Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA

in

Tecnologie e Sistemi di Gestione di Basi di Dati e Big Data M

Un Framework di Astrazione per lo Stream Processing a Supporto di RAM^{3}S

Relatore:
Chiar.mo Prof.
Marco Patella

Presentata da:
Nicolò Scarpa

Sessione III
Anno Accademico 2020/2021

Alle Donne nella mia Vita.

Non temere, perché io sono con te;
non smarrirti, perché io sono il tuo Dio.

Isaia 41,10

Sommario

L'elaborazione di quantità di dati sempre crescente ed in tempi ragionevoli è una delle principali sfide tecnologiche del momento. La difficoltà non risiede esclusivamente nel disporre di motori di elaborazione efficienti e in grado di eseguire la computazione coordinata su un'enorme mole di dati, ma anche nel fornire agli sviluppatori di tali applicazioni strumenti di sviluppo che risultino intuitivi nell'utilizzo e facili nella messa in opera, con lo scopo di ridurre il tempo necessario a realizzare concretamente un'idea di applicazione e abbassare le barriere all'ingresso degli strumenti software disponibili. Questo lavoro di tesi prende in esame il progetto RAM^{3}S, il cui intento è quello di semplificare la realizzazione di applicazioni di elaborazione dati basate su piattaforme di *Stream Processing* quali Spark, Storm, Flink e Samza, e si occupa di esaudire il suo scopo originale fornendo un framework astratto ed estensibile per la definizione di applicazioni di stream processing, capaci di eseguire indistintamente sulle piattaforme disponibili sul mercato.

Parole chiave— Stream Processing, RAM3S, Framework, Big Data, Spark, Storm, Flink, Samza, RabbitMQ, Kafka, SPAF

Ringraziamenti

Innanzitutto vorrei ringraziare il Prof. Marco Patella per avermi proposto questo lavoro di tesi, ma soprattutto per la grande disponibilità al confronto verbale concessami durante l'intera fase realizzativa del progetto; numerosi sono stati gli spunti sorti durante questi incontri, resi piacevoli anche da momenti di ilarità. Per me è stato un onore lavorare a questo progetto.

Vorrei inoltre ringraziare tutti i colleghi incontrati durante questo percorso, ho avuto la fortuna di conoscere persone brillanti, disponibili e serie; ricorderò sempre con piacere i momenti di studio passati con alcuni colleghi e gli attimi di leggerezza che ci siamo regalati.

Ringrazio anche tutti i professori incontrati in questi anni, per gli insegnamenti ricevuti, ma soprattutto per le sfide che mi sono state poste; grazie a queste mi sono confrontato con me stesso e superandole sono cresciuto.

Un sentito ringraziamento anche tutto il personale dell'Università di Bologna; alle persone che, in ogni ruolo e in ogni modo, rendono vivo questo ambiente e permettono a noi studenti di sentirci accolti e al sicuro.

Il ringraziamento più grande, immenso, è per la mia fidanzata Elisabetta. Hai condiviso con me tutto di questo percorso. Non riesco a descrivere a parole quanto questa cosa sia per me importante, la più importante.

Un ringraziamento speciale va alla mia famiglia di Prato, perché nel mio cuore li considero tali: Giulia, per l'affetto che provi per me e per avermi fatto prendere sempre il massimo dei voti da quando sei arrivata al mondo; Valentina, per l'interesse e l'entusiasmo con cui hai condiviso il superamento dei miei esami e per le chiacchierate distensive che mi hai regalato; Alessio, per aver ascoltato le mie perplessità e non aver giudicato i miei limiti, ti considero un fratello maggiore; Patrizia, per l'attenzione con cui hai seguito

tutto questo percorso, per le preghiere che lo hanno reso sereno e per la tua remota ma costante vicinanza; Giuseppe, per avermi trasmesso fiducia, per la considerazione e per avermi ispirato a cercare solidità in questo percorso.

Ringrazio con tutto me stesso la mia famiglia, per il rispetto che mi ha dimostrato durante questo percorso: Eleonora, mia sorella, per il costante interesse e supporto che ha manifestato durante l'intero percorso, e per i momenti di festa che ha organizzato; Rosalba, mia madre, per l'energia e la determinazione che mi ha donato, e per aver riletto la mia tesi; Gianluca, mio padre, per la vena tecnica e la curiosità innata che mi ha trasmesso, e per avermi avvicinato all'informatica.

Ringrazio tanto mia nonna Ersilia, per il suo amore incondizionato, per la sua sensibilità discreta e per l'incrollabile forza che mi insegna.

Ringrazio i miei nonni che non ci sono più: Luisa, per la grande dignità che hai sempre trasmesso; Silvio, avrei voluto conoscerti con tutto il cuore, grazie per la serenità che mi hai regalato quando sono venuto a trovarti a Brisighella, nei viaggi di ritorno da Bologna; Dino, per l'attenzione ai particolari e l'amore per il rispetto che mi hai insegnato, e per avermi aiutato a lottare fino in fondo; Elisabetta (Lisetta), per la solennità della tua presenza e per l'amore con cui mi hai accolto, la tua presenza è viva più che mai.

