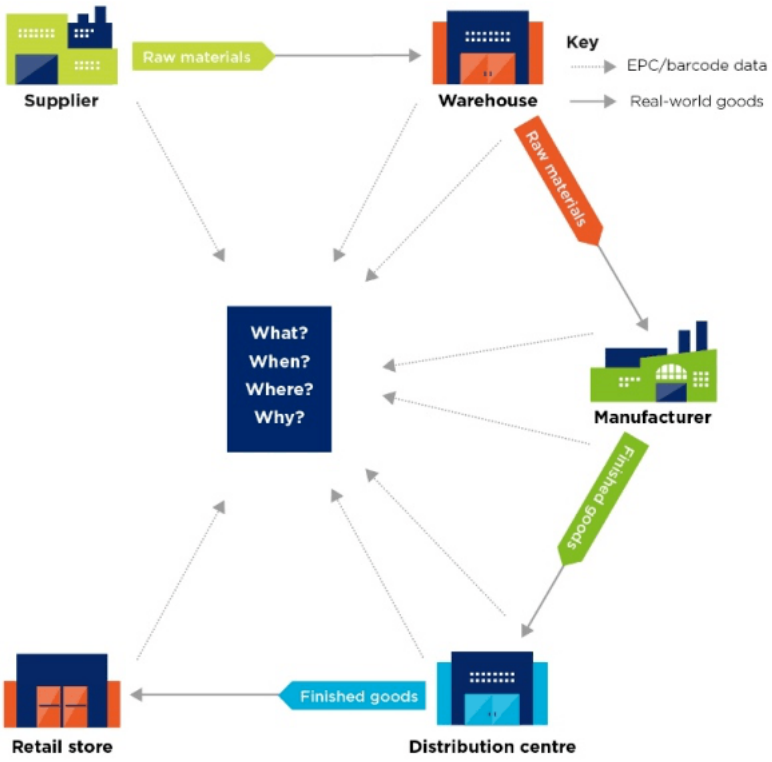

Figura 3.1. GS1 EPCIS

3.1.1 Struttura degli eventi EPCIS

Lo standard EPCIS specifica gli elementi di dato in un evento EPCIS. Questi elementi di dato possono essere divisi in 4 dimensioni:

- La dimensione **what**, che contiene uno o più identificatori univoci per gli oggetti, che siano essi fisici i digitali;

- La dimensione **when**, che indica il momento in cui un evento EPCIS è stato scatenato;
- La dimensione **where**, formata dall'unione di due identificatori, che forniscono due aspetti differenti del luogo in cui l'evento è stato generato:
  - *Read Point*: Il luogo in cui si è svolto l'evento EPCIS.
  - *Business Location*: il luogo dove il prodotto risiede fino a quando un altro evento non indicherà diversamente, dopo la sua acquisizione
- La dimensione **why** fornisce informazioni sul processo aziendale associate all'evento, inclusa la fase del processo aziendale che ha "attivato" l'acquisizione dell'evento;

La dimensione *what* di un evento EPCIS precisa quali oggetti sono coinvolti in un evento EPCIS. Lo standard fornisce due modi per identificare i suddetti oggetti:

- *Instance-level*: identificativo univoco per il singolo oggetto, una volta assegnato non può essere assegnato nuovamente ad un altro oggetto;
- *Class-level*: identificativo che può essere assegnato a più oggetti;

L'identificativo *Class-level* viene utilizzato solo quando non è possibile assegnare un identificativo *Instance-level* ad ogni oggetto. Si preferisce assegnare un *Instance-level* perché in questa maniera è possibile correllare più eventi EPCIS se essi hanno lo stesso identificativo. In questo modo, ad esempio, si può capire che lo stesso oggetto EPCIS è stato coinvolto più volte negli eventi.

La struttura che viene utilizzata per gli oggetti che vengono identificati da un *Class-level Identifier* prende il nome di *QuantityElement*. Esso oltre ad essere utilizzato quando non è possibile avere una stima precisa sul numero esatto di oggetti, può essere utilizzato anche quando abbiamo un numero preciso ed enumerabile di elementi, magari per una questione di comodità. Il *QuantityElement* è composto da tre campi:

- *epcClass*: *Class-level Identifier* per la classe a cui appartiene la quantità specificata di oggetti;
- *quantity*: numero che indica la quantità di elementi presenti appartenenti alla *epcClass* indicata nel campo precedente. Questo campo può anche essere omesso, nel caso in cui la quantità esatta fosse sconosciuta;
- *uom*: Unity of measure, sta ad indicare l'unità di misura della quantità descritta nel campo precedente. Anche questo campo è opzionale, e deve essere omesso se il campo quantity fosse NULL;

La dimensione *where* definisce quattro tipi *location*, solo che due di questi sono riferiti ai *lettori* di oggetti, utilizzando tecnologia RFID ad esempio. Quindi le *location* vere e proprie, in questa dimensione sono solamente due:

- *ReadPointID*: *location* che indica, nella maniera più specifica, dove l'evento EPCIS ha avuto luogo. Concettualmente quindi, il punto di lettura è progettato per identificare "dove si trovavano gli oggetti al momento dell'evento EPCIS";
- *BusinessLocationID*: la *location* che sta ad identificare "dove gli oggetti seguono l'evento EPCIS". Un futuro evento EPCIS può cambiare questa location;

Un esempio della distizione tra le due location è data dalla figura 3.2

La dimensione *why*, che definisce le cause che hanno scatenato l'evento, quindi è cru-

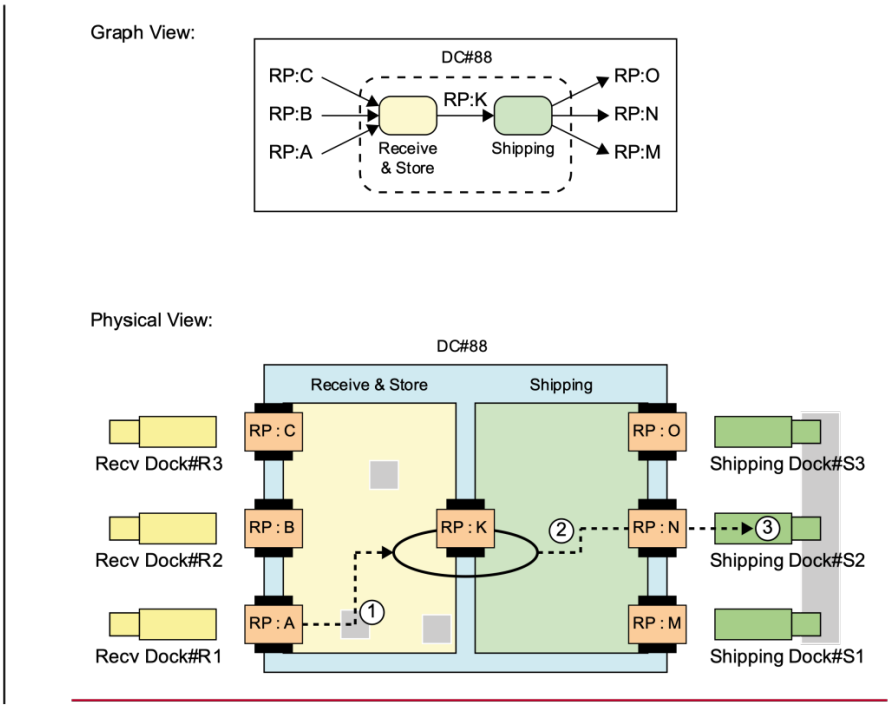

Figura 3.2. Esempio, stanze con porte fisse ai confini delle stanze. In tal caso, i Read Point corrispondono alle porte (con strumentazione RFID) e le Business Location corrispondono alle stanze

ciale per le applicazioni che utilizzano lo standard per dare un senso ai dati EPCIS. L'attributo più importante è il *bizStep*, che definisce il contesto di business di un evento,

ovvero quale è il processo di business che ha scatenato l'evento. Questo attributo è un ID, e deve essere conosciuto e mappato in anticipo dalle aziende, in modo da poterne capire il significato. Questa mappatura è possibile grazie all'uso del CBV, descritto in seguito.

Un altro attributo importante della dimensione *why* è la *disposition*Si presume che la *disposition* sia vera fino a quando un altro evento non indica un cambiamento di disposizione. Così come il *bizStep* anche questo valore è un ID mappato con lo Standard Vocabulary.

L'attributo *persistentDisposition* denota lo stato aziendale persistente di un oggetto. Il campo *persistentDisposition* di un evento viene utilizzato per impostare o disattivare la condizione aziendale del soggetto dell'evento (le cose specificate nella dimensione *what*), a seguito dell'evento. A differenza della disposizione, la disposizione persistente non è sovrascritta da successive *disposition* o *persistentDisposition*, e può essere negata o annullata solo se settata esplicitamente ad "unset".

L'attributo *bizTransactionList* è una lista di operazioni commerciali.Ogni Business Transaction è composta da un paio di identificatori:

- *type*: identificativo che denota il tipo di transizione. Questo campo può anche essere omesso;

- *bizTransaction*: identificativo della transazione;

Infine gli attributi *Source* o *Destination* denotano un'origine o una destinazione del tipo indicato dal tipo di transazione commerciale. L'attributo specifica un trasferimento di proprietà, custodia o responsabilità di un determinato oggetto, sia esso fisico o digitale. Vengono specificati due identificatori per le origini e le destinazioni:

- *Source or Destination Type*: che tipo di origine o destinazione la *Source* o *Destination* denota;

- *Source or Destination Identifier*: se si tratta di una Source o di una Destination;

3.1.2 Master data

Lo standard EPCIS lavora con due tipi di dato, gli event data e i master data. I master data sono dati aggiuntivi che danno un contesto agli event data, per rederli comprensibili. Così come gli event data, che sono catturati da *EPCIS Capture Interface* e resi disponibili attraverso l'*EPCIS Query Interfaces*, anche i master data possono essere interrogati da questa interfaccia, ma nello standard non è specificato nessuno modo con la quale essi debbano entrare nel sistema.

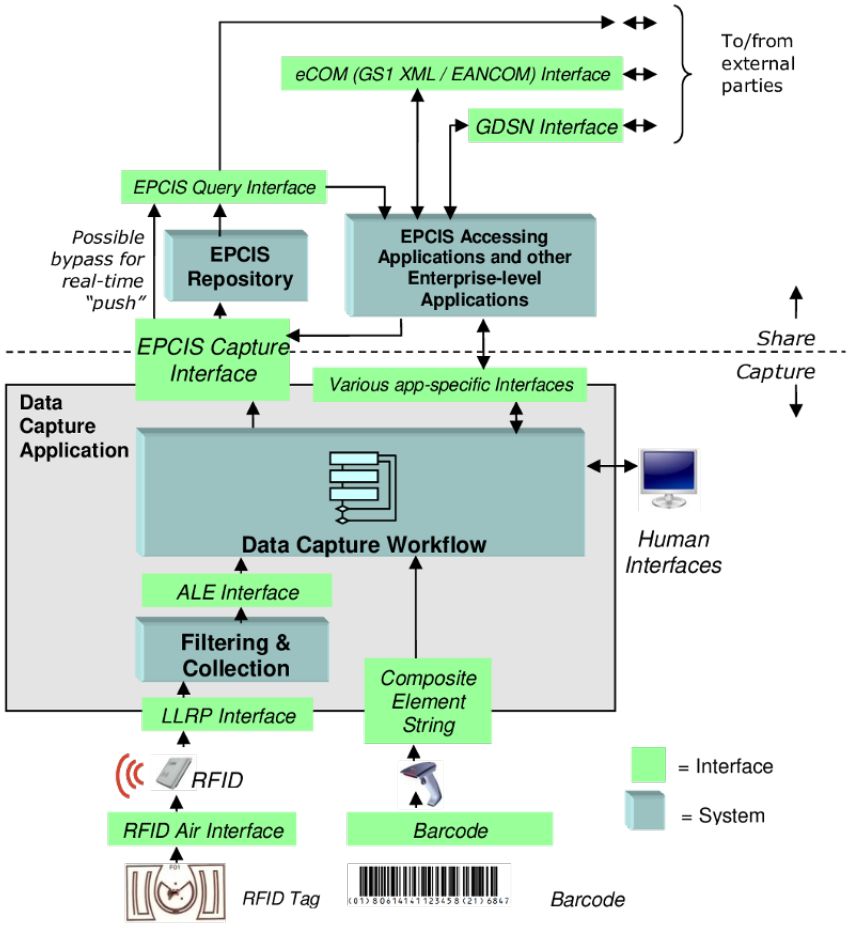

Figura 3.3. Flusso EPCIS in un'azienda

3.2 CBV

Il Comprehensive Business Vocabulary (**CBV**) fornisce valori standardizzati per gli elementi del vocabolario utilizzati all'interno di EPCIS. Quando tutti i dati sono conformi allo standard c'è una maggiore interoperabilità tra le aziende e in generale tra gli attori che utilizzano lo standard EPCIS.

