Università degli Studi di Torino

Dipartimento di informatica

Tesi di Laurea Magistrale in Informatica

Progetto Velocity

Relatore:

Petrone Giovanna

Candidato:

Dentis Lorenzo

Matricola 914833

ANNO ACCADEMICO 2023/2024

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Abstract

Il progetto *Velocity* si propone di sviluppare un sistema di gestione e diffusione dei dati ad eventi,basato su microservizi. L'obiettivo primario è l'estrazione di informazioni da qualsiasi sistema aziendale, inclusi quelli "Legacy" non progettati con un'architettura ad eventi, per renderle disponibili nel minor tempo possibile. Questi dati verranno quindi elaborati, arricchiti e resi visibili a tutte le divisioni aziendali e ai clienti. Un esempio concreto è il processo di tracciamento di un ordine: il sistema pubblicherà ogni evento relativo alla consegna di un prodotto al cliente finale e ai vari passaggi intermedi entro un massimo di 5 minuti dall'evento stesso, garantendo così una completa tracciabilità della merce per il cliente attraverso il portale di tracking dell'ordine.

Indice

Capitolo 1

Tecnologie utilizzate

1.1 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform.[]

`Apache Kafka` è una piattaforma open-source per l'archiviazione e l'analisi di flussi di dati. Si basa sul concetto di *Flusso di eventi (event Stream)*, cioè la pratica di catturare dati in real-time da diverse fonti (databases, sensori, software, ...) sotto forma di **Eventi**.
Un **Evento** è un record all'interno del sistema di qualcosa che si è verificato (il rilevamento di un sensore, un click , una transazione monetaria, etc ...). In Kafka un evento è costituito da una *key*, un valore, un *timestamp* ed eventualmente altri metadati. Un esempio di evento potrebbe essere il seguente:

- **Key**: Alice
- **Value**: "Pagamento di 200€ a Bob"
- **timestamp**: 1706607035

`Kafka` può eseguire 4 operazioni su un **Evento**:

- **Scrittura**: L'evento può essere generato da un *Producer*(1.1.2) che lo pubblica all'interno di un *Topic*.
- **Lettura**: L'evento può essere letto da un *Consumer*(1.1.2) che è iscritto ad un *Topic* e ne riceve gli aggiornamenti.
- **Archiviazione** o **Storage**: Un evento può essere salvato su un *Topic* in maniera sicura e duratura. Differentemente da un `Message Broker`, che offre le stesse funzionalità di lettura e scrittura, i record all'interno di un *Topic* sono permanenti, questo argomento è maggiormente approfondito nella sezione *Topic*(1.1.1)
- **Elaborazione**: Gli **Eventi** possono essere elaborati, sia in gruppo che singolarmente, questa elaborazione può essere effettuata tramite i cosiddetti `Kafka Streams`(1.1.3)

In ultimo `Kafka` è un sistema distribuito, è quindi possibile avere più istanze, dette `Kafka Brokers`, che collaborano in un `Kafka Cluster`. Grazie a questa caratteristica si possono implementare meccanismi di parallelizzazione, high-availability e ridondanza. In particolare su ogni `Broker` sono salvati uno o più *Topic* ed i differenti endpoints (siano essi *Consumers,Producers,Streams o Connectors*) vi dialogano per leggere o scrivere sui *Topic*. Un `Cluster` di esempio è mostrato in figura 1.1

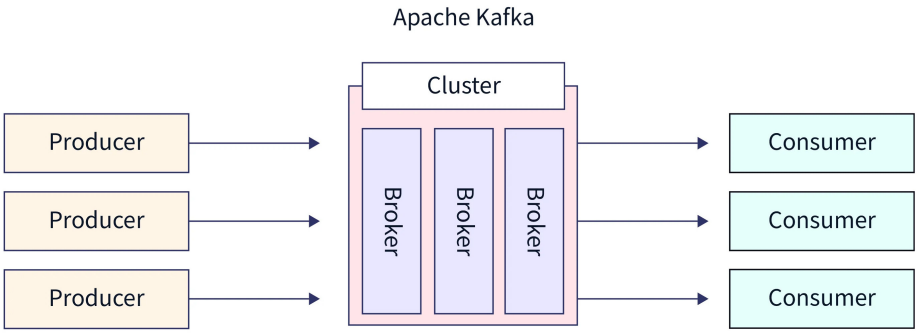

Figura 1.1: Kafka Cluster

1.1.1 Topic

Un *Kafka Topic* è un database ad eventi, al posto di pensare in termini di oggetti, si pensa in termini di eventi. Diversi microservizi possono consumare o pubblicare sullo stesso *Topic*, similmente ad un `Message Broker` infatti i *Topic* sono *multi-producer* e *multi-subscribers*. A differenza di un `Message Broker` però un *Topic* può mantenere dei record in maniera sicura per una durata di tempo indefinita, come se fosse un database.Gli **Eventi** infatti non sono eliminati dopo esser stati letti da un *Consumer*, il tempo di mantenimento di un record può essere configurato in modo da stabilire un equilibrio tra quantità di dati salvati e efficienza delle elaborazioni, dato che ad un numero maggiore di record corrisponde un tempo di elaborazione maggiore.

I *Topic* sono partizionati per permettere high-availability, fault-tollerance e soprattutto consentire la lettura/scrittura in parallelo. Infatti ogni *Topic* è distribuito tra vari *buckets*, che si trovano nei `Kafka Brokers`. Eventi definita dalla stessa *Key* sono scritti nella stessa partizione e *Kafka* garantisce che qualsiasi *Consumer* iscritto a tale partizione leggerà gli eventi nello stesso ordine in cui sono stati scritti. Come citato prima il partizionamento permette anche la scrittura in parallelo, infatti se la partizione su cui due *Producer* scrivono è differente e possibile effettuare l'operazione senza doversi preoccupare dei problemi generati dalla scrittura concorrente, anche se il *Topic* è il medesimo. Un esempio di partizionamento è mostrato in figura 1.2

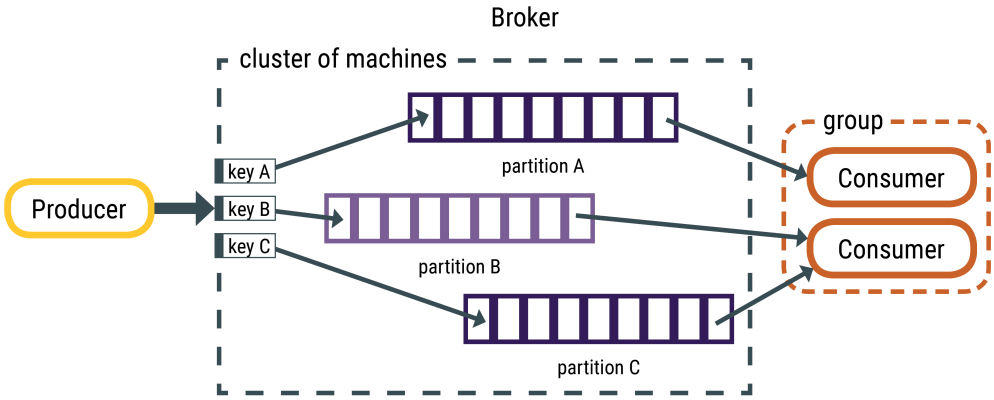

Figura 1.2: Partizionamento Kafka Topic

1.1.2 Clients

I `Consumers` ed i `Producers`, insieme ai *Topics*, sono gli elementi alla base del funzionamento di *Kafka*. Il *Cluster Kafka* composto dai vari *Brokers* svolge il ruolo di **Server**, mentre i `Consumers` ed i `Producers` fungono da **Clients**, collegandosi al cluster e interagendo con i dati presenti sui *Topics*. I *Clients* sono i componenti che si occupano di implementare la logica di business e sono quindi interamente scritti dallo sviluppatore che sfruttera le due API messe a disposizione da Kakfa, *Consumer API* e *Producer API*