Un pensiero va anche ai miei zii, Graziella e Daniele, e alle mie cugine, Isabella e Valentina; ci siamo frequentati poco durante questo periodo, ma ho sempre sentito la vostra vicinanza.

Ringrazio coloro che mi hanno indirizzato, pensato e in qualche modo aiutato durante questi anni; temo di aver dimenticato qualcuno, ma il mio ringraziamento è nel mio cuore, lì per voi.

Prima di concludere, anche se probabilmente non leggerà mai queste parole, ringrazio il mio giocatore di pallacanestro preferito Stephen Curry, per avermi ispirato ad affrontare le sfide sempre con il sorriso e anche divertendomi, ma soprattutto per la sua testimonianza che «Tutto posso in colui che mi dà la forza.» (Filippesi 4,13).

Ringrazio Gesù, Maria e Giuseppe.

Ringrazio Dio.

Indice

Elenco delle figure

Prefazione

La soluzione di un problema tramite programmazione di codice può risultare un'esperienza confusa e, a volte, angosciante; si alternano l'utilizzo dell'istinto e della logica, ma ciò può non bastare; entrambe le modalità necessitano, se non di conoscenza, almeno di informazioni; senza avere alcuna certezza, si corre il rischio di provare approcci e strumenti in maniera disordinata e di perdersi.

L'ambito dello stream processing appariva, almeno all'autore, come un mondo la cui esplorazione dovesse risultare ardua e che dovesse richiedere molto tempo; il numero di strumenti di programmazione e di sistemi di vario tipo esistenti nel contesto dello stream processing è numeroso; chi vuole approcciarsi a questo mondo può ritrovarsi spaesato e non sapere da dove iniziare.

RAM^{3}S si pone proprio l'obiettivo di accompagnare lo sviluppatore nel mondo dello stream processing, facilitandone l'ingresso. Uno strumento di questo genere, perdipiù proveniente da un ambito sconosciuto, ha suscitato nell'autore un certo interesse. Un tale strumento non può nascere dal nulla, deve necessariamente essere sviluppato a partire da un'esperienza diretta con il problema da risolvere. Oltre allo strumento, dunque, c'è qualcosa di più profondo. C'è il viaggio di chi lo ha creato; c'è l'esperienza di chi ha superato gli ostacoli; c'è la sicurezza di chi ha risolto almeno una volta quel problema.

Avere dunque la possibilità di prendersi cura di uno strumento di questo tipo, di metterlo a posto, di valorizzarlo, ha costituito per l'autore un'occasione a cui non voler rinunciare. D'altra parte, risultava chiaro fin dall'inizio come il progetto risultasse sfidante dal punto di vista ingegneristico e fornisse un buon banco di prova dove collaudare le conoscenze acquisite durante il percorso di studi.

Come RAM^{3}S ha accompagnato l'autore nel mondo dello stream processing, ora sarà l'autore ad accompagnare il lettore nel percorso di realizzazione di questo lavoro di tesi.

Nicolò Scarpa

Introduzione

L'ambito dello Stream Processing è sufficientemente ampio e complesso in termini di concetti, strumenti e pratiche di programmazione da richiedere una buona dose di studio prima di immedesimarsi nella realizzazione di un'applicazione di elaborazione dati.

Il progetto RAM^{3}S è nato per facilitare al programmatore estraneo al mondo dello Stream Processing la realizzazione di una applicazione di analisi di grosse quantità di dati e il test sulle piattaforme di stream processing disponibili sul mercato.

Questo progetto di tesi ha l'obiettivo di migliorare il "come" RAM^{3}S realizza il proprio scopo, mantenento inalterato il "cosa". Vedremo che questo obiettivo verrà raggiunto tramite la realizzazione di un framework astratto per la creazione di applicazioni di Stream Processing.

Affronteremo il percorso realizzativo di questo lavoro di tesi iniziando dalla descrizione dello stato attuale del progetto RAM^{3}S, alla luce degli ultimi sviluppi apportati dai recenti lavori di tesi dei colleghi; passeremo dunque alla descrizione della fase realizzativa di questo progetto, affrontando l'esposizione secondo il classico percorso proposto dal processo di sviluppo software nell'ingegneria; esporremo dunque alcuni aspetti peculiari delle piattaforme di stream processing considerate e un possibile utilizzo (per così dire avanzato) del progetto frutto di questo lavoro; concluderemo dunque esponendo i limiti di questa prima versione del progetto, ma fornendo numerosi spunti per superare tali limiti tramite possibili sviluppi futuri.

Prima di iniziare l'esposizione del progetto, è utile chiarire meglio che cosa sia lo **stream processing** e cosa si intende per **framework**.

Che cos'è lo Stream Processing?