Ci sono però casi in cui gli utenti finali (o i gruppi commerciali), hanno la necessità

di estendere questo vocabolario. Per questo vengono definiti due livelli di conformità per i documenti EPCIS:

- *CBV-Compliant*: documento che utilizza solo vocaboli CBV già presenti nello standard. Un applicazione per essere CBV-compliant deve soddisfare due requisiti:
  - se l'applicazione si dichiara essere CBV-Compliant per ogni documento in input, essa deve poter accettare qualunque documento CBV-Compliant come input;
  - se l'applicazione dichiara di pordurre documenti CBV-Compliant in output, allora essa deve produrre solo documenti del genere in output;
- *CBV-Compatible*: documento che utilizza un mix di vocaboli standard e altri vocaboli da standard esterni

3.3 EPCIS 2.0

Rispetto alla versione precedente dello standard, la 1.2, la nuova versione introduce novità sotto molti punti di vista.

3.3.1 Una nuova dimensione, how

La prima di tutte è l'introduzione di una quinta dimensione: **how**. La nuova dimensione da informazioni aggiuntive sulle condizioni in cui è stato scatenato l'evento EPCIS, come ad esempio quale era il contesto ambientale in cui questo evento è stato scatenato, che può essere utile anche come supporto per i dati acquisiti dai sensori negli eventi EPCIS.

3.3.2 Maggior supporto per le tecnologie Web

Questa nuova versione supporta la cattura dei dati in formato JSON, in opposizione al solo XML della versioni precedenti. Quest'ultima rende inoltre possibile l'introduzione al il supporto per API REST. Precedentemente veniva supportato solo SOAP, un protocollo per lo scambio di messaggi tra componenti software basato su XML.

3.3.3 Supporto a GS1 Digital Link

GS1 Digital Link è uno standard simile a un URL, il classico indirizzo di ogni sito web. GS1 Digital Link può abilitare collegamenti a tutti i tipi di informazioni business-to-business e business-to-consumer [7]. L'introduzione di questo URL nasce dall'esigenza del consumatore di sapere cosa si sta acquistando, ma la scansione dei prodotti rimanda a link obsoleti e non sempre pertinenti. GS1 digital link è nato appositamente per risolvere questo problema. Utilizzando questo nuovo standard e scannerizzando i QR code, il consumatore potrà ora accedere ad una grande quantità di informazioni sul prodotto, come ad esempio dimensioni, immagini, date di scadenza, dati nutrizionali, garanzia, istruzioni per risolvere problemi e persino link ai social media. GS1 Digital Link va a completare il tradizionale codice a barre GS1 e apre le porte alla possibilità di migrare in futuro verso un unico codice a barre abilitato per il web.

3.3.4 Compatibilità con i sensori IoT

L'introduzione della nuova dimensione *how* e quindi l'avere maggiore informazioni sul contesto ambientale in cui l'evento EPCIS è stato scatenato, permette di allargare gli use cases di EPCIS, aprendo le porte al mondo dell'IoT e dell'industria 4.0. Un esempio di use case può essere quello della misurazione della temperatura in una stanza

3.3.5 Miglioramento del CBV

Il miglioramento del Core Business Vocabulary (CBV) consiste nell'inclusione di nuovi attributi custom al vocabolario, che supportano il formato JSON, aggiornandosi di pari passo con EPCIS 2.0

Capitolo 4

Architettura generale e tecnologie di iChain for IoT

iChain è una piattaforma cloud e routing di eventi EPCIS. L'architettura pre-esistente era pensata per lavorare con eventi EPCIS nella versione 1.2 e non prevedeva ancora l'utilizzo di TimescaleDB (vedremo tra poco) per l'immagazinamento dei dati dai sensori IoT. Il lavoro di questa tesi mira quindi ad ampliare le funzionalità di iChain. L'architettura per questo nuovo servizio può essere divisa in 3 parti:

- I **dispositivi IoT**, che hanno dei sensori , a bassa potenza quali temperatura, umidità ecc. Questi dispositivi, collegati ai dei nodi intermedi, formano una rete BLE. I nodi intermedi hanno il compito di amplificare il raggio dei dispositivi, che in una rete BLE è di massimo dell'ordine di un centinaio di metri. Questi nodi intermedi infine portano le informazioni ad un gateway che manderà i dati raccolti al back end;

- Un **backend** che ha il compito di prendere i raw data che vengono mandati dal gateway dei dispositivi IoT, validarli e processarli, nonchè servire le richieste dei client di questa piattaforma;

- Un **front-end** con la quale le aziende possono visualizzare i dati raccolti tramite i dispositivi IoT e analizzarne l'andamento per il processo decisionale;

Il backend tramite un interfaccia di *capturing*, prima di tutto valida i dati che arrivano dal gateway dei sensori, questi dati grezzi verrano poi memorizzati in TimescaleDB e tramite un connector, verranno poi copiati in un topic di Kafka e consumati da Agent di Faust. Tutte le API sono state realizzate utilizzando FastAPI, e la validazione dei modelli viene fatta tramite Pydantic. L'autenticazione avviene grazie all'utilizzo di uno standard Internet che prende il nome di JSON Web Token (**JWT**). Il linguaggio di programmazione usato è Python. Approfondiamo ora le principali tecnologie utilizzate.

4.1 Apache Kafka

Apache Kafka è un publish subscribe messaging system distribuito [10], concepito per avere alte prestazioni sulle data pipeline e per gestire lo streaming dei dati in tempo reale. Usato da migliaia di aziende è scritta in JAVA e SCALA ed è open-source. Kafka può gestire grandi moli di dati in maninera sicura ed affidabile permettendo il consumo dei suddetti dati anche in modalità offline.

Kafka in quanto piattafroma di streaming di eventi combina in se 3 concetti chiave:

- **Scrittura** e **lettura** di streaming di eventi (*Publish and Subscribe* rispettivamente) includendo import ed export in tempo reale di dati da altri sistemi;
- **Memorizzazione** di questi eventi in maniera duratura e affidabile;
- **Processamento** degli eventi in tempo reale e retroattivamente se ce ne fosse la necessità;

Essendo una piattaforma distribuita, Kafka è composta da una serie di server e client che comunicano tra loro in maniera molto efficiente:

- **Server**: Kafka lavora in un cluster formato da molti server distribuiti in diversi datacenter. I server che hanno il compito di memorizzare gli eventi vengono chiamati *broker*, quelli che invece importano ed esportano i dati, quindi che gestiscono l'integrazione con gli altri sistemi, fanno girare i *Kafka Connect*. Il cluster è costruito in modo tale che se uno o più server presentassero un crash, altri prenderanno il loro posto e continueranno a fare il lavoro dei server non più lavoranti. In questo modo si garantisce il fault tolerant e l'alta scalabilità del sistema;
- **Client**: permettono di scrivere applicazioni che leggono e processano lo streaming di eventi, anche in parallelo, sempre in maniera scalabile e fault-tolerant. La libreria utilizzata per costruire queste applicazioni (o microservizi) è *Kafka Streams* ed è disponibile per un ampia varietà di linguaggi, tra cui Python;

4.1.1 Concetti principali e terminologia

Un evento sta a significare "è successo qualcosa". Concettualemente un evento ha una chiave, un valore e un timestamp e quando noi leggiamo o scriviamo qualcosa da Kafka, lo facciamo tramite il processing di questi eventi.

Prima di entrare nel dettaglio con la spiegazione del funzionamento di Apache Kafka, è opportuno introdurre un po' di terminologia:

- *Topic*: posto dove gli eventi sono organizzati in maniera persistente. Spiegato in maniera molto semplice, il topic può essere paragonato ad una cartella del file system di un calcolatore, e gli eventi sono l'analogo dei file al suo interno. Un topic può essere composto da una o più partizioni, che sono il principale meccanismo di concorrenza dei topic, come illustrato in figura 4.1;

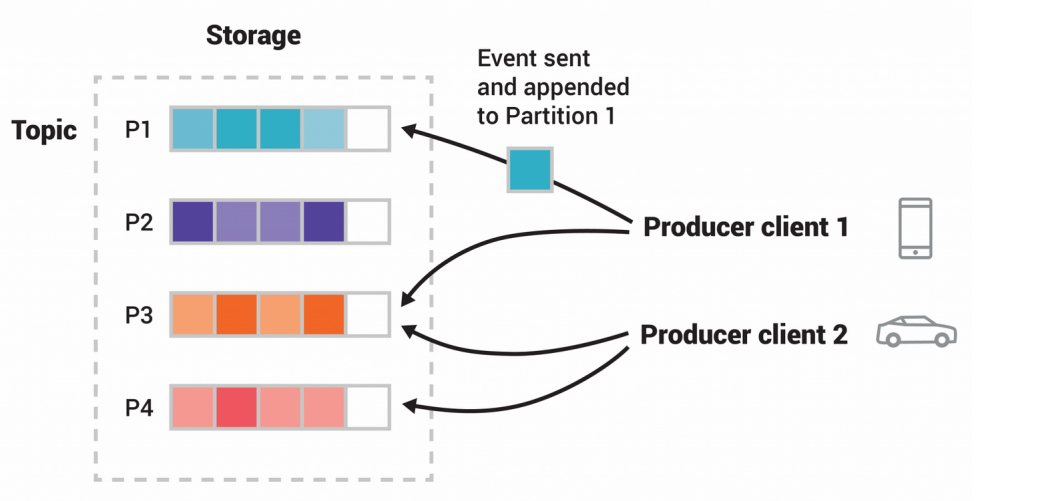

Figura 4.1. Esempio di un topic con 4 partizioni e due producers. Più producers possono pubblicare nella stessa partizione e sono agnostici sul fatto che ci siano altri producers che pubblicano insieme a lui in una stessa partizione;

- *Partizione*: permette a producers e consumers di operare in maniera concorrente sul topic. Le partizioni permettono anche ad un topic di essere distribuito su più Kafka brokers. Questa distribuzione è molto importante per la scalabilità del sistema, perché permette alle applicazioni di leggere e scrivere dati contemporaneamente, su più brokers. Quando un evento viene scritto in un topic, esso viene posizionato fisicamente in una partizione. Gli eventi con la stessa chiave vengono messi nella stessa partizione e Kafka garantisce inoltre che ogni evento venga letto nello stesso ordine con la quale è stato scritto;
- *Replica della partizione*: significa semplicemente mantenere una copia dei dati sul cluster in modo da tenere dei "backup". La replica in Kafka è a livello di partizione. Ogni partizione ha 0 o più repliche nel cluster.
- *Producers*: Coloro che pubblicano un evento in uno o più topic. Il Producer può decidere se scrivere l'evento in una partizione specifica;
- *Consumers*: Coloro che consumano i messaggi dai brokers. Essi si sottoscrivono a uno o più topic e consumano i messaggi pubblicati. In Kafka, producers e consumers sono completamente disaccoppiati e indipendenti l'uno dall'altro, il che è un elemento chiave del design per ottenere l'elevata scalabilità per cui Kafka si contraddistingue;
- *Leader*: Nel momento in cui c'è la replicazione di un topic, che viene fatta a livello di partizione, una di queste repliche viene eletta a leader, mentre le altre vengono elette a followers (spiegati nel punto seguente).