1.1.3 Streams

`Kafka Streams` è una API per processare eventi su un `Topic Kafka` (filtrare, trasformare, aggregare, ...). Ad esempio se volessi sapere quanti ordini di trasporto sono stati spediti oggi, potrei fare un filtro per data e poi un count, quello che il `Kafka Stream` fornirà in output sarà un altro flusso di dati filtrato, che potrò salvare su un nuovo `Topic` o in un database.
Nascono con l'intenzione di "astrarre" tutte le operazioni di basso livello quali la lettura o la scrittura su un *Topic*, permettendo allo sviluppatore di preoccuparsi solamente di come i dati devono essere modificati, senza dover scrivere codice per ottenerli o ripublicarli. In pratica qualsiasi operazione implementabile tramite `Kafka Streams` sarebbe allo stesso modo implementabile da un microservizio che legge da un *Topic*, elabora i dati e li riscrive su un *Topic*(lo stesso o un altro), ma grazie agli `Streams` si possono delegare le operazioni di collegamento con il `Kakfka Cluster` e concentrarsi solamente sull'elaborazione dei dati. L'utilizzo dei `Kafka Streams` ha i seguenti vantaggi:

- Efficienza: Il tipo di computazione è per-record, cioè ogni dato pubblicato sul *Topic* a cui lo `Stream` è collegato viene subito processato. Non c'è bisogno di effettuare "batching", cioè richiedere i dati ad intervalli di tempo regolari ed elaborare solo i dati giunti in tale intervallo. Il sistema può lavorare quasi in tempo reale.

- Scalabilità: Gli `Stream` sono scalabili e fault-tollerant. Essendo *Kafka* pensato per essere un sistema distribuito anche gli `Streams` sono pensati per essere

scalati e distribuiti, se si creano diverse istanze dello stesso `Stream` queste collaboreranno automaticamente suddividendosi il carico computazionale.

- Riuso del codice: si utilizza una chiamata all API al posto di riscrivere lo stesso codice per differenti microservizi.

1.1.4 Connector

I `Connectors` sono particolari tipi di `Consumers/Producers`, il cui scopo è mettere in comunicazione *Kafka* con altri sistemi. I `Connectors` producono flussi di eventi partendo da dati ricevuti da un altro sistema (*Source Connector*), oppure consumano da un topic e inviano i dati letti ad una applicazione esterna(*Sink Connector*). Per esempio un `Connector` ad un database relazionale potrebbe catturare tutte le operazioni effettuate su una tabella e generare un flusso di eventi in cui ogni evento corrisponde ad un cambiamento.
Similmente ai *Kafka Streams* (1.1.3) i principali vantaggi di utilizzare un `Connector` piuttosto che scrivere da se il codice per svolgere lo stesso compito sono **efficienza**, **scalabilità** e **riuso del codice**. Inoltre sono presenti diversi repository online dove trovare `Connector` già pronti, sviluppati dalla community o dagli sviluppatori delle applicazioni esterne (sqlserver, JDBC, Amazon S3), uno dei più diffusi è il `confluent-hub` https://www.confluent.io/product/connectors/

deve diventare un link o essere postato in bibliografia

1.2 Debezium

Debezium is an open source distributed platform for change data capture. Start it up, point it at your databases, and your apps can start responding to all of the inserts, updates, and deletes that other apps commit to your databases.[]

`Debezium` è una piattaforma distribuita open source per la cattura dei dati di modifica, permette di catturare le operazioni di modifica effettuate su un database (*insert, update e delete*) e di trasformarle in eventi. Nativamente tutti i più comuni database sono supportati, tra cui *MySQL, PostgreSQL, MongoDB, SQL Server, Oracle, ...*

`Debezium` è costruito sulla base di *Apache Kafka*, di conseguenza è facilmente integrabile con esso. Ci sono infatti due modi per utilizzare questa piattaforma: come un sistema distriubuito a se stante oppure tramite un `Kafka Connector`(1.1.4). Il primo modo presenta tutti i vantaggi di un sistema distribuito, come la scalabilità e la fault-tollerance, ma ha un costo in termini di risorse, di configurazione e di gestione. In questo modo è inoltre possibile consumare gli eventi prodotti tramite un qualsiasi sistema esterno, non si è obbligati ad utilizzare *Kafka*.
Se invece si ha già una infrastruttura già basata su *Kafka* è molto più conveniente utilizzare `Debezium` come `Connector`, con un considerevole risparmio di risorse e di tempo.

1.3 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale.[]

Apache Flink è sia un framework che un motore di elaborazione distribuito per la computazione di flussi di dati (`streams`), *bounded* e *unbounded*. Può essere utilizzato per elaborare dati in tempo reale, ma anche per elaborare grandi quantità di dati in batch. Fornisce sia una API per la creazione di applicazioni di elaborazione di dati, sia un *runtime* environment distribuito per eseguirle, per questo motivo si può considerare sia un *framework* che un *motore di esecuzione*. Come anticipiato i tipi di dato trattato sono sempre `streams` che possono essere:

- **Unbounded**: un flusso di dati che non ha un inizio o una fine, come ad esempio un flusso di eventi generati da un sensore.

- **Bounded**: un flusso di dati con un inizio e una fine, come ad esempio un file o una tabella di un database.

Sui flussi *Bounded* possono essere eseguite tutte le operazioni eseguibili sui flussi *Unbounded*, ma non viceversa. Per fare ciò bisogna ricorrere a delle operazioni di *windowing*, cioè dividere il flusso in finestre (temporali o di conteggio [1]) e poi eseguire le operazioni su queste finestre.

1.3.1 Statefull stream processing

Flink può svolgere operazioni che richiedono il mantenimento di uno stato, cioè un insieme di informazioni riguardanti gli eventi passati. Semplici esempi di elaborazioni che richiedono uno stato sono: la ricerca di un pattern, il calcolo di una media (mobile se si parl di *Stream Unbounded*), il calcolo di una somma, etc ...*Flink* sfrutta lo stato anche per garantire la *fault-tollerance*, cioè la capacità di ripristinare il sistema in caso di guasto.

Il metodo più comune con cui una applicazione *Flink* sfrutta lo *Statefull processing* è tramite l'uso di *Keyed Stream*. Operando su un `DataStream` può essere effettuata una operazione di `keyBy(key)`, dove *key* è un qualsiasi oggetto java (POJO) che implementa il metodo `hashCode()`. Tale operazione permette di raggruppare gli eventi in base ad una chiave, in modo che tutti gli eventi con la stessa chiave appartengano alla stessa partizione logica. Si ottiene quindi un `KeyedStream` su cui è possibile eseguire operazioni di *statefull processing*, il seguente codice di esempio mostra come calcolare la somma di un `KeyedStream` di ogetti composti da una chiave (*f0*) e un valore(*f1*):

[1] Per *finestra di conteggio* si intende una finestra che contiene un numero fisso di elementi, ad esempio una finestra di 100 elementi. In inglese si parla di *count window*. maggiori informazioni nella sezione *Windowing* 1.3.2

```
...

dataStream.keyBy(value -> value.f0)
.reduce((accumulator, value2) -> {
    accumulator.f1 += value2.f1;
    return accumulator;
    });

...
```

Listing 1: Esempio di operazione statefull su un KeyedStream

Oltre ad essere necessario per mantenere uno stato il partizionamento tramite chiave permette anche di parallelizzare le operazioni, dato che ci assicura non ci saranno conflitti tra le operazioni eseguite su partizioni diverse. Nell'esempio precedente (listing 1) la somma potrebbe venire calcolata in parallelo per ogni chiave, dato che lo stato mantenuto dal sistema, che corrispone semplicemente alla variabile `accumulator`, non viene mai acceduto durante la computazione di un dato avente un'altra chiave, posto naturalmente che si abbia a disposizione sufficienti risorse computazionali. Il discorso di come *Flink* gestisca le risorse è approfondito nella sezione 1.3.3.1.

parlare del ch pointing?