Lo Stream Processing (in italiano, "elaborazione di flussi") è un ambito dei Big Data che ha l'obiettivo di fornire strumenti di programmazione che consentano di elaborare enormi quantità di dati nella maniera più efficiente possibile. La mole di dati che un'applicazione di stream processing può trovarsi a gestire può essere tale da non consentire l'elaborazione completa di tutti i dati, questo non solo per limiti di tempo ma anche per limiti di risorse (come la banda delle connessioni, lo spazio di archiviazione, la quantità di memoria RAM e le CPU). Per queste ragioni, l'infrastruttura tipica di un sistema di stream processing ricalca quella dei sistemi distribuiti di livello enterprise; ovvero, è previsto l'utilizzo di cluster di macchine connesse tra loro e in grado di coordinarsi per l'esecuzione della logica di elaborazione sui dati. In questo scenario, negli ultimi dieci anni circa, sono nate numerose proposte di piattaforme software che si specializzano nella risoluzione dei problemi sopra esposti, consentendo di definire applicazioni di elaborazione dati in modo semplficato e occupandosi della distribuzione del calcolo su nodi di un cluster.

Le piattaforme così nate prendono il nome di framework di stream processing e offrono un modello di programmazione relativamente semplice e basato sul concetto di topologia, di cui daremo una definizione precisa nel secondo capitolo. Questi framework sono corredati solitamente da un ambiente di esecuzione che permette di eseguire in maniera distribuita le applicazioni così definite.

Che cos'è un Framework?

Per dare la definizione di framework, aiutiamoci con quella fornita nel libro *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1994a):

> *A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software (Deutsch 1989). [...] You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework.*

Un framework, pertanto, determina l'architettura generale di un'applicazione (Gamma et al. 1994a). Un'applicazione sviluppata utilizzando un certo framework non potrà prescindere da utilizzarne le classi e gli oggetti, e adeguarsi a tutto ciò che ne consegue,

come le ripartizione delle responsabilità, le modalità di collaborazione tra le classi e il controllo esercitato a tempo di esecuzione (Gamma et al. 1994a).

In sostanza, i framework sono per le applicazioni quello che le classi sono per gli oggetti, solo a un livello macroscopico. Le classi costituiscono l'elemento fondamentale per la riutilizzabilità del codice nella programmazione orientata agli oggetti. I framework, invece, si concentrano sulla **riutilizzabilità** dell'architettura di una soluzione, esponendo ai programmatori di applicazioni solo i parametri necessari a realizzare il comportamento desiderato da questa, messo in atto tramite i meccanismi interni del framework stesso.

Quanto detto significa che le applicazioni sviluppate su di un framework hanno certi gradi di libertà, ma questi gradi di libertà debbono essere previsti a priori dal framework. Per quest'ultima ragione, è di fondamentale importanza che il framework sia connotato anche dalla qualità di **estensibilità**, poiché è praticamente impossibile prevedere a priori tutte le necessità delle applicazioni concrete; inoltre, tramite questa qualità, il framework sarà in grado di meglio supportare la prova del tempo, poiché potrà eventualmente evolversi tramite possibili integrazioni.

Maggiore è la complessità intrinseca della categoria di applicazioni che si intende sviluppare, più senso ha prevedere per queste un framework che incarni i meccanismi per la gestione di tale complessità e le decisioni di progettazione che inevitabilmente si dovrebbero prendere.

Le applicazioni di stream processing sono complesse poiché devono considerare la gestione delle sorgenti e destinazioni dati e la molteplicità dei sistemi che le realizzano; devono ovviamente prevedere la possibilità di specificare le operazioni di elaborazione sui dati, favorendo anche in questo caso la riutilizzabilità della logica già realizzata da altri programmatori; devono infine gestire tutti gli aspetti operativi relativi all'esecuzione dei concetti sopra elencati, aggiungendo il grado di complessità non indifferente dato dalla volontà di distribuire il calcolo su più macchine.

Per queste ragioni, l'ambito dello stream processing costituisce un ottimo candidato per la realizzazione di framework. In questo progetto, ci occuperemo di realizzare un'astrazione dei framework esistenti per lo stream processing, con lo scopo di razionalizzarne l'interfaccia di programmazione e dunque l'utilizzo.

Capitolo 1

RAM^{3}S - Lo stato attuale

In questo capitolo analizzeremo lo stato attuale del codice di RAM^{3}S e cercheremo di capire qual è il suo approccio per consentire l'utilizzo facilitato di un framework di stream processing per la realizzazione di un'applicazione di elaborazione dati.

Ci concentreremo sia sulla modalità di utilizzo dei framework sottostanti che sulla modalità con cui RAM^{3}S si interfaccia ai sistemi a code di messaggi, impiegati da RAM^{3}S come sorgenti e destinazioni dei dati elaborati.

Trarremo infine delle conclusioni sulla natura del progetto, per fornire una categorizzazione più precisa di RAM^{3}S come strumento di sviluppo software.