  Il leader è quindi il nodo responsabile per tutte le scritture e le letture su quella data partizione;

- *Follower*: Sono delle repliche che copiano quello che succede nel leader, nel momento in cui il leader non è più in grado di operare uno dei follower prende il suo posto;
- *Kafka Consumer Group*: Un gruppo di consumatori che condivide il lavoro. Il concetto è pensato per migliorare l'efficienza dividendo il lavoro tra i Consumers;
- *Offset*: sono una sequenza di id che vengono dati internamente agli eventi dentro una partizione. Questi id non sono globali, sono solo locali all'interno della partizione;

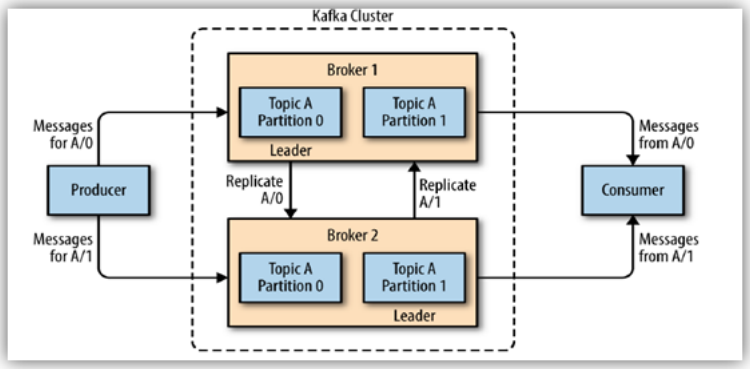

Figura 4.2. Architettura di Kafka (single cluster) [11]

4.1.2 Zookeeper

Per poter funzionare correttamente, Kafka ha bisogno di **Zookeeper**. ZooKeeper è un servizio centralizzato per la gestione delle informazioni di configurazione, la denominazione, la sincronizzazione distribuita e la fornitura di servizi di gruppo [3].

Kafka necessita di Zookeper per mantenere lo stato del suo cluster e per eleggere i leader. In altri termini Zookeeper coordina i broker Kafka all'interno del cluster, in modo tale da notificare i producers e consumers sulla comparsa di nuovi broker, oppure sul guasto di uno di essi. Quindi è lo stesso Zookeeper a eleggere il broker leader e in caso di guasto a sceglierne uno nuovo tra gli altri brokers, come possiamo vedere in figura 4.3,

Zookeeper è composto da 4 componenti principali:

- *Node*: I sistemi installati sul cluster;
- *ZNode*: I nodi in cui lo stato viene aggiornato da altri nodi nel cluster;
- *Client Applications*: Il tool che interagisce con l'applicazione distribuita;
- *Server Applications*: fornisce un'interfaccia per il Client Application;

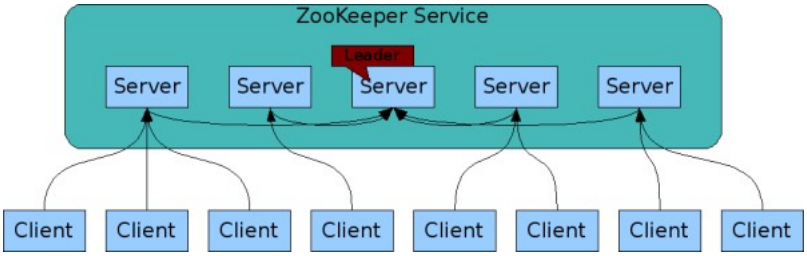

Figura 4.3. Servizio di Zookeeper per Kafka [9]

4.1.3 Vantaggi del processing in streaming

Presentiamo ora i vantaggi del processing in streaming, spiegando il motivo per cui esso è stato scelto come uno dei pilastri fondamentali nella progettazione e realizzazione di iChain. Definiamo prima cosa si intende per *data streaming*:

Il data stream è un'astrazione che rappresenta un insieme non limitato di dati [14].

Non limitato sta a significare "infinito" e "in continua crescita", questo perché nuovi dati sono sempre in arrivo. Oltre ad avere una natura non limitata, lo streaming di eventi è caratterizzato anche da altri attributi:

- *Ordinati*: Esiste un ordine con cui gli eventi occorrono, ovvero è sempre possibile dire quale evento sia occorso prima;
- *Immutabili*: Una volta che un evento è occorso non può più essere modificato. Questo vuol dire che se l'effetto di un evento vuole essere annullato, allora dovrà occorrere un altro evento che indicherà che il precedente evento non ha più validità;
- *Ripetibili*: Un evento è ripetibile, e questo è fondamentale per correggere errori e per provare nuovi metodi. Questa caratteristica è fondamentale se si vuole ripetere un evento che è occorso mesi o anni prima;

Ora possiamo procedere a capire il funzionamento del data streaming e i suoi vantaggi.

Prima di tutto bisogna notare come non viene menzionato da nessuna parte il tipo di data che stiamo analizzando, questo perché possiamo avere una grande varietà di possibilità: dal JSON all'XML, e applicabili a tutte le moli di dati, dai milioni di eventi per secondo a pochi eventi al minuto.

Il data streaming è un paradigma di programmazione che si pone a metà tra *Request-Response* (che è il paradigma che garantisce la latenza minore in assoluto, nell'ordine del millisecondo) e *Batch processing* (che è quello con latenza maggiore, anche dell'ordine delle giornate). Molte applicazioni di business non richiedono una risposta nell'ordine dei millisecondi (come nel primo caso) ma non possono nemmeno attendere una quantità di tempo nell'ordine dei giorni (come nel secondo caso), inoltre molti processi avvengono

in maniera continua, ovvero si tratta di un flusso continuo nel tempo di dati (come nel caso dei dispositivi IoT con i sensori in essi incorporati). Questo fa si che non è possibile utilizzare il Request-Response in quanto bloccante.

Il data streaming inoltre gira intorno a 4 concetti chiave:

- *Tempo*: Concetto più importante nel data streaming, perché non è possibile definire un unica nozione temporale, ne vengono definite 3 e sono:
  - Event time: Tipicamente la nozione di tempo che conta di più nel processing, si riferisce al timestamp dell'evento, ovvero quando esso è occorso;
  - Log append time: Istante temporale in cui l'evento arriva ad un broker Kafka (nella nostra trattazione)
  - Processing time: Instante di tempo in cui il dato viene poi processato dall'applicazione;
- *Stato*: In alcuni casi, non è importante ne di interesse analizzare il singolo evento, piuttosto c'è la necessità di analizzare gli eventi con una granularità superiore (es. quanti eventi di questo tipo sono avvenuti nell'ora precedente?). Le informazioni memorizzate che si riferiscono a questi tipi di operazioni, vengono chiamate stati;
- *Dualità table-stream*: Sono due facce della stessa medaglia, la tabella in un database cotiene record che sono mutabili nel tempo, ma quando si va ad interrogarla si ottiene solamente lo stato corrente delle cose. Al contrario lo stream contiene tutta la storia (tutta la serie di eventi) che ha portato le cose allo stato corrente. Avere entrambe le rappresentazioni è molto meglio che averne una sola, e Kafka (nel nostro caso) consente di passare da una rappresentazione ad un'altra;
- *Finestre Temporali*: Molte operazioni sugli stream, come detto anche in precedenza, avvengono su finestre temporali (si pensi alla media di un valore in un intervallo di tempo). Queste finestre temporali possono essere di due tipi, *Tumbling* e *Hopping*;

Per concludere questa trattazione, andiamo ad analizzare i motivi per cui lo stram processing è fondamentale per iChain.

Come abbiamo già detto, i sensori dei dispositvi IoT mandano in continuazione dati e lo scopo è quello di monitorarli per rilevare anomalie o semplicemete per fare delle analisi. Lo scopo è quindi quello di monitorare una grande quantità di dati e analizzarli in un tempo ragionevole, senza però rendere bloccanti le operazioni di processing, e in una maniera ordinata, così come vengono mandati. Tutto questo in maniera resiliente a possibili complicanze esterne ad iChain, come ad esempio un down della rete.

4.2 TimescaleDB

Per quanto riguarda la memorizzazione dei *Raw Data* provenienti dal gateway dei dispositivi IoT si è scelto di utilizzare **TimescaleDB**.

TimescaleDB è un database relazionale per dati basati sul tempo [18]. Usa completamente SQL e il suo utilizzo è semplice come quello di ogni database relazionale, ma con la scalabilità di un database NoSQL. Esso è implementato come estensione di PostgreSQL, ovvero può lavorare in un qualunque server PostgresSQL come parte dello stesso processo.

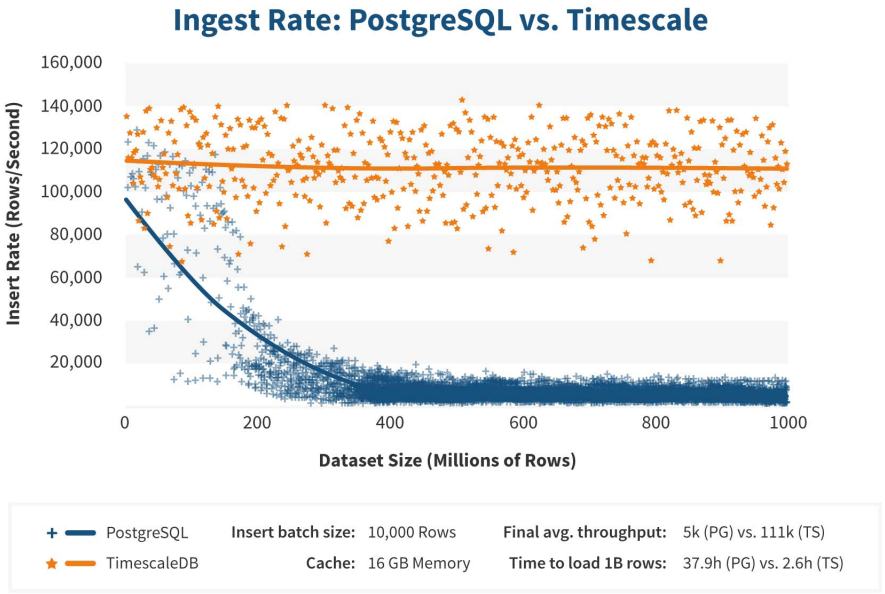

Figura 4.4. Paragone tra TimescaleDB e PostgreSQL sul Tasso di ingestion

Concetti chiave

Tra i punti di forza di questo database possiamo elencare i seguenti:

- *Hypertable*: A livello logico, tutte le interazioni tra utente e DB sono fatte tramite hypertable. Essa provvede a realizzare un'astrazione della tabella standard SQL. Nella realtà però la hypertable è una visione logica di molte tabelle individuali, dove sono realmente e fisicamente memorizzati i dati, chiamate *chunks*. Le hypertable sono tipicamente partizionate in chunk tramite la colonna del tempo, ovvero un chunk può avere ad esempio tutti i valori compresi in uno specifico intervallo di tempo. Sono possibili anche altri chiavi di divisioni in chunk (come in base ad un semplice id). I chunk vengono creati in maniera automatica da TimescaleDB mano a mano che nuovi dati vengono inseriti nella hypertable: Timescale decide autonomamente se un nuovo record deve essere inserito in un chunnk esistente oppure se è opportuno crearne uno nuovo. Questa divisione in chunk porta con se numerosi benefici, tra cui: utilizzo di

indici locali al singolo chunk, facile conservazione dei dati, facile riordino dei dati e infine una più facile migrazione e replica dei dati (entrambi vengono fatti a livello di chunk). Infine altra caratterstica principale delle hypertable in TimescaleDB, è che esse possono essere distribuite, ovvero possono anche risiedere su più nodi;