1.3.2 Windowing

Alcune elaborazioni richiedono di operare su un sottoinsieme di dati, soprattutto quando si tratta di flussi di dati *Unbounded*. Ad esempio se si volesse calcolare la media di un flusso di dati, sarebbe necessario calcolare la media solo sui dati arrivati in uno specifico intervallo, non è possibile calcolare la media su tutti i dati del flusso dato che il flusso non ha un inizio o una fine. Il *windowing* è una tecnica di elaborazione di flussi di dati che permette di dividere un flusso in finestre, su cui poi eseguire operazioni di aggregazione o riduzione. Tornando all'esempio della media, si potrebbe dividere il flusso in finestre temporali di 1 minuto e poi calcolare la media su ciascuna finestra.

Le finestre possono essere *temporali* o *di conteggio*, le prime sono divise in base al tempo, ad esempio una finestra di 1 minuto, le seconde in base al numero di elementi, ad esempio una finestra di 100 elementi. Rispettando questa distinzione si possono avere altri 3 tipi di finestre:

- **Tumbling Window**: una finestra temporale o di conteggio che non si sovrappone ad altre finestre, ad esempio una finestra di 1 minuto.
- **Sliding Window**: una finestra temporale che si sovrappone ad altre finestre, ad esempio una finestra di 1 minuto che si sposta di 30 secondi ad ogni nuovo evento.
- **Session Window**: una finestra temporale che si basa su un intervallo di tempo inattivo, ad esempio una finestra di 1 minuto che si chiude quando non ci sono eventi per 5 secondi. Come le sliding window anche le session window si possono sovrapporre.

Un esempio dei vari tipi di finestra è mostrato in figura 1.3

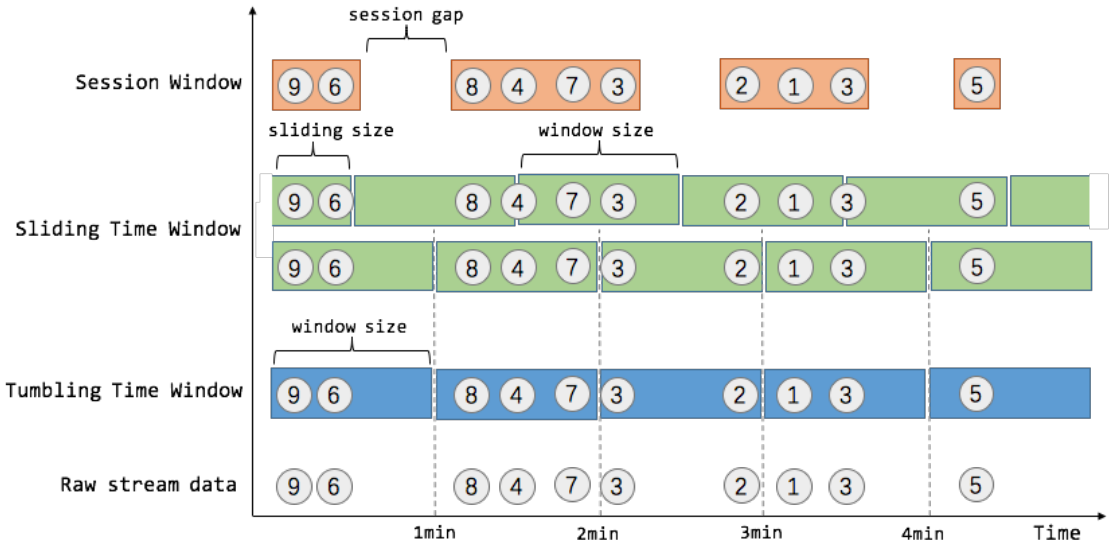

Figura 1.3: Tipi di finestra

1.3.2.1 Wartermarks

Legato al concetto di finestra temporale è il concetto di *Watermark*, un *Watermark* è un marcatore temporale che indica il punto in cui non ci saranno più eventi precedenti. Nascono dalla necessità di misurare il tempo quando si lavora con *finestre temporali* dato che l'operatore *finestra* ha bisogno di essere notificato quando è passato più tempo di quanto specificato ed è ora che la finestra si chiuda (non accetti più eventi) ed inizi la computazione.

I *watermarks* vengono gestiti da *Flink* come se fossero normali eventi, sono inseriti all'interno del flusso di eventi e contenfono un timestamp t. quando il *Watermark* con timestamp t raggiunge l'operatore finestra l'assunzione implicita è che non ci dovrebbero più arrivare elementi con timestamp t' tali per cui $t' \leq t$. Cioè eventi verificatisi "prima" del tempo t del *watermark*. Questo strumento non sembra molto utile finchè si considerano flussi di dati ordinati ma diventa fondamentale se il flusso su cui stiamo operando presenta eventi non ordinati (rispetto al timestamp). Situazione che si verifica di frequente quando si ha a che fare con sistemi complessi e distribuiti dove non è garantito che gli eventi giungano al sistema di elaborazione mantenendo l'ordine con cui sono avvenuti. Un esempio di flusso con eventi non ordinati ed il relativo uso dei *watermarks* è mostrato nella figura 1.4

Nel caso di eventi fuori ordine si possono impostare le *finestre* in modo che permettano un "ritardo" nell'arrivo di alcuni eventi anche dopo che il *watermark* ha raggiunto la finestra. Gli eventi in ritardo sono formalmente definiti come gli eventi aventi $t' \leq t$ ma che giungono alla *finestra* dopo l'arrivo del *watermark* e possono essere gestiti in diversi modi. Tra i più comuni abbiamo la possibilità di reindirizzarli in un apposito flusso (*side output*) oppure di rielaborare tutti i dati nella *finestra* scartando la precendente computazione (*firing*)

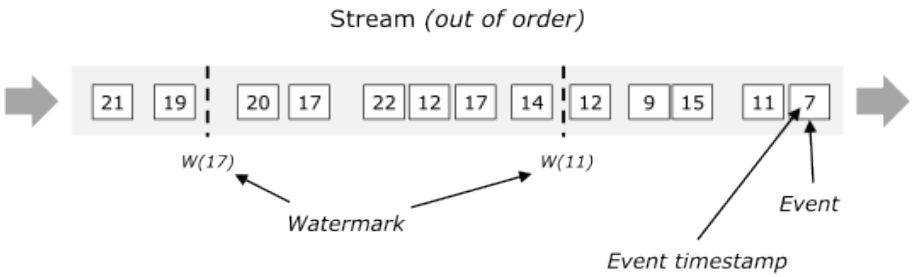

Figura 1.4: Watermarks in out-of-order stream

1.3.3 Flink Cluster

Un *Flink Cluster* è penasto per essere eseguito in un ambiente distribuito, come ad esempio un *Hadoop YARN* o *Kubernetes*, ma non è limitato a ciò, può essere eseguito anche in un ambiente *standalone* oppure si può sfruttare solo la API di Flink senza dover necessariamente eseguire un *Cluster*.