1.1 Modalità di programmazione

La versione attuale di RAM^{3}S consente di sperimentare i vari framework di stream processing mettendo a disposizione tre applicazioni di esempio differenti, tutte basate sul processamento di immagini in input:

- *plate*: riconoscimento delle targe;
- *ocr*: riconoscimento di testo;
- *face*: riconoscimento di volti;

Come vedremo tra breve, RAM^{3}S tenta di astrarre l'utilizzo dei framework di stream processing fornendo un'applicazione generica distinta per ognuno di questi. Ciascuna di queste applicazioni consente l'esecuzione degli esempi sopra citati sul rispettivo framework di stream processing. Tra le applicazioni generiche e le applicazioni di esempio, RAM^{3}S

interpone uno strato di astrazione basato su alcune interfacce. Tali interfacce tentano di modellare da una parte gli aspetti di processamento dei dati, dall'altra gli aspetti di ricezione dei dati.

Prendiamo ora in esame l'insieme di interfacce definite da RAM^{3}S per astrarre i concetti di stream processing. Dopo di ciò, vedremo come tali interfacce vengono usate dalle applicazioni di esempio e dalle applicazioni generiche messe a disposizione dallo stesso RAM^{3}S.

La Figura 1.1 mostra tutte e sole le interfacce predisposte da RAM^{3}S per modellare un'applicazione di stream processing.

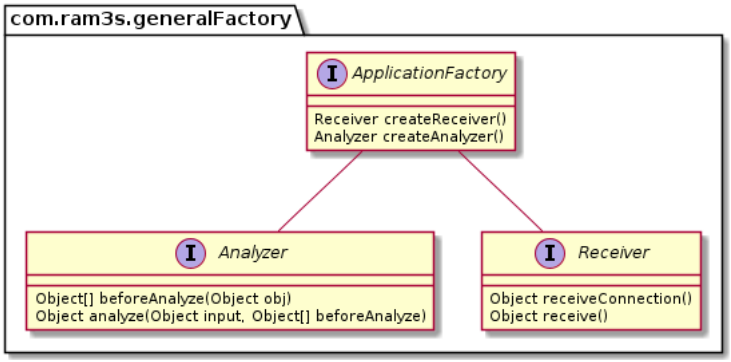

Figura 1.1: Interfaccia di programmazione di RAM^{3}S

In sostanza questo strato costituisce le API di RAM^{3}S, poiché vedremo presto come le applicazioni di esempio, che rappresentano appunto ciò che dovrebbe realizzare il programmatore finale, utilizzino direttamente le interfacce in questione.

Vediamo dunque il significato di ciascuna interfaccia e qual è il loro utilizzo inteso per la realizzazione di un'applicazione di stream processing. Per determinare il significato dei singoli metodi si è fatto riferimento al codice dell'applicazione di esempio per il riconoscimento dei volti, di cui si mostra il diagramma delle classi nella Figura 1.2.

- `ApplicationFactory`: questa interfaccia si occupa di rappresentare l'applicazione come unità; in sostanza, serve da raccoglitore per l'implementazione dell'*Analyzer* e per l'implementazione del *Receiver*; all'atto pratico, si occupa infatti di istanziare le classi concrete di tipo `Analyzer` e `Receiver` definite nel contesto dell'applicazione.

- `Analyzer`: questa interfaccia rappresenta il contenitore di tutta la logica di elaborazione dell'applicazione; propone di "spezzare" la logica applicativa in due fasi: fase

di pre-elaborazione e fase di elaborazione; all'atto pratico, le due fasi devono essere implementate rispettivamente nei metodi `beforeAnalyze` e `Analyze`; va però sottolineato che per ogni oggetto da sottoporre all'elaborazione verranno eseguiti entrambi i metodi (come vedremo presto dal codice di un'applicazione generica); pertanto, il metodo `beforeAnalyze` non va inteso come luogo dove compiere operazioni da eseguire una volta sola durante tutta la vita dell'applicazione.

- `Receiver`: questa interfaccia tenta di rappresentare il sistema esterno da cui l'applicazione dovrà ricevere dati; il significato dei due metodi dell'interfaccia si è inferito dal codice delle classi concrete, definite per le applicazioni di esempio; il metodo `receiveConnection` sembra servire alla restituizione dei parametri di connessione al sistema esterno (non alla restituzione della connessione vera e propria, come invece il nome dà a intendere); il metodo `receive` sembra invece inutile, poiché la maggior parte delle implementazioni restituiscono l'oggetto nullo (probabilmente era inteso per accogliere la logica di ricezione del singolo oggetto dal sistema esterno).