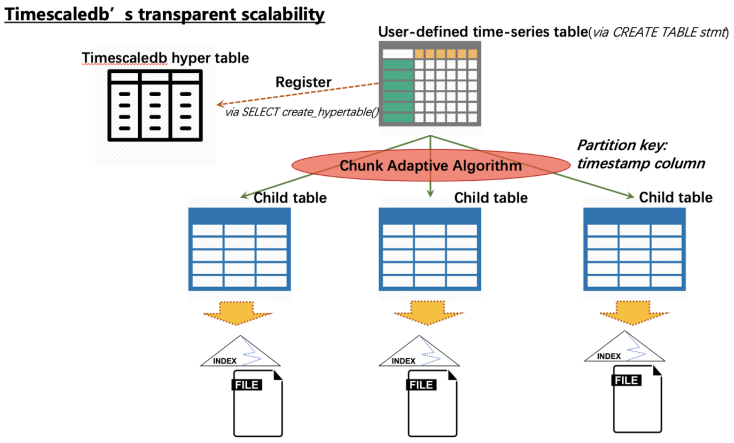

Figura 4.5. Chunk all'interno dell'architettura di Timescale [4]

- *Performance accelerate*: Sulle query, raggiuge velocità dalle 10 alle 100 volte superiori a PostgreSQL, InfluxDB e MongoDB. TimescaleDB utilizza una serie di ottimizzazioni per raggiungere queste performance, tra cui la parallelizzazione delle query, le continuous aggregates e la divisione in chunk per le *hypertable*;
- *Grande scalabilità*: è possibile scrivere milioni di dati al secondo, con la possibilità di avere centinaia di Terabyte memorizzati in un sigolo nodo, o migliaia di Terabyte in più nodi. Questa grande scalabilità è elargita anche grazie alle hypertable distribuite;
- *Compressione Nativa*: La compressione delle serie temporali in Timescale non richiede nessun software esterno o aiuto di un file system. La compressione viene effettuata a livello di chunk, dove si passa dalla forma a righe (nessuna compressione) alla forma a colonne (con compressione). Timescale, per decidere quali chunk comprimere, va a vedere quelli vecchi abbastanza, ovvero valuta quali chunk saranno meno oggetto a delle query nel breve termine. Trovata la "vittima", essa verrà man mano convertita nella forma a colonna. Dal punto di vista dell'utente questa cosa è completamente trasparente, nel caso in cui quest'ultimo facesse query che riguardino chunk compressi, essi verranno prima decompressi e poi utilizzati.
- *Continuous aggregates*: Sono qualcosa di simile a quelle che in PostgreSQL sono le *View Materializzate*. Il problema che cerca di affrontare la Continuous aggregates è quello di rendere l'esecuzione delle query più veloce quando si tratta di scansionare un grande ammontare di dati. Al contrario delle viste materializzate di PostgresSQL, una continuous aggregates può essere aggiornata in modo continuo e incrementale, sia in maniera manuale che tramite politiche in background;

4.3 Faust

Faust è una libreria di processing in streaming in Python, che si inspira a Kafka Streams [13]. Faust utilizza solamente Python (nelle versioni dalla 3.6 in poi, per la necessità di utilizzare i costrutti di async/await che sono stati introdotti da quella versione), questo vuol dire che insieme a Faust possiamo utilizzare qualunque libreria Python esistente per processare gli stream di eventi.

4.3.1 Concetti chiave

Come abbiamo già detto Faust si basa su Kafka Stream e mette a dispozione delle astrazioni di alto livello in Python per poter sia consumare da un topic Kafka (comportarsi da Consumer), che per poter scrivere su un topic Kafka (comportarsi da Producer). In altri termini Faust permette di lavorare sul data stream di Kafka tramite Python. Le astrazioni di alto livello che rendono possibile lavorare sugli stream sono essenzialmente le seguenti:

- *Application*: Prima di poter fare qualunque cosa in Faust, è necesseraio instanziare un applicazione, che fornisce le API con cui utilizzare Faust. L'applicazione sarà colei che poi definirà gli agent, i topic, le table ecc.;
- *Agent*: Un agente è un sistema distribuito che elabora gli eventi in uno stream. Gli agent sono delle funzioni asincrone che hanno il controllo completo sulle iterazioni in un stream (un topic o un channel più nello specifico). Un agent viene associato ad uno stream e ne diventa il suo consumer. Possono sia scrivere su un topic (diverso da quello in cui sono *attached*, che leggere da esso (cioè da quello a cui sono *attached*, come in figura 4.6;

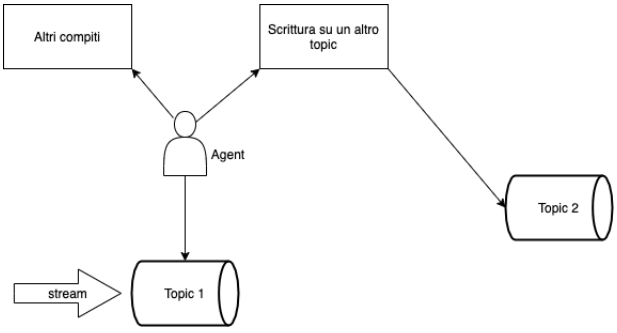

Figura 4.6. Piccolo esempio di interazione tra agent e topic

- *Channels and Topic*: I topic sono molto più specifici di Kafka, mentre i channel no. In Faust, un agent lavora sugli stream, ma gli stream iterano su dei channel. Detto in altri termini, uno stream legge da un channel che sarebbe una sorta buffer/coda dove i messaggi (gli eventi) vengono letti oppure scritti. Un topic invece non è nient'altro

che un channel a cui è stato dato un nome, e fanno riferimenti ai topic veri e propri presenti in Kafka;

- *Table*: Una table è un dizionario distribuito collocato in memoria grazie a RocksDB e sostenuto dai change log di Kafka. La table viene usata per tenere informazioni che siano persistenti anche in presenza di un crash del sistema;

- *Task and Timer*: Un applicazione Faust, oltre ad avere degli agent che consumano sui topic o sui channel, può anche far partire task asincroni per fare altre operazioni che prescindono dagli stream. Uno dei task più utilizzati è proprio quello del Timer, che permette di eseguire operazioni a intervalli di tempo precisi, prestabiliti durante la definizione della stesso Timer;

4.3.2 RocksDB

Uno dei principali motivi del successo di Faust come libreria di processing in streaming, è senz'altro RocksDB. RocksDB è un database integrato ad alte prestazioni, scritto in C++, che fornisce un sistema di key-value storing [16]).

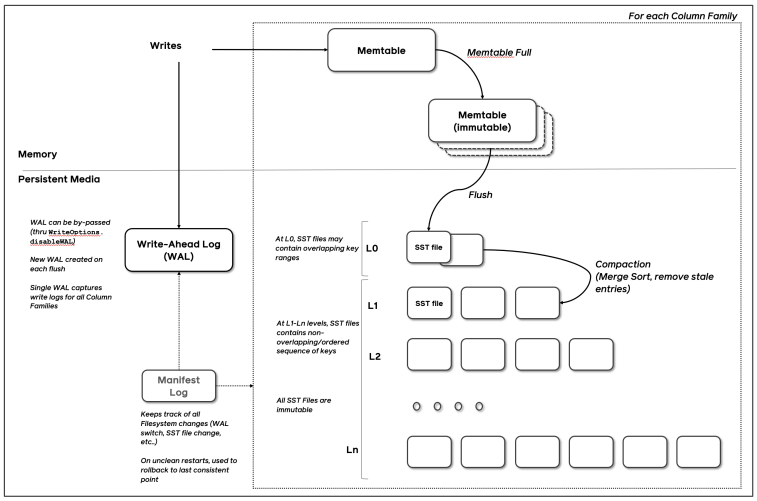

Figura 4.7. Architettura ad alto livello di RocksDB [17]

Caratteristiche chiavi di questo DB sono la capacità di memorizzazione di dati da fonti con throughput molto elevato, quali possono essere appunto degli stream di eventi, nonchè la capacità di supportare un ampia varietà di hardware per quanto riguarda lo storage. RocksDB ha un Write-Ahead-Log (WAL) nel quale sono inserite tutte le operazioni di Inserzione, Aggiornamento e Cancellazione.

4.4 FastAPI

L'ultima tecnologia utilizzata in questo lavoro di tesi, e che andremo ad introdurre, è quella di FastAPI. FastAPI è un framework web per la costruzione di REST-API in Python (dalla versione 3.6 per la necessità di utilizzare i costrutti di sincronizzazione async/await) [5].

Le caratteristiche principali sono:

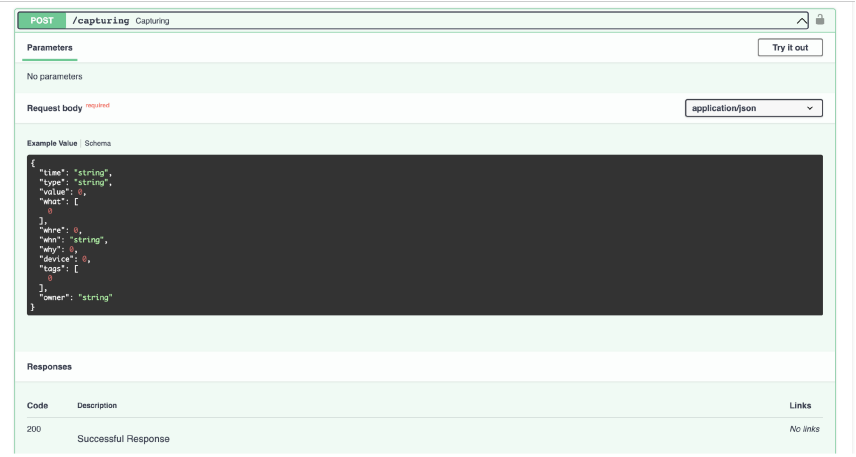

Figura 4.8. Swagger FastAPI

- *Veloce*: Performance molto elevate grazie anche all'utilizzo Starlette e Pydantic (vedremo meglio nel prossimo capitolo), che lo rendono uno dei framework più veloci disponibili per il linguaggio Python;
- *Programmazione veloce*: Permette di diminuire fino al 300% il tempo di scrittura del codice, permettendo allo sviluppatore di concentrarsi solamente sulla logica finale delle proprie API;
- *Minori Bug*: Questa è una diretta conseguenza del punto precedente. Facile da utilizzare, si è stimata una riduzione del circa 40% sugli errori di programmazione;
- *Facile e intuitivo*: Progettato per essere di facile apprendimento, che significa spendere meno tempo sulla consultazione della documentazione;
- *Compatto*:Viene minimizzato il codice evitando duplicazioni inutili, ne consegue una minore presenza di bug nel codice;
- *Robusto*: Scrivendo codice già pronto per entrare in produzione;
- *Standard-based*: Completamente compatibile con gli standard OpenAPI e JSON Schema. OpenaAPI è uno standard che permette di avere un interfaccia per capire

le funzionalità delle API senza avere accesso diretto al codice. OpenAPI è conosciuto come *Swagger*, mostrato in figura 4.8, e nel presente lavoro di tesi se ne è fatto ampio utilizzo per testare le API. JSON Schema invece è uno standard per la validazione dei documenti JSON;

In questo capitolo sono stati introdotte le principali tecnologie utilizzate durante lo sviluppo del backend per i sensori IoT. Un riassunto dell'architettura finale che ne deriva può essere vista in figura.