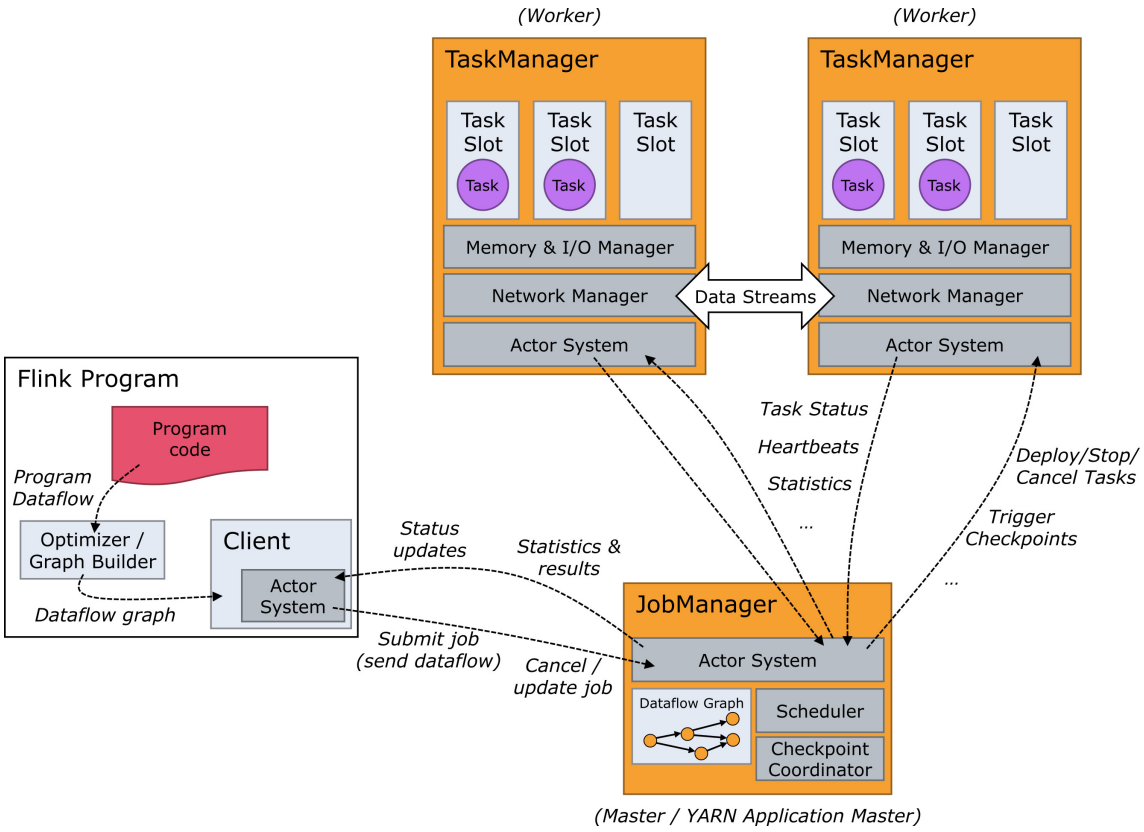

Figura 1.5: Flink Cluster

Il *runtime environment* di Flink è composto da due tipi di processi: i JobManager ed i TaskManager.

- **JobManager**: è il master del *Cluster*, si occupa di coordinare il lavoro dei *TaskManager* decidendo quando eseguire un Task, gestire gli errori in fase di esecuzione ed effettuare checkpointing [2]. Le principali responsabilità del *JobManager* sono due: la gestione delle risorse e la gestione dei job.

[2]Il meccanismo di checkpoint è una strategia di *fault tolerance* basata sul mantenere degli snapshots di vari componenti del sistema per poterlo ripristinare in caso di guasto

1. **Gestione delle risorse**: il *JobManager* gestisce i **Task Slots** (sezione 1.3.3.1) dei *TaskManager*, cioè le risorse computazionali. Se l'ambiente di esecuzione è distribuito (come ad esempio *YARN* o *Kubernetes*) il *JobManager* può richiedere di avviare nuove istanze di *TaskManager* in base alle necessità.
2. **Gestione dei job**: il *JobManager* si occupa di ricevere i job da eseguire, di distribuirli tra i *TaskManager* e di monitorarne l'esecuzione. Inoltre fornisce una REST API ed una interfaccia web per monitorare lo stato del cluster e per ricevere nuovi jobs.

C'è sempre almeno un *JobManager*. Una configurazione *Hig-Availability* potrebbe avere più *JobManager*, di cui uno è sempre il leader e gli altri sono in standby.

- **TaskManager**: sono i worker del *Cluster*, si occupano di eseguire i Task e di ricevere e inviare i dati. La più piccola unità di esecuzione in un TaskManager è un **Task Slots** (sezione 1.3.3.1). Il numero di slot di Task in un TaskManager indica il numero di Task eseguibili in parallelo.

1.3.3.1 Task Slots

Per comprendere il concetto di *Task Slot* è necessario comprendere come viene gestito il *parallelismo* in `Flink`. Un programma scritto tramite l'API di `Flink` non viene eseguito sequenzialmente, ma viene diviso in sottoprocessi che possono essere eseguiti in parallelo. Ad esempio un semplice programma che riceve dei dati, li filtra e li salva su un database potrebbe essere diviso in tre sottoprocessi: ricezione, filtro e scrittura. Questi tre sottoprocessi sono chiamati *Task* e possono essere, parzialmente, eseguiti in parallelo.
Inoltre, come anticipato nella sezione 1.3.1, una operazione di `KeyBy()` suddivide uno stream in partizioni di dati indipedenti tra loro, quindi tutte le operazioni su questi dati possono essere eseguite in parallelo. Potenzialmente potremmo eseguire la computazione di questi dati dedicando un *Task* ad ogni partizione, cioè per ogni *key* presente nello stream, ottenendo così il massimo parallelismo possibile. Qualora non si abbia a disposizione sufficienti risorse computazionali per dedicare ad ogni partizione un *Task* il `Flink runtime environment` automaticamente aggregherà più partizioni in un cosiddetto **Key Group** che verrà trattato come una qualsiasi altra partizione.

Ogni `TaskManager` è sostanzialmente un processo che esegue la JVM ed il codice, quindi può eseguire uno o più sottoprocessi in diversi *threads*. Ogni `Task Slot` rappresenta un sottoinsieme di risorse computazionali di un `TaskManager`. Ad esempio un `TaskManager` con 3 `Task Slot` dedicherà 1/3 delle sue risorse a ciascun `Task Slot`. Più `Task Slot` si hanno a disposizione, più `Task` si possono eseguire in parallelo, ma anche meno risorse si avranno a disposizione per ciascun `Task`. Inoltre se due `Task` operano sugli stessi dati è possibile effettuare *slot sharing*, cioè permettere a due `Task` di condividere un `Task Slot`, limitando il parallelismo ma riducendo il consumo di memoria. Come si può vedere nell'immagine 1.6 dove i `Task` *Source* e *Map* condividono un `Task Slot`.

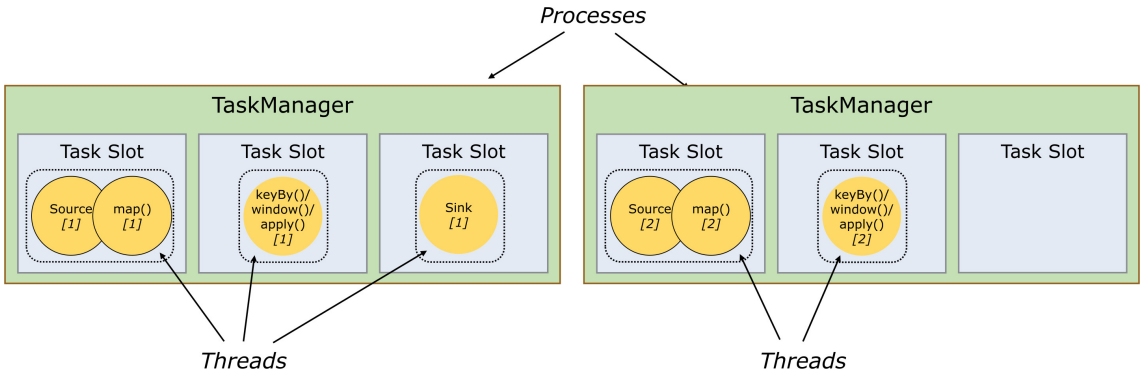

Figura 1.6: Task Slots sharing

1.3.4 Apache Avro

`Apache Avro` è un sistema di serializzazione di dati, simile a `JSON` o `XML`, ma con un formato binario. La sua caratteristica principale è la possibilità di definire uno *schema* per i dati, che viene poi utilizzato per la serializzazione e deserializzazione. Questo *schema* può accompagnare i dati serializzati, permettendo al sistema una ricostruzione dei dati dinamica anche senza conoscere a priori il tipo di dato, oppure può essere salvato in un file separato e condiviso tra i vari sistemi che devono scambiarsi i dati.