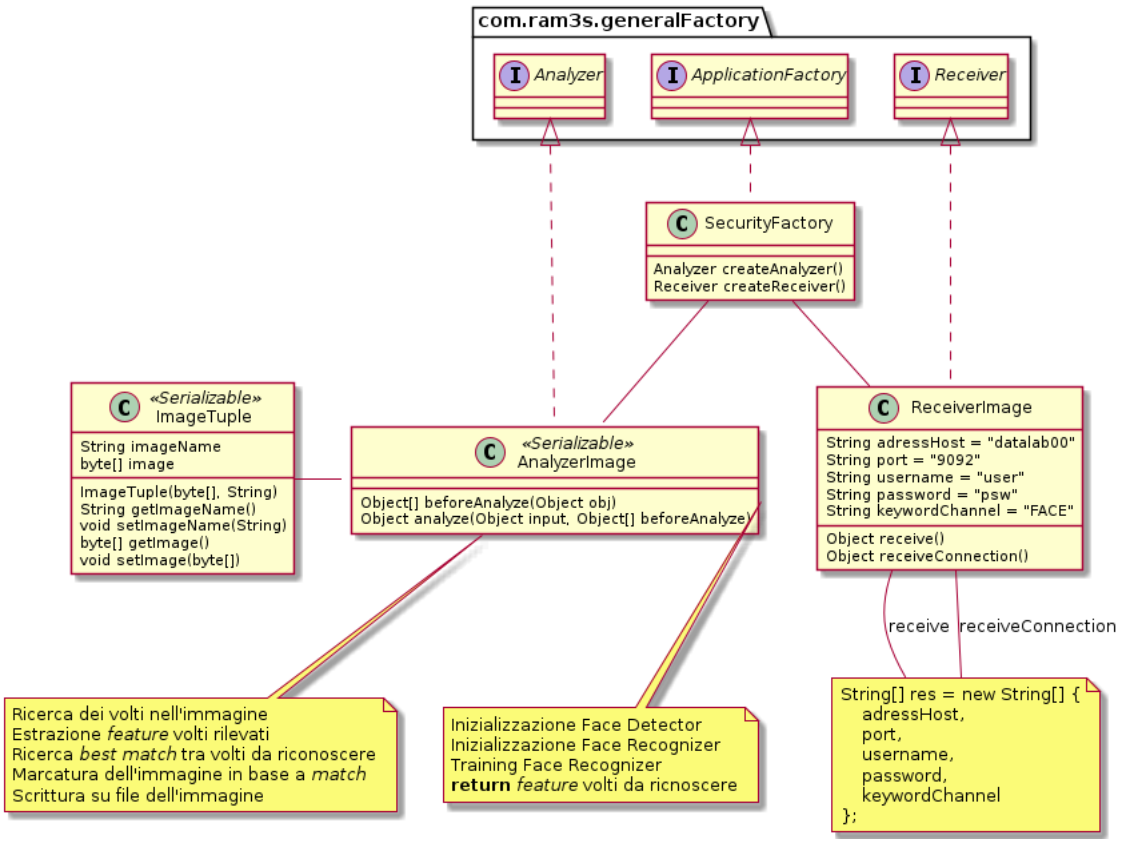

Figura 1.2: Applicazione RAM^{3}S per riconoscimento volti

Spositamo ora l'attenzione su come vengono utilizzate le interfacce di RAM^{3}S dalle applicazioni generiche dei vari framework di stream processing. Lo scopo di un'applicazione generica è quello di mappare sul relativo framework l'applicazione definita in termini delle interfacce sopra esposte, dunque di mettere in esecuzione la logica applicativa sul *runtime* sottostante. Nella Figura 1.3 si propone un diagramma delle classi che contestualizza le applicazioni generiche messe a disposizione da RAM^{3}S rispetto alle interfacce di RAM^{3}S stesso e ai vari framework di stream processing.