I dati raw che provengono dal gateway vengono ingeriti nel sistema tramite un in-

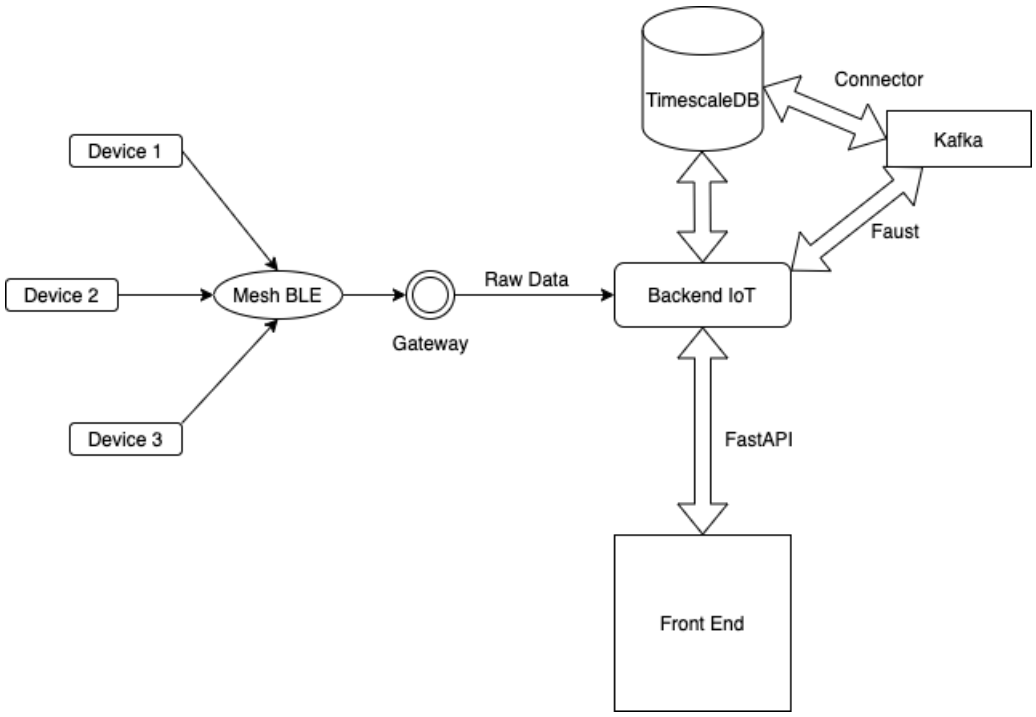

Figura 4.9. Monitoraggio IoT

terfaccia di capturing che li valida e li inserisce nel database. Poi c'è un connector logico con la quale Kafka riesce a rilevare cambiamenti nelle tabelle su cui è in ascolto e consuma questi dati tramite Faust. Il backend riesce a servire il client tramite le API web che esso espone tramite FastAPI. Le interfaccie di FastAPI non sono solo rivolte verso il front end, bensì anche verso il gateway, per la presenza appunto dell'interfaccia di capturing.

Timescale, Kafka, Faust, TimescaleDB, MongoDB e il backend principale della piattaforma di iChain sono tutti contenereizzati tramite docker, in modo da renderne più facile e veloce il deployment della piattaforma. Insieme ad essi è anche presente un container per Kafdrop, un'interfaccia grafica che si collega a Kafka e che permette di visualizzare topic e messaggi all'interno di quest'ultimo in maniera semplice e intuitiva. Il docker compose

di questi container viene effettuato tramite degli scritti bash presenti nella piattaforma e che consentono un rapido e semplice avvio del sistema da terminale.

Nel prossimo capitolo esploriamo in dettaglio il lavoro effettuato sul backend per il monitoraggio, analizzando come sono state combinate queste tecnologie tenendo sempre in considerazione il pattern del Data streaming sulle serie temporali che stiamo trattando.

Capitolo 5

Backend per i sensori IoT

Entrando nel dettaglio del lavoro svolto in questa tesi, in questo capitolo verrà affrontato la questione di come si è scelto di procedere durante la fase di sviluppo, e quali risultati sono stati raggiunti. Una prima analisi ha riguardato la scelta del database, come già descritto nel capitolo in precedenza si è scelto di utilizzare TimescaleDB perchè concepito per la gestione delle serie temporali. In seguito si è proseguito con l'analisi e l'impostazione delle tabelle che esso avrebbe dovuto ospitare, in base alle esigenze del data streaming, e delle query che esse avrebbero dovuto soddisfare, scegliendo in base ad esse gli indici da utilizzare per ottimizzare il tempo di risposta alle query, nonchè le inserzioni, le rimozioni e gli aggiornamenti dei record. Un altro aspetto importante ha riguardato le prestazioni ottenute in caso di ALTER TABLE mentre erano in corso operazioni di inserzioni. Nel seguito del capitolo verrà presentato un nuovo modello dati per EPCIS 2.0, il lavoro effettuato per i master data, ed infine il lavoro vero e proprio sul processing dei RawData.

Partiamo quindi con l'analisi e la struttura della tabella per i rawData.

5.1 Analisi sulla configurazione delle tabelle TimescaleDB

Il primo obiettivo è stato quello di trovare una corretta configurazione per la tabella in TimescaleDB che avrebbe ospitato i RawData. La tabella in questione ha il principale compito di memorizzare i dati raw provenienti dal gateway, quindi si tratta di avere una tabella che riesca sia a gestire l'ingente mole di dati che arrivano, sia a modellarne il contenuto in maniera coerente alle esigenze dell'utente finale.

La prima cosa analizzata è stata la possibilità di inserire nuove colonne all'interno della tabella, questo perché in fase di sviluppo non si era ancora certi al 100% della struttura finale dei record rawdata. Emergeva quindi il problema di dover scegliere tra colonne semi-strutturate di tipo JSON, oppure colonne con tipi standard di PostgreSQL. PostgreSQL e di conseguenza TimescaleDB, supportano i tipi JSON e JSONB, e la differenza tra i due è solamente che il primo memorizza il dato JSON come un esatta copia del JSON

passato in input, mentre il secondo memorizza una versione binaria del JSON passato, consentendo in più operazioni custom su questi dati. Utilizzando i JSON quindi sarebbe stato possibile non dover mai effettuare nessuna operazione di ALTER TABLE. Per verificare quale delle due alternative avrebbe consentito un risparmio in termini di memoria e di velocità delle query, sono stati effettuati dei test. I test consisitevano nell'effettuare operazioni di inserzione e di SELECT su due tipi di tabelle, la prima collassando tutti i campi che sarebbero potuti essere variabili in numero, come i tag, in una colonna di tipo JSONB, e la seconda con una colonna standard per ogni campo. Su quest'ultimo tipo di tabelle sono state inoltre effettuate operazioni di ALTER TABLE durante le già menzionate inserzioni, per verificarne la velocità di esecuzione e l'overhead che portavano con se. I risultati venuti fuori hanno dimostrato come la tabella con i campi standard abbiano prestazioni nettamente migliori nelle operazioni di SELECT e in termini di memoria, a fronte di un leggero svantaggio in termini di tempo nelle operazioni di inserzione. Dato che, anche le ALTER TABLE presentavano solamente un piccolo overhead al sistema, si è optata come alternativa la tabella con campi standard.

5.1.1 Struttura della tabella RawData

Come risultato finale, la tabella TimescaleDB per i rawData ha la seguente struttura:

- *id*: Identificativo univoco e seriale che viene incrementato ad ogni inserzione e prescinde dal resto del record, in quanto è stato introdotto principalmente per creare il connettore manuale tra TimescaleDB e Kafka, vedremo meglio nel seguito del capitolo;
- *Time*: Timestamp di invio del dato da parte del gateway;
- *Type*: indica il tipo di misurazione effettuata in base al sensore, ad esempio può essere un sensore di temperatura, uno di umidità e così via;
- *Value*: indica il valore della misurazione;
- *4w*: rappresentano le dimensioni 4w di EPCIS e sono:
  - *what*: dato che questa dimensione contiene uno o più identificatori univoci, la sua colonna è di tipo vettore e può contenere da uno a N interi;
  - *where*: Questa colonna è rappresentata come un intero, ovvero l'identificativo univoco del luogo in cui l'evento ha avuto successo;
  - *when*: Questa colonna è di tipo timestamp e rapprensenta l'esatto momento in cui il dato è stato generato dal dispositivo;
  - *why*: visto come un intero univoco che indica il motivo per cui il dato è stato generato;
- *device*: indica il device ID del dispositivo, quindi è rappresentato come un intero;
- *tags*: è un vettore che rappresenta dei tag, non noti a priori, associati alla misurazione. Possono essere utili in fase di visualizzazione da parte del client, che potrebbe filtrare contenuti anche in base a questi tag;

- *owner*: è l'owner del device che ha effettuato la misurazione, si tratta di una colonna di tipo stringa e il motivo principale per cui è stato introdotto verrà spiegato meglio nel seguito del capitolo;

5.2 Tabella per EPCIS 2.0

Anche per quanto riguarda la nuova versione di EPCIS è emersa la necessità di avere un modello che riuscisse a validare gli eventi e un database in cui conservarli. iChain come database principale per gli eventi utilizza MongoDB, in esso vengono già memorizzati, molto efficacemente, gli eventi nella versione precedente, e di conseguenza la scelta è ricaduta nuovamente su di lui. MongoDB è un database non relazionale tra i più utilizzati al giorno d'oggi, in grado di memorizzare dati in un formato simile a quello JSON e che presenta molti vantaggi in termini di efficienza e di affidabilità.

Per quanto riguarda il modello invece, si è fatto un forte utilizzo di **Pydantic**. Pydantic è una libreria per la data-validation e il setting management [15] che mappa i dati in arrivo in una classe Python. Consente di modellare e validare i dati in maniera semplice e intuitiva, inoltre fornisce errori user friendly quando i dati sono invalidi. L'utilizzo di questa libreria, come già detto, è molto semplice: una volta importata la libreria, basta creare una classe che eredita da *BaseModel* (Modello base di Pydantic), e che rispecchi negli attributi il dato/i che si vogliono modellare. È inoltre possibile personalizzare le impostazioni per questa classe grazie alla classe *Config*, in modo tale da avere la massima flessibilità nel modello.
Grazie al decorator *validator*, è possibile effettuare anche check più dettagliati su un campo del modello, quando ad esempio la semplice costrizione sul tipo non basta e il singolo attributo deve rispettare delle regole ben precise.
Per quanto riguarda Pydantic infine, c'è da sottolineare come sia possibile rendere opzionali dei campi (fondamentale per i campi EPCIS, come detto nel capitolo 3), oppure settare dei valori di default a degli attributi, che potranno poi essere o meno sovrascritti.

Tornando ad EPCIS dunque, grazie a queste due tecnologie si è potuto creare un nuovo modello per la nuova versione 2.0. Il modello che ne è derivato è rappresentato nelle figure 5.1, 5.2 e 5.3. Le parti del modello che contengono le informazioni più significative sono quella del sensorReport e del sensorMetadata. In esse infatti sono contenute le informazioni che andranno a popolare la collezione in MongoDB e serviranno nella creazione degli eventi EPCIS.

5.3 Connectors TimescaleDB-Apache Kafka

Creata l'infrastruttura in grado di ospitare i Raw Data, lo step successivo è stato quello di analizzare e realizzare un'infrastruttura di processing per quei dati.