Un oggetto `Avro` è quindi composto da due parti:

- Un header dove si possono trovare diversi metadati e lo *schema*, definito in JSON.
- Un body, che contiene i dati serializzati. Questi possono essere codificati in maniera efficiente tramite una rappresentazione binaria oppure in maniera leggibile tramite JSON.

La presenza o meno dello *schema* all'interno dell header dipende da come si decide di utilizzare `Avro`. Se ad esempio ci fosse la necessità di scrivere un file `Avro` su disco, sarebbe molto conveniente includere lo *schema* al suo interno, in modo che chiunque lo legga possa ricostruire i dati.
Se invece si usa *Avro* in RPC (*Remote Procedure Call*) non conviene includere lo *schema* in ogni messaggio, dato che sarebbe ridondante e aumenterebbe inutilmente la dimensione dei messaggi. In questo caso il client ed il server si scambierebbero lo *schema* una sola volta, durante la fase di handshake.

1.3.5 Hybernate

Capitolo 2

Architettura del sistema

Il *Progetto Velocity* è un sistema di Track&Trace il cui scopo è il monitoraggio in near real time della logistica. Al momento il sistema monolitico legacy gestisce tutto, il nuovo sistema, *Velocity*, lo soppianterà gradualmente, seguendo un approcio *brownfield*. In questo momento si stanno sviluppando i nuovi servizi e li si sta integrando con il vecchio sistema, i nuovi clienti vengono direttamente gestiti con le componenti funzionanti del nuovo sistema, i clienti che invece venivano gestiti con il sistema legacy stanno venendo gradualmente trasferiti.

2.1 Sistemi coinvolti

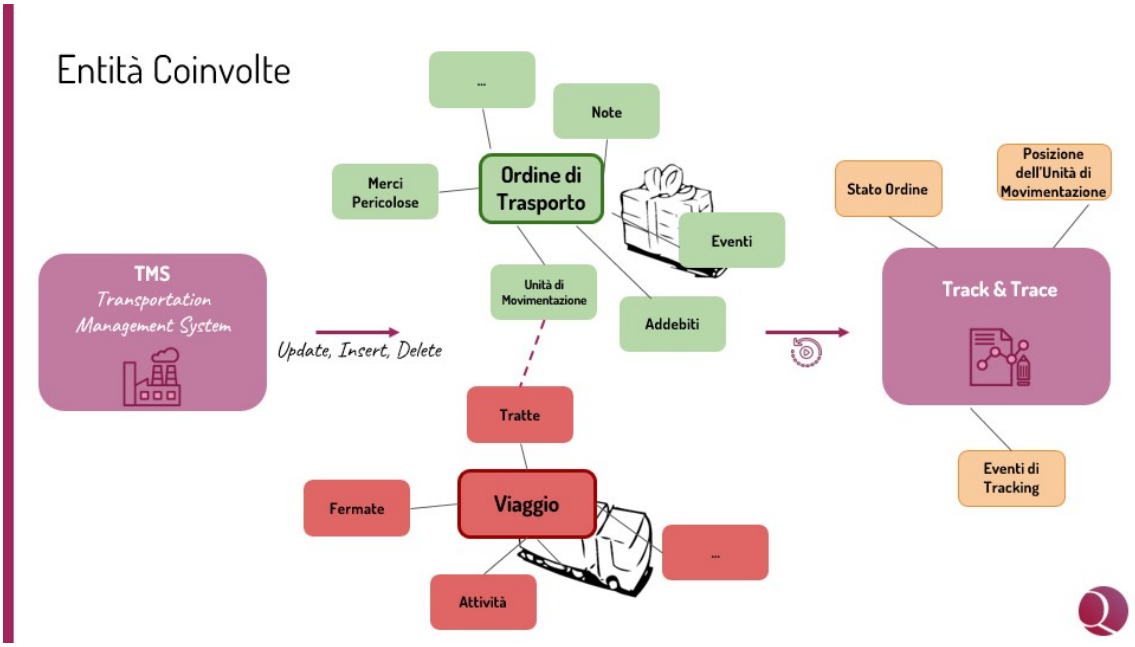

Figura 2.1: Entità coinvolte

La sorgente dei dati è il *Transport Management System (TMS)* un software esterno che effettua modifiche a diverse entità. Con il termine **Entità** si intende qualsiasi caratteristica che definisce un oggetto o situazione reale, ad esempio un *viaggio* potrebbe avere le seguenti entità: *origine,destinazione, durata, etc ...*
Nel sistema le entità sono raggruppabili in diversi domini, i cui 3 principali sono:

- Ritiro
- Ordine di trasporto (Spedizione)
- Viaggio

In figura 2.1 sono presentati due domini d'esempio, *Viaggio* e *Ordine di Trasporto*. I vari domini non sono isolati l'uno dall'alto, bensì la modifica di una entità potrebbe implicare la modifica di un'altra entità. In figura 2.1 ad esempio modificando una *unità di movimentazione* verrebbe conseguentemente modificata una *tratta*.

Lo scopo del sistema di **T&T** è proprio quello di tenere traccia di tutti i cambiamenti subiti dalle varie entità (effettuati dal **TMS**). Ciò avviene tramite `Debezium` (sezione 1.2) che monitora i database del **TMS** generando eventi di dominio, che verranno poi processati da altri microservizi. Il **T&T** si occupa anche di fornire ai clienti informazioni sullo stato e sulla storia di diversi oggetti, composti dalle entità. Ad esempio potrebbe essere fornito l'oggetto *Carico* che descrive lo spostamento di un mezzo e tutte le consegne effettuate, quindi costituito da *Tratta e Fermate* ma anche dalle informazioni riguardo alle merci che trasporta, cioè *Note, Unità di movimentazione, etc ...*

2.2 Track and Trace legacy

Sistema basato su Batch, a regolari intervalli di tempo il sistema va a vedere il nuovo stato delle entità ed aggiorna un suo database interno di oggetti composti.

re più info

2.3 Progetto Velocity

Sistema *Event Driven* basato su `Kafka Streams`. (sezione 2.3.1.1) Quando una entità cambia stato la modifica viene registrata su un `Kafka Topic` e, successivamente, gli eventi sui Topic vengono analizzati o filtrati tramite `Kafka Stream`. Possiamo distinguere 3 tipologie di `Kafka Topic`:

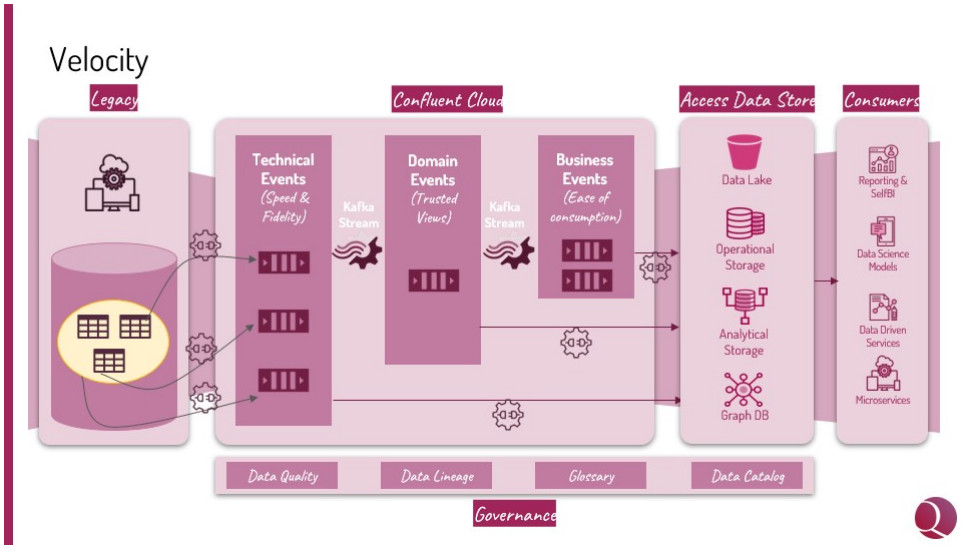

Figura 2.2: Tipologie di Kafka Topics

- **Technical Events**: Contengono gli eventi generati dal **TMS**, sono eventi molto simili a dei log di un database (essendo generati tramite `Debezium` sono sostanzialmente una collezione di operazioni SQL) e spesso sono ridondanti, è infatti comune che il sistema effettui operazioni poco efficienti. Ad esempio qualora io avessi un ordine con una nota e la volessi modificare il TMS potrebbe svolgere la richiesta segnalando una operazione di `DELETE` ed una di `INSERT` piuttosto che effettuare una semplice `UPDATE`.
  É presente un `Topic` di tipo *Technical Events* per ogni tabella del database originale (quindi un topic ogni dominio).

- **Domain Events**: Questi Topic contengono gli eventi filtrati dai *Technical Events* tramite i `Kafka Streams`, non sono più simili a dei log di un DB (già solo la loro struttura è JSON, non SQL) e rappresentano come è fatto un oggetto (ordine di trasporto, viaggio, ...). Tutti i potenziali eventi ridondanti sono stati filtrati dallo *Stream*, non vi è più quindi il problema degli eventi ridondanti.

- **Business Events**: `Topic` opzionali, contengono degli eventi strutturati come i consumatori si aspettano (Ease of consumption). Sono pensati per fornire una vista specifica per un particolare consumer.

2.3.1 Componenti

2.3.1.1 Kafka Streams

`Kafka Streams` è una API per processare eventi su un `Topic Kafka` (filtrare, trasformare, aggregare, ...), questo tema viene approfondito nella sezione 1.1.3

Gli `Stream` che collegano i `Topic` di tipo *Domain Events* a quelli di tipo *Domain Business Event* sono molto dipendenti dalle necessità del consumatore che poi li leggerà quindi non seguono una struttura fissa. Invece gli `Stream` che leggono dai *Technical Events Topic* seguono una struttura precisa e svolgono operazioni suddivisibili in 3 fasi:

1. **Fase di Casting**.
   In questa fase avviene la ricostruzione dell'evento basandosi sui log generati da `Debezium` che sta osservando il **TMS**.
   `Debezium` si occupa di rilevare ogni cambiamento e pubblica l'evento su diversi topic kafka, uno per tabella (quindi uno per ogni dominio).

2. **Fase di Filtro**
   Successivamente gli eventi ridondanti devono essere eliminati. Tutti gli eventi relativi ad una transazione vengono accorpati e viene generato un unico evento risultante, che non riporta gli eventi intermedi.

3. **Fase di Mapping**
   Il nuovo evento viene quindi trasformato in un *Domain Event* ed inserito sul relativo Topic.

2.3.1.2 MicroBatch

Questo microservizio si occupa di "ricostruire" una entità a partire da tutti gli eventi che la riguardano. Non legge direttamente dal Topic *Domain Events* quindi non è

un *consumer Kafka* bensì legge gli eventi di dominio da elaborare da un database SQL chiamato `Fast Storage` che viene continuamente aggiornato da un connettore JDBC. Gli eventi che riceve in input sono quindi dei *Domain Events*, già filtrati dai relativi `Kakfa Stream`.
Dopo la "ricostruzione" l'oggetto viene riscritto nel Fast Storage, eliminando da esso gli eventi che lo riguardavano e che non sono più necessari. Il microservizio `MicroBatch` è scritto usando *Spring Batch* e lo scheduler su cui si appoggia per eseguire i Job è *Quartz*.

Il primo passo, svolto ogni 5 secondi, è un partizionamento. Una classe `Spring Batch` chiamata *Partitioner* divide gli eventi di dominio in *chunks*, in modo da poterli processare in parallelo. Il numero di *chunks* è liberamente configurabile, ma il partizionatore è scritto in modo da raggruppare gli eventi con la stessa chiave di dominio (cioè relativi alla stessa Transazione) nella stessa partizione. Questo non garantisce però che all'interno di un *chunk* ci siano solo eventi con la stessa chiave di dominio. A questo punto vengono eseguiti i vari *Jobs*, uno per ogni *chunk*, la cui esecuzione si può suddividere in 3 passi.

1. **Reading**: Durante la fase di *Reading* vengono recuperati dal Fast Storage tutti gli Eventi di Dominio che sono stati assegnati dal Partizionatore a quello specifico *chunk*.

2. **Processing**: Fase in cui si trasformano gli Eventi di Dominio recuperati durante la fase di *Reading* in una serie di record pronti alla scrittura, ovvero in una serie di oggetti di tipo `Entity`. Le *Entità* andranno quindi a comporre degli oggetti di vario tipo, infatti `MicroBatch` non si occupa di tenere aggiornata una sola tabella, bensì diverse tabelle sullo stesso database. Quindi partendo dagli stessi Eventi di Dominio verranno generati diversi record (diverse `Entity`) che verranno scritti su diverse tabelle.

o avro acces-
DB potrò in-
questi esempi

3. **Writing**:Fase finale di scrittura sul Fast Storage. É una scrittura transazionale, quindi deve rispettare le proprietà `ACID`, requisito di cui si occupa il *Job*. Inoltre il *Job* si occupa di verificare per ciascuna tabella se i dati che ha generato devono essere inseriti o solamente aggiornati.

2.3.1.3 Event Engine

Similmente a `Micro Batch` (sezione 2.3.1.2) l'*Event Engine* si occupa di "costruire" degli oggetti di business partendo dal *Fast Storage*, questi oggetti sono pensati per la *ease of consumption* di eventuali client.
In altre parole si occupa di osservare i cambiamenti di stato dei diversi eventi di dominio (segnalati dall **SGA** che monitora il **TMS**) e generare una serie di eventi di business associati (es: "spedizione partita", "ritiro fallito", "arrivo stimato", ...)

Rispetto al caso `Micro Batch` (2.3.1.2)la fase di *Reading* ritorna solo un record per ogni chiave di dominio, quindi ad ogni *chunk* corrisponde una e solo una chiave di dominio. Invece le tre altre due fasi (*Processing e Writing*) sono sostanzialmente identiche, con la differenza che la fase di Processing non va a generare un oggetto, bensì calcola una serie di metriche come "orario di partenza", "Tragitto", "Stato dell'ordine", etc ....

Capitolo 3

Microservizio EventExport

3.1 Panoramica del microservizio

Lo scopo del microservizio *EventExport* è quello di elaborare dei **Business Event**, che arrivano dal microservizio *EventEngine* (sezione 2.3.1.3) e di comunicarli al cliente.

EventExport è configurabile in maniera differente per ogni cliente, di modo che ad un cliente vengano inviati solo alcuni tipi di eventi. Ad esempio, un cliente potrebbe essere interessato solo agli eventi di creazione di un ordine, mentre un altro potrebbe voler ricevere tutti gli aggiornamenti riguardo alla posizione dell'ordine da lui effettuato. Questa configurazione è gestita tramite una tabella *TradingPartnerEventLookup* che contiene le regole di filtro per ogni cliente.