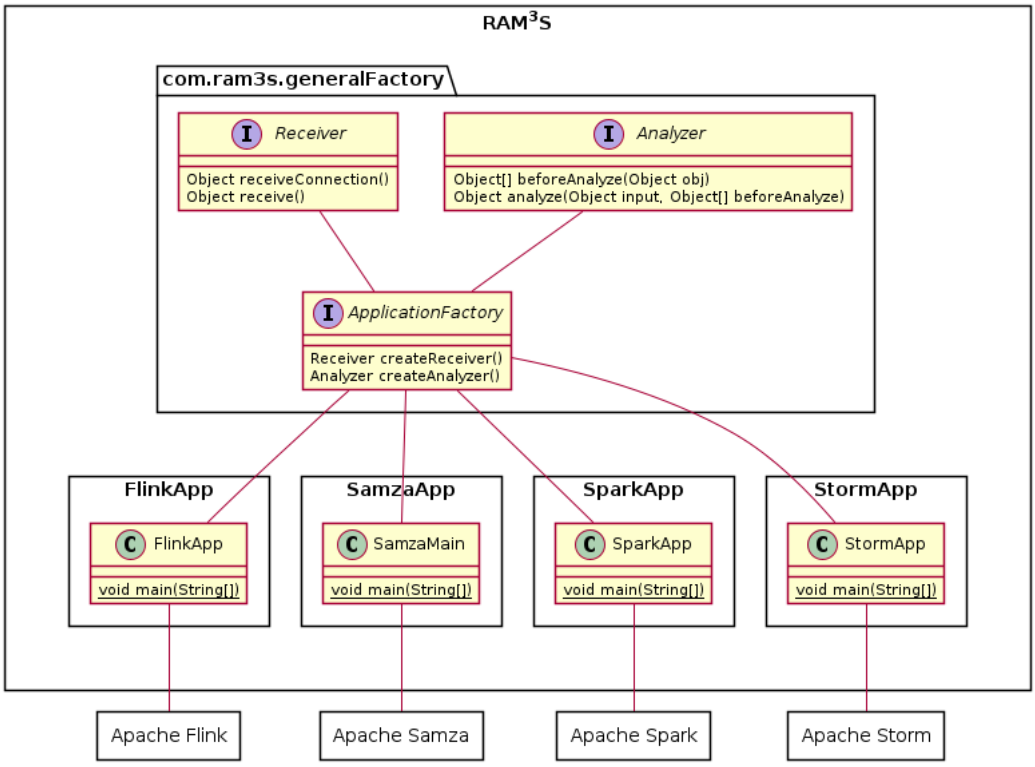

Figura 1.3: Applicazioni RAM^{3}S per ciascun framework di stream processing

Il codice delle applicazioni generiche è specificato, quasi sempre, completamente dentro il metodo `main` e ha una struttura ricorrente. Proponiamo qui sotto il listato rappresentate tale codice (al quale sono state apportate alcune semplificazioni non significative, per ragioni espositive).

```
public static void main(String[] args) {
    String appFact = args[0];
    String outputStream = args[1];
    ApplicationFactory factory = null;

    if (appFact.equalsIgnoreCase("plate")) {
        factory = PlateImageFactory.class.newInstance();
    } else if (appFact.equalsIgnoreCase("ocr")) {
```

```
        factory = OCRimageFactory.class.newInstance();
    } else if (appFact.equalsIgnoreCase("frame")) {
        factory = FrameImageFactory.class.newInstance();
    } else if (appFact.equalsIgnoreCase("face")) {
        factory = SecurityFactory.class.newInstance();
    } else if (appFact.equalsIgnoreCase("mock")) {
        factory = MockApplicationFactory.class.newInstance();
    } else {
        System.out.println("Factory " + appFact + " non valida!");
        System.exit(-2);
    }

    Receiver receiver = factory.createReceiver();
    String[] connection = (String[]) receiver.receiveConnection();
    Analyzer analyzer = factory.createAnalyzer();

   /* Boilerplate Code dipendente dal Framework di Stream Processing */
    ...
}
```

In sostanza, la struttura del codice di un'applicazione generica è così composta:

1. selezione di una delle applicazioni di esempio sopra elencate, basandosi sugli argomenti con i quali l'applicazione generica è stata lanciata, e creazione del corrispondente oggetto `ApplicationFactory`, rappresentante l'applicazione stessa;

2. recupero degli oggetti di tipo `Receiver` e `Analyzer`, contenenti rispettivamente le informazioni per la connessione al sistema dati esterno e la logica di elaborazione dati dell'applicazione;

3. traduzione degli oggetti recuperati al punto 2. negli oggetti e procedure peculiari esposte dalle API del framework sottostante, con lo scopo di stabilire concretamente la connessione al sistema dati esterno e di mettere in atto la logica applicativa. Il codice di questa fase non è stato riportato poiché è diverso per ciascuna applicazione generica, possiamo però dire che si tratta di cosiddetto *boilerplate code* peculiare al framework sottostante.

Dobbiamo evidenziare il fatto che le applicazioni generiche messe a disposizione da RAM^{3}S sono in realtà fortemente dipendenti dalle applicazioni di esempio, pertanto la creazione di una nuova applicazione di stream processing non può avvenire senza modificare il codice sorgente di RAM^{3}S stesso.

Poiché non è nello scopo del nostro progetto entrare nei dettagli della versione attuale di RAM^{3}S, tralasceremo l'esposizione di come avvengono nel concreto le operazioni descritte al punto 3. della struttura appena illustrata.

In ogni caso, va rilevato che già in questa versione di RAM^{3}S si era intuita la necessità di gestire un altro aspetto fondamentale nella creazione di un'applicazione di stream processing: la definizione della sorgente dati di un'applicazione e la creazione della relativa connessione. Questa intuizione è stata in realtà approfondita e resa più esplicita nello strato di supporto ai sistemi dati di tipo *message broker* (o sistemi a code di messaggi) realizzato dal lavoro di tesi precedente a questo (cfr. Berni 2021). Poiché tale strato ha in parte ispirato alcune scelte implementative nel nuovo progetto, e vi si sono riscontrate delle scelte progettuali affini, ne riporteremo nella prossima sezione i dettagli più rilevanti.

1.1.1 Supporto ai *Message Broker*

La prima versione di RAM^{3}S prevedeva il solo supporto di RabbitMQ come sistema dati esterno, e ne consentiva l'utilizzo come sola sorgente dati. Nell'ambito di un successivo progetto di estensione di RAM^{3}S (cfr. Berni 2021), è stato invece realizzato uno strato supplementare per integrare sistemi dati esterni di altro tipo, tra cui il *broker* di messaggi Apache Kafka. Vedremo come tale supporto dovrà necessariamente tenere conto delle peculiarità dei singoli framework di stream processing presi in considerazione da RAM^{3}S.