Come già spiegato in precedenza, stiamo parlando di gestire una grande mole di dati secondo il paradigma del data streaming, ciò significa il dover attenersi ad una serie di

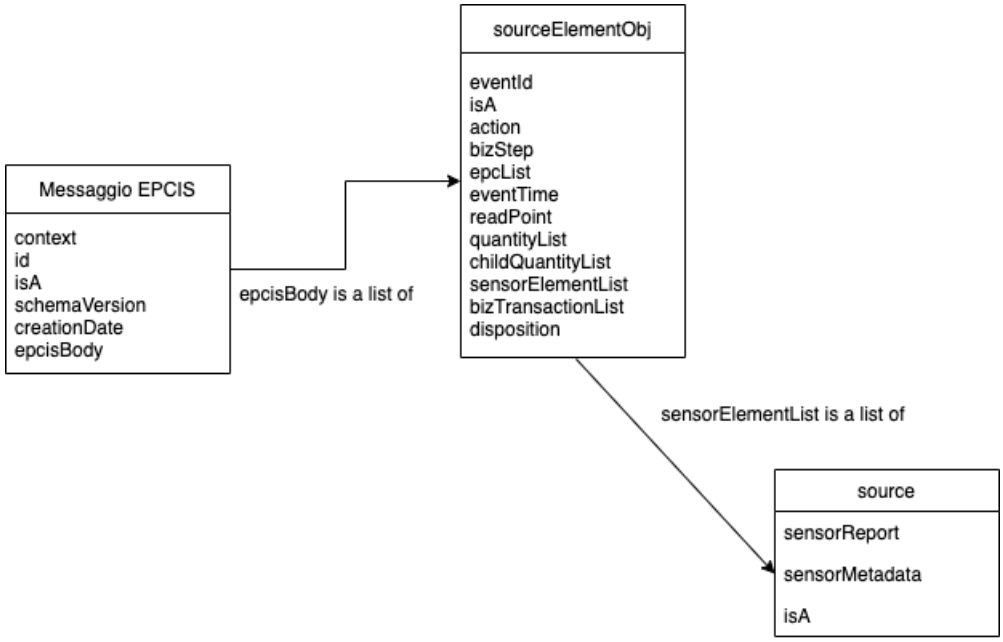

Figura 5.1. Modello epcis 2.0, parte 1

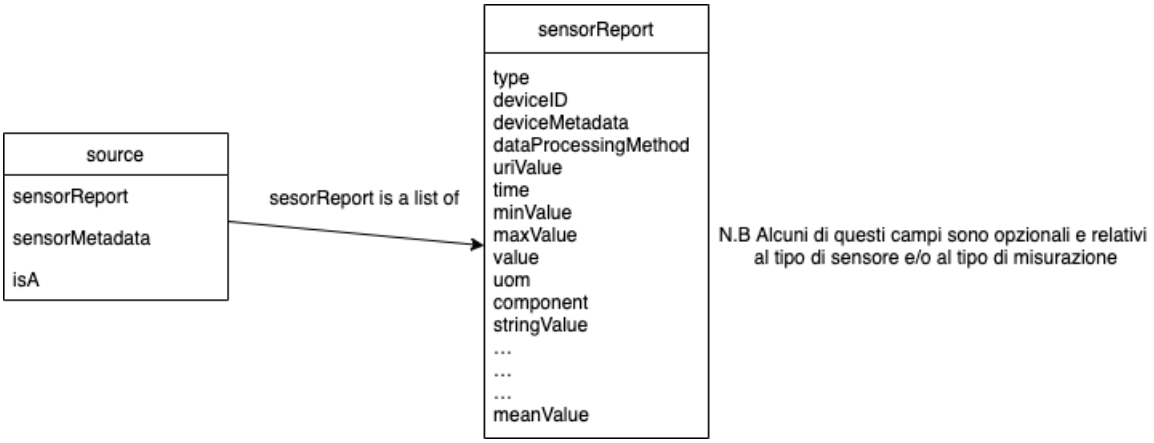

Figura 5.2. Modello epcis 2.0, parte 2

principi quali tempi di processamento contenuti e operazioni non bloccanti. In particolare, per ovviare a queste necessità, si creano dei Connectors che rendono le operazioni di trasferimento da database a backend non bloccanti, e immediatamente disponibili per il processamento anche sui topic Kafka. Più in dettaglio esistono 2 tipi di connectors:

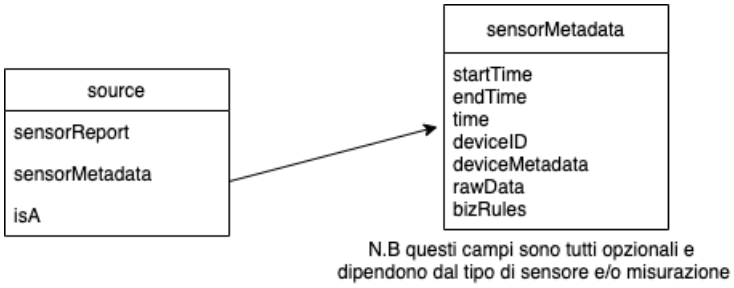

Figura 5.3. Modello epcis 2.0, parte 3

- Source Connectors: Connector che rileva cambiamenti in uno storage di dati, TimescaleDB nel nostro caso, e trasferisce queste informazioni automaticamente all'interno di un topic;

- Sink Connectors: Connector che viaggia nella direzione opposta al Source, ovvero trasferisce automaticamente i dati da un topic ad uno storage man mano che arrivano;

In questa sezione verrà analizzata l'implementazione del Source Connector tra TimescaleDB e Apache Kafka. Il diagramma a blocchi è mostrato in figura 5.4. Dopo aver memorizzato i dati all'interno del nostro store entra in gioco il Source Connector. Questo connettore è stato creato manualmente grazie ai timer Faust di cui abbiamo parlato nel capitolo 3. In particolare il timer è impostato per lanciare una funziona asincrona ogni secondo. La funzione asincrona è incaricata di stabilire una connessione con il nostro database tramite la libreria **asyncpg** [1]. Questa libreria è un'interfaccia per database PostgreSQL ideale per le operazioni asincrone, quali possono essere le operazioni all'interno di un DB, risultando anche 3 volte più veloce rispetto le principali alternative. Stabilita la connessione, si performa una query per rilevare possibili inserzioni all'interno della table. Per tenere il conto delle precedenti inserzioni, e quindi essere sicuri di selezionare solamente quei dati che ancora non sono stati selezionati, si utilizza una variabile globale in cui è contenuto l'id dell'ultimo record rilevato dal timer.

L'id è il campo *id* della tabella Raw Data già accennato nella prima sezione di questo capitolo, ed è stato introdotto appositamente per questa ragione. Durante la fase di analisi si è optata per questa soluzione in quanto utilizzare il timestamp del record avrebbe potuto generare dei buchi, ovvero un mancato prelevamento di dati. Il motivo va ricercato nella naturale incertezza che un sistema IoT dimostra nella send "ordinata" dei dati: un esempio potrebbe essere quello della mancata ricezione di misurazioni, da parte del gateway, dovuta magari a problemi sulla rete locale o altro, che fa si che potrebbe far arrivare al gateway più misurazioni contemporaneamente oppure far arrivare prima le misurazioni successive (in ordine di tempo). Il campo *id* invece assicura che tutti i record siano prelevati e processati.

Selezionati quindi solamente i record di nostro interesse, si procede con il fetching della

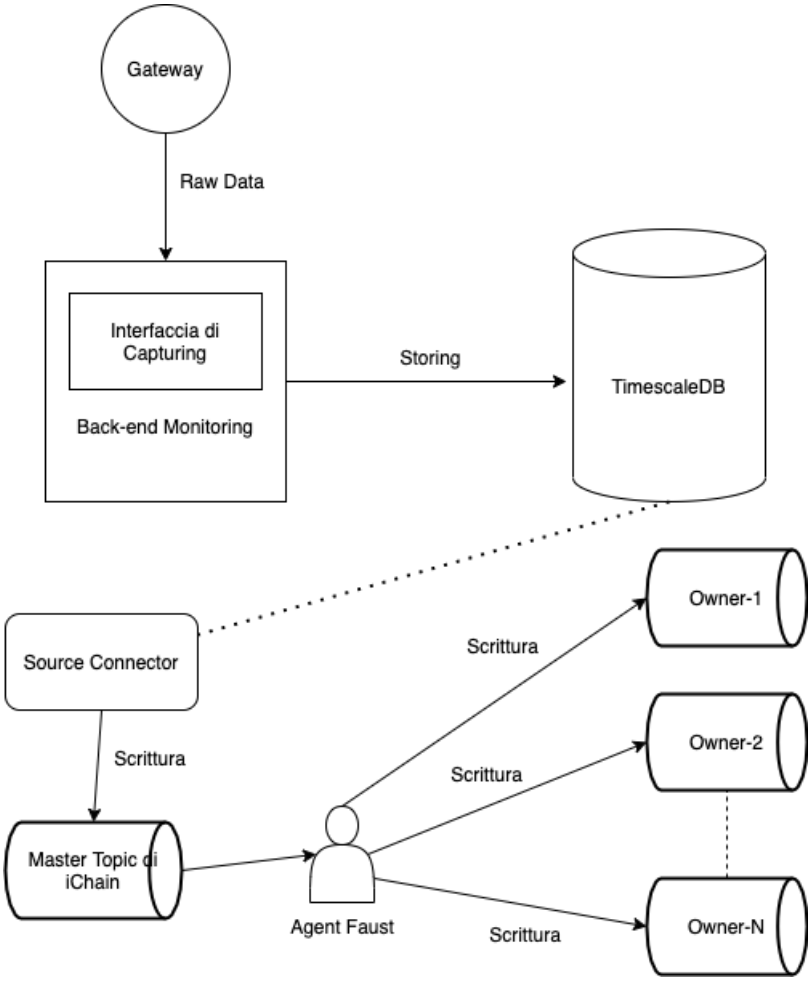

Figura 5.4. Source Connector TimescaleDB-Kafka

query una row per volta. Ogni row viene modellata grazie ad un modello Faust creato appositamente, che permette quindi di inserirne il contenuto all'interno di una classe Python, in modo tale da poterci lavorare su. Per ogni riga infine, l'oggetto temporaneo creato con il record viene inviato a quello che in figura 5.4 è indicato con "Master Topic di iChain". L'esistenza di un Master Topic è giustificata da due motivi:

- Alleggerire il più possibile il lavoro del Source connector, facendo in modo che si occupi solo del suo lavoro vero e proprio;
- Gestire con maggior efficienza e sicurezza dati in base al proprio owner;

Infine, la mancanza di un Master Topic, potrebbe portare a perdite di dati nel caso siano relativi ad un nuovo partner, in quanto non ci sarebbe nessun Agent o in generale una componente che rilevi automaticamente la presenza di un nuovo owner. Questo porterebbe

alla necessità di un intervento manuale da parte di un sistemista di iChain per creare il topic in questione.
Nel dettaglio, il processing dei messaggi all'interno di questo Topic è gestito secondo i seguenti step:

- Valutazione di chi sia l'owner;
- Costruzione del nome per la ricerca/creazione del topic: il nome viene costruito secondo una logica di prefissi di iChain, che consente in una più efficiente gestione dei topic all'interno di Kafka, considerando anche lo scopo ultimo dello stesso;
- Ricerca del topic tra quelli già esistenti e
  - Invio del record nel topic se esso è presente in una lista popolata con i topic;
  - Creazione del topic, inserzione nella lista e invio del record se esso non è trovato nella sopramenzionata lista;

Dopo quest'ultimo passaggio in ogni topic saranno presenti tutti rawData divisi per owner e saranno pronti per essere consumati secondo regole che ogni owner definirà in seguito.

5.4 Master Data Connectors

Come già detto nel capitolo 3, i Master Data sono dati aggiuntivi agli Event Data, che hanno il compito di fornire un contesto al business process che stiamo trattando. Sono identificati con una chiave GS1 e descrivono quindi un entità nel mondo reale grazie alle 4w.

Questi dati vengono raccolti dagli eventi EPCIS e sono memorizzati in apposite collezioni in MongoDB, tramite il backend principale di iChain. Questi dati sono però anche di fondamentale importanza per la parte di visualizzazione dei grafici, e ciò rende necessario poter creare un canale che consenta a questi dati di essere disponibili anche dal backend del monintoring tramite TimescaleDB. Le collezioni di interesse sono le seguenti:

- *api_items*: Dati del What;
- *Capture_Event_template*: Dati del Why;
- *api_places*: Dati Business Location;
- *api_subplaces*:Dati Read Point;

Per rendere possibile questa operazione è necessario implementare un source connector che riesca a rilevare i cambiamenti all'interno delle collezioni MongoDB e che copi i suddetti in appositi topic Kafka. Come è possibile vedere in figura 5.5 topic sono quattro, ognuno corrispondente ad una delle 4 collezioni Mongo menzionate sopra.