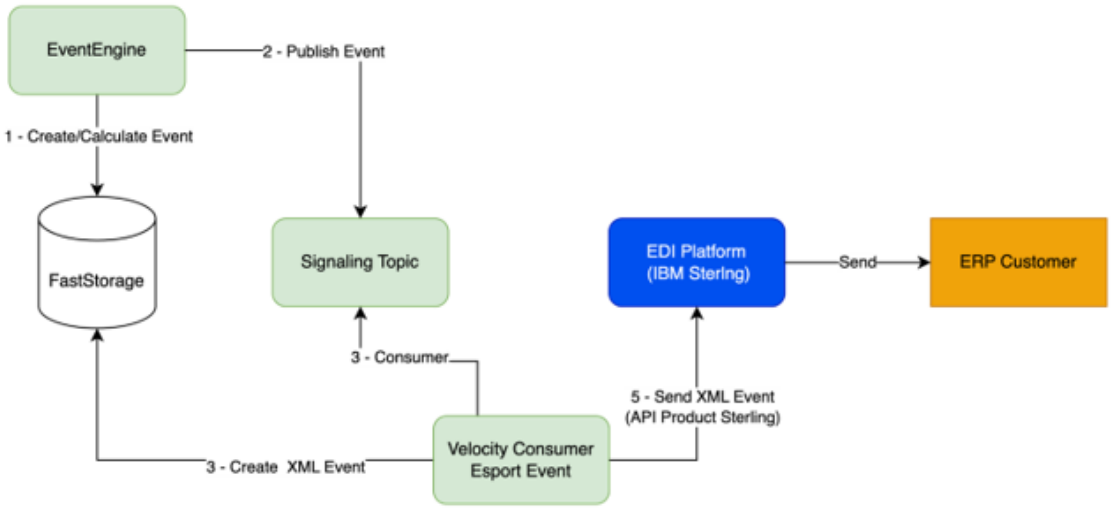

Figura 3.1: architettura con il microservizio *EventExport*

Il microservizio compie le seguenti operazioni:

1. Consumare il *Signaling Topic* (topic Kafka) applicando un filtro basato su una tabella di configurazione.

2. Controllare se l'evento deve essere inviato. Non è necessario inviare eventi già inviati a meno di modifiche su campi significati (anche questi definiti in base alla configurazione).
3. Creare un file XML con i dati richiesti dal cliente. I dati sono definiti in base a dei template che vengono compilati in base alla configurazione.
4. Inviare il file XML ad un altro servizio che si occuperà di inviare al cliente una mail.
5. Loggare l'evento, compreso di XML, in una tabella (*TransportOrdersExportEvents*) per tenere traccia degli eventi inviati.

3.2 Lettura del Signaling Topic

La lettura del *Signaling Topic* avviene tramite un *Consumer* Kafka. Flink mette a disposizione un *Kafka Consumer* che permette agevolmente di leggere i messaggi da un topic Kafka: `org.apache.flink.connector.kafka.source.KafkaSource` Il *Broker* Kafka su cui vive il *Signaling Topic* è configurato per usare un sistema di autenticazione basato su *SASL/SSL*, inoltre gli eventi sul topic sono serializzati tramite *Apache Avro* (sezione 1.3.4) quindi è necessario configurare il *Consumer Kafka* di conseguenza.

```
public KafkaSources() {
  Properties properties = new Properties();
    try {
      Reader reader = new FileReader(new File("kafka.properties"));
      properties.load(reader);
      reader.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
      exit(1);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }

    //Creazione del KafkaSource per il topic "SignalingTopics"
    EventSource = KafkaSource.<SignalingTopicRecord>builder()
      .setProperties(properties)
      //Properties per la connessione a Kafka (server, parametri SASL, etc..)
      .setTopics("SignalingTopics")
      .setStartingOffsets(OffsetsInitializer.earliest())
      .setValueOnlyDeserializer(ConfluentRegistryAvroDeserializationSchema.
      forSpecific(SignalingTopicRecord.class,
        properties.getProperty("schema.registry.url"),
        properties))
      //deserializzo i messaggi in formato Avro
      //specificando il tipo di record che mi aspetto (SignalingTopicRecord)
      .build();
  }
```

Listing 2: Configurazione del Kafka Consumer

Come mostrato nel codice 2, il *Consumer Kafka* è configurato per leggere i messaggi dal topic *Signaling Topic* ed il deserializzatore utilizzato è `org.apache.flink.api.common.serialization.DeserializationSchema`, una classe messa a disposizione da Flink per deserializzare i messaggi *Avro*. Le righe 15,18,19 e 26 sono tipiche di un *Consumer Kafka* scritto con *Flink*, è presente infatti la creazione del *builder* (riga 15), la definizione del *topic* da cui leggere (riga 18), l'offset da cui iniziare a leggere (riga 19) e la creazione del *Consumer* (riga 26). Invece nelle righe 16 e 20-21 si possono leggere rispettivamente la configurazione del *Kafka Consumer* e la configurazione del deserializzatore *Avro*.

Nella chiamata alla funzione `forSpecific` a riga 21 i parametri sono la classe in cui verrà deserializzato il messaggio, l'url dello *Schema Registry* e le properties contenenti credenziali per accedere allo *Schema Registry*. La configurazione del *Kafka Consumer* e del deserializzatore *Avro* è definita in un file *.properties* e principalmente contiene l'url dell'endpoint, le credenziali e diversi parametri per la connessione. In un caso (*Kafka*) è riferita al *Broker Kafka* e nell'altro allo *Schema Registry* (*Avro*).

3.3 Lettura della tabella di configurazione

La tabella di configurazione è una tabella *SQL* chiamata *TradingPartnerEventLookup* e contiene le regole di filtro per ogni cliente. La connessione al database è gestita tramite *JPA* e la libreria *Hibernate* (sezione 1.3.5) con un sistema di caching per evitare richieste ripetute al database. La cache viene aggiornata ogni ora. La tabella 3.1 è un esempio di come potrebbe essere configurato il microservizio per inviare eventi a diversi clienti, I campi più rilevanti sono:

- **TradingPartner**: il nome del cliente, in questo caso sono tutti clienti fittizi di test.
- **OT**: *BusinessObjectType* il tipo di evento (1 Spedizione – 2 Ritiro – 5 Ordini).
- **ERP**: Lo specifico evento, ad esempio *08* è l'evento di creazione di un ordine, *GEO* è l'evento di modifica della posizione di un ordine.
- **Active**: se il filtro è attivo.

Flink è basato sulla computazione distribuita di stream di dati (vedasi sezione 1.3), ciò va tenuto durante la creazione della cache per la tabella di configurazione. Infatti implementando una semplice cache locale (attraverso una `HashMap` ad esempio), si potrebbero avere problemi di consistenza dei dati e di sincronizzazione tra i vari nodi. Inoltre tale implementazione non sfrutta le potenzialità di *Flink* e la sua gestione automatica della scalabilità. Per mostrare le problematiche di una cache implementata in maniera non corretta, si consideri il seguente esempio:
All'interno di un *job* viene creato un metodo che legge la tabella di configurazione e la memorizza in una `HashMap`, ogni 10 minuti la cache viene aggiornata. Automaticamente il *job* viene scalato su più nodi da *Flink*, ogni nodo avrà la sua copia

Id	TradingPartner	OT	ERP	ID	Depot	Active	QueryCondition
1	TEST_1	5	08	null	null	Y	null
2	TEST_1	5	20	null	null	Y	null
3	TEST_1	5	21	null	null	Y	null
4	TEST_1	5	GEO	null	null	Y	null
5	TEST_2	1	COR	null	null	Y	null
6	TEST_2	1	GEN	null	null	Y	null
7	TEST_3	5	10	null	null	Y	null
8	TEST_4	1	COR	null	null	Y	null
9	TEST_4	1	VIC	null	null	Y	null
10	TEST_4	1	VSC	null	null	Y	null
11	TEST_5	1	CMG	null	null	Y	null
12	TEST_5	1	CMO	null	null	Y	null
13	TEST_5	1	CMR	null	null	Y	null
14	TEST_5	1	COR	null	null	Y	null
15	TEST_5	1	ERR	null	null	Y	null
16	TEST_5	1	GIA	null	null	Y	null
17	TEST_5	1	OAC	null	null	Y	null
18	TEST_5	1	STC	null	null	Y	null

Tabella TradingPartnerEventLookup di esempio

della cache e tale cache verrà aggiornata in maniera indipendente. Questo comporta prima di tutto un problema di consistenza dei dati, in quanto un nodo potrebbe avere una copia della cache diversa qualora fosse impegnato in una computazione al momento dell'aggiornamento. Ma soprattutto ogni nodo effettua una query al database in fase di aggiornamento, causando una serie di richieste tutti identiche e quindi ridondanti. Se ad esempio il *job* viene scalato su 10 nodi, ci saranno 10 richieste al database ad ogni aggiornamento, quando invece ne basterebbe una sola.