Aiutiamoci con il diagramma delle classi mostrato in Figura 1.4 per descrivere questo strato di astrazione verso i sistemi a code di messaggi.

Possiamo dividere la trattazione in due blocchi principali: strato di astrazione rivolto ai *message broker*, strato implementativo peculiare al framework di stream processing.

Lo strato di astrazione rivolto ai *message broker* è costituito dal package `messageBroker` mostrato in figura. All'interno di questo package ci sono due categorie di classi: le interfacce per la rappresentazione astratta di "lettori" (o *reader*) e "scrittori" (o *writer*) per sistemi di tipo *message broker* (interfacce `Reader` e `Writer`); le classi concrete per l'implementazione di *reader* e *writer* per uno specifico *message broker* (qui abbiamo riportato solo quelle per Apache Kafka, `KafkaReader` e `KafkaWriter`).

Analizziamo prima il significato dei metodi dell'interfacca `Reader`, alla luce dell'implementazione concreta fornita per Kafka:

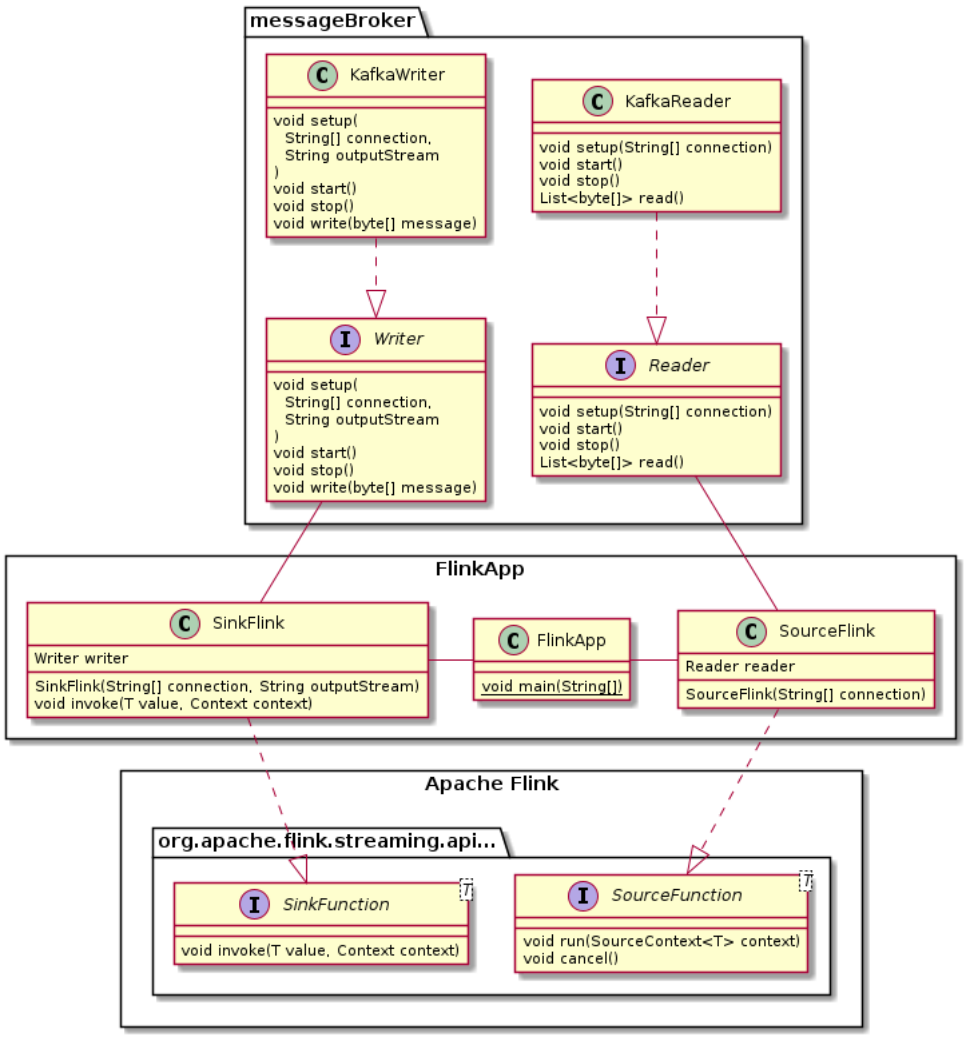

Figura 1.4: Strato di supporto a *message broker* per RAM^{3}S

- `setup(String[] connection)`: questo metodo riceve in input un array di stringhe, da interpretare in maniera posizionale, contenente i parametri di connessione al sistema dati esterno (questo array corrisponde al valore di ritorno dell'implementazione del metodo `receiveConnection` dell'interfaccia `Receiver`, esposta nella sezione precedente); nel caso di Apache Kafka, un parametro significativo è il nome del *topic* da cui consumare messaggi; questo metodo, dunque, si occupa di inizializzare gli oggetti necessari a stabilire la connessione con il sistema esterno, eventualmente instaurandola immediatamente;

- `start()`: questo metodo in realtà ha ricevuto un'implementazione vuota per ciascun *message broker* integrato (Kafka e RabbitMQ); proabilmente è stato previsto per librerie client di sistemi esterni che separano la fase di configurazione da quella di

avvio della connessione;

- `stop()`: questo metodo prevede, senza troppe sorprese, di terminare la connessione;
- `List<byte[]> read()`: questo metodo consente di specificare le operazioni per la ricezione di un lotto di messaggi dal sistema esterno, e di restituirne una rappresentazione come array di byte per singolo messaggio; nel caso di Kafka, è stata adottata la modalità *polling* offerta dalla libreria client.