Una volta che i dati sono presenti nei topic, è necessario avere un Sink connector per

rilevare le inserzioni all'interno dei topic e permettere di conseguenza la scrittura all'interno di TimescaleDB. I Sink connector sono degli Agent Faust e sono 4, ognuno quindi che consuma da uno dei 4 topic Kafka. All'interno di questi Agent, nel momento in cui un nuovo dato si inserisce nel topic, si verifica che tipo di operazione è stata effettuata (una insert, un update o una delete) e in base a quella si procede con la relativa operazione su TimescaleDB. Per effettuare queste operazioni nelle table si utilizza la stessa libreria usata per il Source Connector, ovvero asyncpg. All'interno del database Timescale, i dati vengono memorizzati in 4 apposite table, ognuna corrispondente al relativo topic. Di qui sarà quindi possibile poter interrogare le table da parte del Back-end Monitoring. Inoltre in base alle richieste dell'utente, sulla visualizzazione di questi dati, sarà infine possibile unire queste table tramite delle Join.

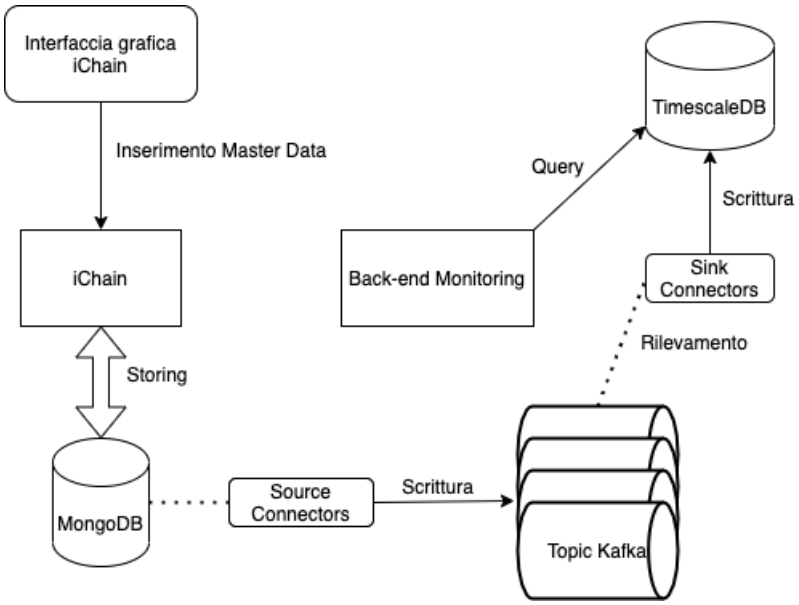

Figura 5.5. Master Data management

5.5 Autenticazione JWT

Un aspetto che non può essere sottovalutato durante la progettazione e l'implementazione di un back-end è sicuramente quello della sicurezza. La sicurezza va fatta man mano che si sviluppa codice e ogni aspetto della progettazione e realizzazione del codice deve essere fatta tenendo sempre questo a mente.

Il processo che maggiormente ha interessato questa prima fase di sviluppo del back-end per il monitoring è quello dell'autenticazione. L'autenticazione è quel processo di verifica di un'identità che richiede l'accesso ad una determinata risorsa informatica.

Proseguendo sulla scia del back-end principale di iChain, l'autenticazione è stata sviluppata secondo lo standard JSON Web Token (JWT) [2]. Lo standard JWT definisce un modo compatto e self contained per trasmettere in maniera sicura informazioni tra più parti sotto forma di oggeetti JSON. Le informazioni saranno fidate in quanto firmate digitalmente.

Il token che viene scambiato è composto da 3 parti:

- *Header*: Consistente di 2 parti, la prima che dice di che tipo di token stiamo parlando e la seconda che indica che tipo di algoritmo è stato usato per la firma;
- *Payload*: Contenente le attestazioni. Le attestazioni contengono delle informazioni sull'entità che ha generato il token. Esse sono di 3 tipi: registrate, pubbliche e private;
- *Signature*: Ovvero la firma digitale, che ha lo scopo di autenticare il token e verificare che esso non sia stato manomesso durante il tragitto nella rete. Quindi questa parte fornisce l'autenticazione e l'integrità;

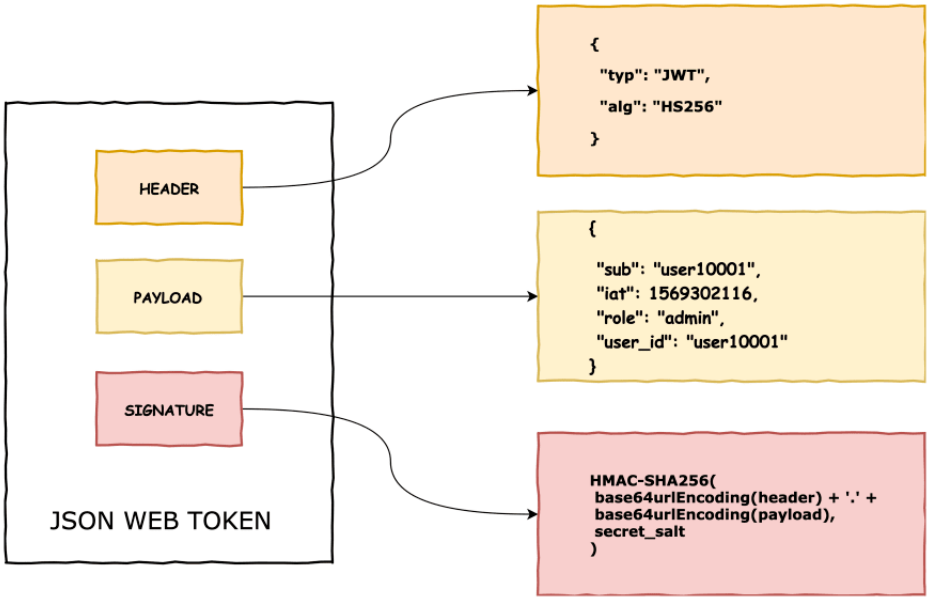

Figura 5.6. Parti del JSON Web Token [12]

Il JWT all'interno del back-end monitoring ha l'obiettivo di autenticare le API scritte grazie a FastAPI. FastAPI fornisce una forte integrazione con questo tipo di standard, fornendo librerie e funzioni per poter scrivere codice ad alto livello.
Durante l'implementazione si è pensato innanzitutto alla realizzazione delle funzioni che servono per la generazione del token. La generazione del token si basa sulle librerie OAuth2PasswordBearer e OAuth2PasswordRequestForm, che generano il token partendo

da un form username-password. Grazie a queste librerie si sono potute creare le funzioni di generazione, verifica e validità del token. Tra queste, di fondamentale importanza è la funzione che recupera le informazioni dello user a partire dal token.

Lo step successivo è stato quello della creazione di un middleware. Per middleware si intende un insieme di programmi informatici che fungono da intermediari tra diverse applicazioni e componenti software [20]. Nel nostro caso si pone come intermediario tra il back-end e le API che vengono esposte all'utilizzatore finale. Per dichiararlo e utilizzarlo basta utilizzare il decorator apposito fornito da FastAPI, definendosi la propria funzione in base alle proprie necessità. Il suo compito qui è quello di verificare nell'header dell'API REST, se la richiesta è quella dell'authorization a un servizio: se è così ne verifica i permessi e salva lo username in una cache, per poter rendere più veloce l'operazione in un secondo momento.
La parte di autenticazione vera e proprio è svolta controllando la validità del token ad ogni richiesta esterna, quindi verificando che l'user o qualunque altra entità che stia comunicando con il server, abbia un token valido e non expired.

Gli utenti o più in generale le entità che sono abilitate alla comunicazione con il server del monitoring sono memorizzate in una collezione Mongo accessibile solo dal back-end principale di iChain, e in versioni future del nostro back-end sarà sicuramente necessario avere a disposizioni un'interfaccia tra i due back-end per interrogare il principale sui permessi di una determinata entità (se l'user è registrato, se può accedere a determinate risorse ecc.)

5.6 Data Processing

Il data proccesing riguarda il processamento dei dati una volta che essi arrivano nei topic delle aziende, quindi dopo i processi descritti nel paragrafo sul Source Connector TimescaleDB-Kafka. In questa fase è necessario definire cosa farne di questi RawData, implementando la vera logica di monitoring. L'obiettivo è quello di avere un sistema in grado di processare dati secondo regole predefinite dalle aziende monitorare i sensori in modo tale da rilevare anomalie e preservare il prodotto finale.

La gestione di queste regole coinvolge anche il back-end principale di iChain, in quanto esse verranno impostate dall'interfaccia grafica principale. Entriamo ora nel dettaglio dei processi che porteranno ad instaurare delle regole di processing. Lo schema completo è rappresentato in figura 5.7

Le API che vengono esposte al client hanno lo scopo di permettere sia la creazione di una configurazione secondo la quale i dati verranno smistati, sia di creare delle regole vere e proprio secondo cui i dati saranno processati. La prima API quindi consente di scegliere in base a quale colonna i dati saranno smistati, se ad esempio sul type, sul device, sulla location ecc. La seconda API invece consente di permettere la vera e propria scelta sul

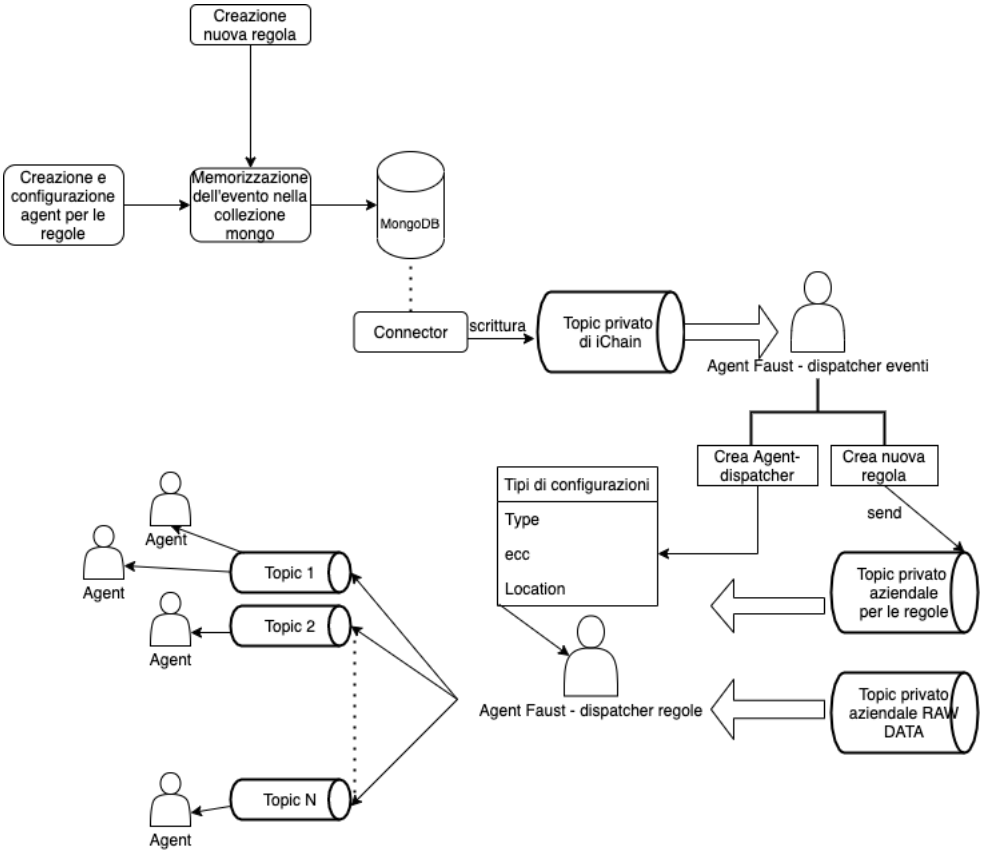

Figura 5.7. Impostazione delle regole di processing

"come" verranno poi processati i dati. La creazione di queste due API si è basata fortemente sulle altre API per la gestione degli altri tipi di regole già presenti in iChain. In particolare si è ricalcata la creazione del modello Pydantic, per la validazione della richiesta, e le interazioni di tipo CRUD con MongoDB. Le regole infatti vengono memorizzate nella collezione Mongo delle rules già presente nel Database, e vengono differenziate dalle altre in base all'attributo *type* del modello pydantic sopra menzionato. In questo modo è possibile conservare tutte le logiche di modellamento e storing già presenti nel back-end di iChain, senza dover ripartire da zero.