Il modo corretto di gestire questa cache è di utilizzare un datastream che legga la tabella di configurazione e la mantenga aggiornata, in collaborazione con il pattern **Broadcast State Pattern** per la distribuzione della cache tra i vari nodi.

3.3.1 Rules Based Stream Processing

Il *Broadcast State Pattern* è un pattern di *Flink* che permette di distribuire uno stato tra tutti i nodi di un job. Questo stato è distribuito in maniera efficiente e scalabile, inoltre è possibile aggiornarlo in maniera asincrona. Il *Broadcast State Pattern* è fondamentale per la creazione di un sistema di *Rules Based Stream Processing*, cioè un sistema che applica delle regole ad un flusso di dati.
Ci sono quindi due flussi di dati, uno contenente le regole e l'altro i dati a cui applicare le regole. Ogni nodo mantiene un *Broadcast State*, cioè una mappa che viene aggiornata all'arrivo di nuove regole. Quando una nuova regola arriva dalla sorgente del flusso di regole, questa viene distribuita a tutti i nodi che andranno ad aggiornare il loro *Broadcast State* con la nuova regola, secondo quanto definito nel codice. In questo modo ogni nodo ha sempre una copia aggiornata delle regole

da poter confrontare con i dati in arrivo sul flusso principale. Quando invece un dato arriva sul flusso principale questi viene direttamente analizzato dal nodo che, eventualmente avvalendosi del *Broadcast State*, lo elabora secondo quanto definito dalla logica implementativa.

Ad esempio (), se una compagnia volesse analizzare le abitudini di acquisto su un e-commerce dei loro clienti potrebbe avvalersi di un *Rules Based Stream Processing*. Quindi si avrebbero due stream, uno contenente tutte le operazioni svolte da un utente, l'altro contenente il pattern che vogliamo analizzare, come mostrato in figura 3.2.

esempio preso documentazio FLink, citarlo

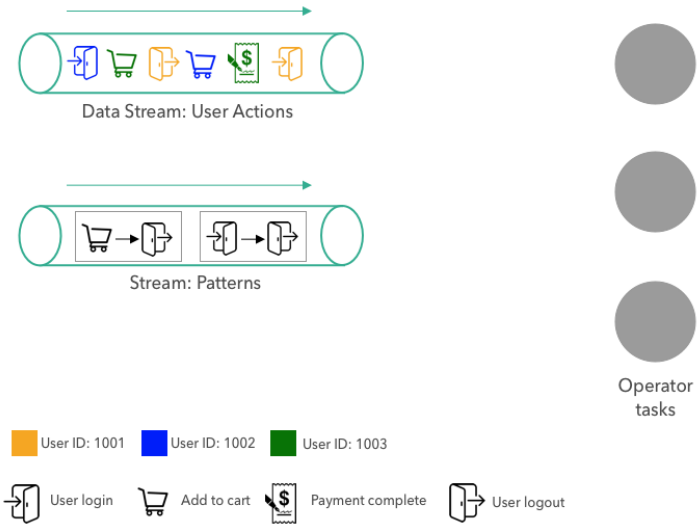

Figura 3.2: Stream di dati e stream di regole

Appena arriva una nuova regola, questa viene distribuita a tutti i nodi, che aggiorneranno i loro *Broadcast State*. Nel caso in figura 3.3 ad esempio, il pattern appena giunto è l'operazione di ingresso al sito seguita da un'uscita, senza acquisti.

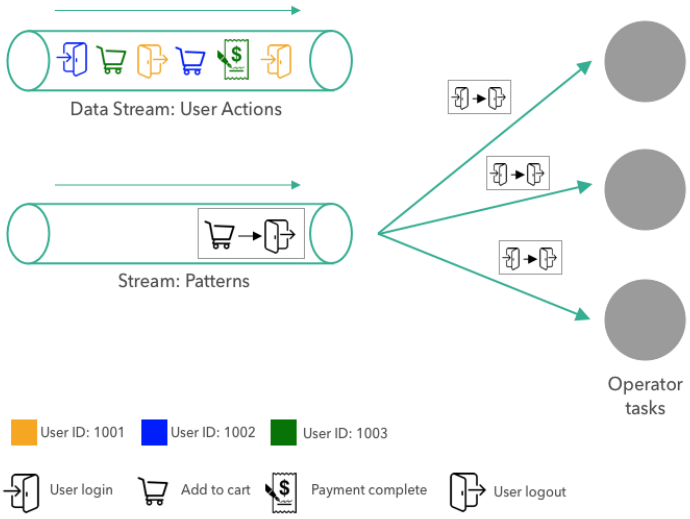

Figura 3.3: Aggiornamento del *Broadcast State*

Successivamente, quando arriva un nuovo dato, questo viene analizzato dal nodo che, avvalendosi del *Broadcast State*, lo elabora secondo quanto definito dalla regola. Ad esempio nella sezione sinistra della figura 3.4, si può notare che viene ricevuto il seguente dato *il cliente 1001 entra nel sito*. Nel frattempo gli altri due clienti effettuano altre operazioni che vengono inviate ai relaviti nodi. Successivamente arriva un altro dato *il cliente 1001 esce dal sito* (parte destra della figura), che viene inviato al primo nodo. Questo nodo, avendo memoria della regola impostata nel *Broadcast State*, riconosce il pattern e lo invia al sistema di analisi.

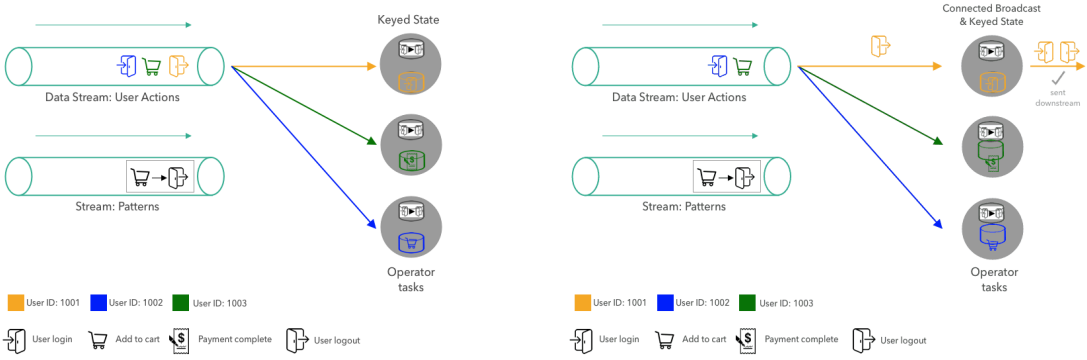

Figura 3.4: Esempio di pattern riconosciuto

3.3.2 Implementazione del Rules Based Stream Processing

Come descritto in precedenza (sezione 3.3.1), il Rules Based Stream Processing è un sistema che applica delle regole ad un flusso di dati. In questo caso le regole sono quelle contenute nella tabella *TradingPartnerEventLookup* e il flusso di dati è quello dei *Business Event* provenienti dal *Signaling Topic*.

Bibliografia

[] *Apache Flink*. URL: https://flink.apache.org/.

[] *Apache Kafka*. URL: https://kafka.apache.org/.

[] *Debezium*. URL: https://debezium.io/.