La collezione Mongo è collegata ad un Source connector che garantisce l'immediata rilevazione di cambiamenti all'interno di essa. Il connector è implementato secondo la stessa logica dei connector già descritti nel paragrafo sui master data: in uno script bash di iChain vengono impostate delle echo che scriveranno su particolari file di Kafka, per la creazione dei topic indicati e il collegamento tra essi e le rispettive collezioni mongo. Lo script bash viene lanciato al setup. In questo modo è possibile riutilizzare efficacemente

anche il connector ed il topic "rules" già presenti in iChain, nonchè l'agent Faust che consuma da questo topic.

In questa fase si è scelto però di alleggerire il lavoro di questo agent il più possibile, per fare in modo di non comprommetere l'efficienza della gestione rules già presente. L'aggiunta quindi, è stata solamente quella di filtraggio delle rules, in base all'attributo tipo come già detto, e l'invio tramite la cooroutine *send* ad un altro topic, specifico per le rules IoT, da cui consuma un altro Agent che avrà il compito di fare da vero dispatcher per gli eventi "monitoring-IoT".

Prima di andare avanti con la spiegazione del data processing, è opportuno spiegare cosa si intende con "creazione dinamica di un agent", in quanto è di fondamentale importanza nelle restanti parti del processo. Tipicamente un Agent Faust viene dichiarato tramite un apposito decorator che permette di impostare inoltre il topic cui l'agent in questione consumerà. Utilizzando questo pattern però, esiste la necessità di sapere già in fase di sviluppo quanti e quali siano gli agent che la nostra applicazione Faust utilizzerà, per definizione stessa di decorator. Questo rappresenterebbe però un limite evidente se a priori non sapessimo quali e quanti topic la nostra piattaforma di Kafka avrà. La questione diventa quindi trovare un modo per creare dinamicamente un agent, nella stessa maniera con cui si crea dinamicamente un topic. I decorator in Python non sono nient'altro che dei callable che modificano il comportamento di una funzione e la ritornano, la soluzione alla questione menzionata sopra è quindi scendere di livello nel loro utilizzo, ovvero utilizzarli senza la notazione con la @, creando altri callable che verranno castati durante la chiamata dinamica di app.agent.

Tornando al processing, l'Agent creato nell'ultimo step è ancora considerato un Agent interno al sistema di iChain e, in base alla richiesta effettuata, ha come principale compito quello di:

- Creare un Agent Faust in maniera dinamica, in modo tale da essere settato con il tipo di configurazione richiesta;
- Popolare una table Faust con la regola indicata, e da cui un timer provvederà con la scansione. Se sono presenti nuove regole, verrà creato un Agent dinamicamente che provvederà a soddisfare la richiesta;

L'Agent di cui si parla nel primo punto, sarà un dispatcher dei dati raw che lavora in base alla configurazione. Esso consumerà quindi direttamente dal topic che contiene tutti i dati raw di un'azienda, ovvero il topic venutosi a creare dopo i passaggi descritti nel paragrafo sul Source Connector TimescaleDB-Kafka.

L'ultimo passaggio di questo data processing riguarda il vero e proprio monitoraggio su questi dati. Esisteranno tanti topic quante sono le possibilità di un tipo di configurazione, spiegando meglio con un esempio, possiamo dire che se la configurazione è fatta in base al "type", allora esisterà un topic per ogni tipo di sensore: temperatura, umidità ecc. I sopramenzionati topic vengono creati dall'Agent che in figura 5.7 è chiamato Agent Faust

- dispatcher regole, e che è stato creato dinamicamente nel passaggio precedente. Logicamente questi topic, insieme a quello contenente tutti i dati raw, appartengono all'azienda e non sono interni al sistema di iChain: questo garatisce ad un owner di avere pieno controllo sui propri dati raw.

Su ognuno dei topic finali verranno istanziati altri Agent creati dinamicamente ognuno responsabile di una singola regola. Come si può notare anche dalla figura 5.7, possono coesistere più Agent su un singolo topic, in modo tale da poter avere più regole su una singola suddivisione. Spiegando meglio con lo stesso esempio di prima, dividendo per "type" e avendo su uno di questi topic i dati sulla temperatura, possiamo creare la regola:

- Media temperatura compresa in un intervallo;
- Temperatura massima minore di tot;

Come ultimo passaggio del Data Processing c'è la segnalazione di un eventuale malfunzionamento o anomalia riscontrata all'interno dei dati processati. Per intenderci, tornando all'esempio precedente, se la regola è "Temperatura massima" e uno dei record processati ha al suo interno un valore di temperatura maggiore di quello indicato, allora l'Agent che sta gestendo la suddetta regola invierà una segnalazione ad un ultimo topic apposito per gli allarmi. Questo topic verrà consumato da un ultimo agent che provvederà con la segnalazione.

Capitolo 6

Conclusioni

Il back-end realizzato vuole sopperire ai problemi che nascono quando ci si interfaccia con la logistica ed in particolare con la tracciabilità di un prodotto. Al giorno d'oggi, attraversando quella che molti definiscono una nuova rivoluzione industriale, le imprese stanno interfacciandosi con il mondo dell'IoT per migliorare e automatizzare gran parte dei loro processi produttivi, rendendo allo stesso tempo più semplice la gestione della tracciabilità grazie proprio ai sensori sui dispositivi, che rendono il Monitoring più efficace e immediato. I benefici di un sistema del genere si traspongono anche in una migliore Customer Experience, infatti il consumatore finale può reperire tutte le info sul quella determinata merce dimostrando tra l'altro come si siano rispettate tutte le norme vigenti da parte di tutti gli attori della Supply Chain, sia quelli che hanno partecipato in maniera diretta che quelli che hanno partecipato in maniera indiretta. Se si pensa ad esempio al mondo della filiera alimentare, questo rappresenta un'ottima attestazione di qualità per il cliente, che sa cosa porterà nella sua tavola.

Nella realizzazione di un servizio che soddisfi i requisiti preindicati, la prima fonte di attenzione va dedicata ai dati forniti dai sensori che rientrano nella categoria delle serie temporali, e con la quale l'approccio deve essere mirato a non introdurre penalizzazioni dovute all'ingente mole di dati che essi rappresentano. Infatti, esse necessitano di un pattern preciso e dedicato che è quello del Data Streaming. Grazie a questo pattern possiamo processare le serie senza problemi e con una latenza minima.

Tenendo presente le problematiche e gli obiettivi predisposti, il presente lavoro di tesi ha l'obiettivo di estendere le funzionalità di Monitoring della piattaforma di iChain, inserendo al suo interno le funzionalità di ingestion e processing di dati dai dispositivi IoT. Per quanto riguarda la parte di Monitoring, dopo aver analizzato un'adeguata configurazione per la tabella Raw Data in TimescaleDB, è stata prima di tutto implementata un'interfaccia di capturing in grado di validare e modellare i dati raw provenienti dai sensori, in seguito si è implementato un connector in grado di rilevare inserzioni sulla nostra tabella ed infine si è gestita la vera e propria funzione di monitoraggio tramite la definizione di regole per il processing. La sfida principale in questo lavoro è stata quella di mettere insieme una serie di tecnologie e adattarle al nostro scopo finale, rimanendo fedeli ai principi di ottimizzazione, scalabilità e interoperabilità che caratterizzano la piattaforma iChain, senza trascurare le nuove problematiche introdotte con le serie temporali.

Tutti gli aspetti della progettazione e realizzazione di questo servizio sono stati caratterizzati alla conformità con gli standard GS1, ed in particolare con lo standard EPCIS, garantendo di conseguenza un meccanismo sicuro e collaudato di interoperabilità tra imprese della Supply Chain. In particolare si è sviluppato un modello per la nuova versione dello standard EPCIS, la 2.0, che porta con se molte migliorie rispetto alla già efficiente precedente versione, in questo modo si sono gettate le basi per una transizione in tutta iChain verso la nuova rielaborazione dello standard.

In conclusione si può affermare che, con l'implementazione di un back-end in grado di soddisfare le principali funzioni di Monitoring IoT, gli obiettivi del presente lavoro sono stati raggiunti. Per versioni future, sono possibili ancora delle migliorie per un monitoraggio aziendale più completo, con l'aggiunta di altre regole e di altri tipi di allarme, nonché con l'introduzione di nuove API per successive funzioni disponibili ai client, come quella della visualizzazione in tempo reale dei dati acquisiti.

Bibliografia

[1] Apache 2.0 license: *Asyncpg*. – https://magicstack.github.io/asyncpg/current/ Accessed: 31.10.2021.

[2] auth0, MIT license: *JWT*. – https://jwt.io/introduction Accessed: 02.11.2021.

[3] Capodieci, Giuseppe: *Big Data : Dalla Teoria All'Implementazione*. – http://losviluppatore.it/big-data-dalla-teoria-allimplementazione/ Accessed: 25.10.2021.

[4] Cloud, Alibaba: *SQL and TimescaleDB*. – https://medium.datadriveninvestor.com/sql-and-timescaledb-e4676aac38a9 Accessed: 27.10.2021.

[5] FastAPI: *FastAPI*. – https://fastapi.tiangolo.com/ Accessed: 29.10.2021.

[6] GS1: *CBV 2.0*. – Accessed: 17.10.2021

[7] GS1: *Digital link*. – https://gs1it.org/content/public/79/26/7926d04c-d97a-44e8-998d-bfa00b32f928/cs_gs1_digital_link_nuovo_standard.pdf Accessed: 18.10.2021.

[8] GS1: *EPCIS Standard 2.0*. – Accessed: 16.10.2021

[9] Jegatheswaran, Logeesan: *Zookeeper in Kafka*. – https://medium.com/@logeesan/zookeeper-in-kafka-ce31b3dd55b1 Accessed: 25.10.2021.

[10] Kafka, Apache: *Documentation*. – https://kafka.apache.org/introAccessed: 20.10.2021.

[11] Kumar, Narayan: *Kafka Architecture Internal*. – https://mail-narayank.medium.com/kafka-architecture-internal-d0b3334d1df Accessed: 20.10.2021.

[12] Kumar, Suresh: *How JWT (JSON Web Token) authentication works?*. – https://dev.to/sureshdsk/how-jwt-json-web-token-authentication-works-2635 Accessed: 03.11.2021.

[13] Markets, Robinhood: *Faust - Python Stream Processing*. – https://faust.readthedocs.io/en/latest/#faust-python-stream-processing Accessed: 28.10.2021.

[14] Neha Narkhede, Gwen Shapira Todd P.: *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale.* – https://book.huihoo.com/pdf/confluent-kafka-definitive-guide-complete.pdf Accessed: 25.10.2021.

[15] Samuel Colvin, MIT L.: *Pydantic.* – https://pydantic-docs.helpmanual.io/ Accessed: 30.10.2021.

[16] Source, Facebook O.: *RocksDB, Getting started.* – http://rocksdb.org/docs/getting-started.html Accessed: 28.10.2021.

[17] Source, Facebook O.: *RocksDB, Overview.* – https://github.com/facebook/rocksdb/wiki/RocksDB-Overview Accessed: 28.10.2021.

[18] Timescale, Inc: *TimescaleDB Overview.* – https://docs.timescale.com/timescaledb/latest/overview/#timescaledb-overview Accessed: 26.10.2021.

[19] Wikipedia: *GS1.* – https://it.wikipedia.org/wiki/GS1 Accessed: 15.10.2021.

[20] Wikipedia: *Middleware.* – https://it.wikipedia.org/wiki/Middleware Accessed: 04.11.2021.