Possiamo ora chiarire il funzionamento dell'interfaccia `Writer`, andando a descrivere l'unico metodo che ha semantica differente da quelli appena esposti:

- `void write(byte[] message)`: questo metodo consente di definire le operazioni per l'invio di un singolo messaggio al sistema esterno, accettando una rappresentazione dello stesso sotto forma di array di byte; nel caso di Kafka, il messaggio viene inviato al *topic* specificato dal parametro `outputStream`, passato al metodo setup.

Passiamo all'analisi dello strato implementativo specifico per il framework di stream processing. Le classi di questo strato realizzano la mappatura tra i concetti di *reader* e *writer* appena esposti e i concetti di *source* e *sink* che, come vedremo nel prossimo capitolo, sono tipici del dominio dello stream processing, e che rappresentano rispettivamente la sorgente e la destinazione dei dati processati dall'applicazione. Per questo "lato" dello strato di supporto, prendiamo come riferimento quanto previsto per Apache Flink.

- La classe `SourceFlink` mantiene, da una parte, un riferimento a un oggetto di tipo `Reader` di cui abbiamo appena chiarito il significato e, dall'altra, utilizza le API specifiche del framework sottostante per realizzare, secondo le modalità da questo specificate, una sorgente dati in grado di alimentare l'applicazione; nell'esempio in questione, viene implementata l'interfaccia `SourceFunction` delle API di Flink, che prevede l'implementazione del metodo `run` per avviare il consumo dei messaggi (realizzato concretamente dall'oggetto `KafkaReader`) e l'implementazione del metodo `cancel` per terminare il consumo.
- La classe `SinkFlink`, dualmente, mantiene un riferimento a un `Writer`, e implementa il metodo `invoke` dell'interfaccia `SinkFunction` di Flink per realizzare il concetto di

destinazione per i dati processati dall'applicazione; in questo esempio, le operazioni di invio dei messaggi vengono delegate all'oggetto `KafkaWriter`.

Infine, dal diagramma in Figura 1.4, si noti che le classi `SourceFlink` e `SinkFlink` vengono utilizzate direttamente dall'applicazione generica messa a disposizione da RAM^{3}S per Apache Flink, realizzata dalla classe `FlinkApp`.

Concludiamo l'ispezione del codice di questa parte di RAM^{3}S sottolineando il fatto che per ciascuna applicazione generica sono, pertanto, previste una coppia di classi che implementano i concetti di sorgente e destinazione con le API del framework sottostante, utilizzando le astrazioni di *reader* e *writer* per comunicare con i *message broker* supportati.

1.2 Modalità di esecuzione

La messa in opera di un'applicazione RAM^{3}S su un certo framework di stream processing si riconduce all'esecuzione di una delle tre applicazioni generiche di cui abbiamo parlato.

I framework di stream processing sono pensati per eseguire applicazioni in maniera distribuita, su di un cluster di nodi. Ha perfettamente senso, però, eseguire un'applicazione anche in maniera locale, sulla macchina dello sviluppatore per consentire il debug e il test dell'applicazione.

Prima di eseguire un'applicazione RAM^{3}S è necessario produrre il pacchetto *jar* corrispondente; ciò può avvenire avvalendosi delle funzionalità dell'IDE utilizzato (ad esempio, Eclipse).

Anticipiamo che non illustreremo le modalità di avvio dei cluster corrispondenti a ciascun framework di stream processing; per informazioni a riguardo, si rimanda alle documentazioni ufficiali di ciascun framework.

Vediamo dunque come ciascuna applicazione generica può essere messa in esecuzione in modalità locale o distribuita, partendo dal *jar* corrispondente:

- *SparkApp.jar*: il *jar* può essere invocato localmente da riga di comando; oppure è possibile utilizzare lo script di deployment su cluster `spark-submit`, messo a disposizione dal framework (*Spark Streaming - Spark 3.2.1 Documentation* 2022).

- *StormApp.jar*: come nel caso dell'applicazione su Spark, il *jar* può essere eseguito da riga di comando; oppure è possibile avvalersi del client Bash `storm`, incluso nella release binaria del framework (*Running Topologies on a Production Cluster* 2022), per sottoporre l'applicazione al cluster.

- *FlinkApp.jar*: il *jar* può essere messo in esecuzione localmente invocandolo da riga di comando; oppure può essere caricato sul cluster Flink tramite l'interfaccia web messa a disposizione dal framework (*Deployment - Resource Providers - Standalone* 2022), o inviato tramite lo script `./bin/flink run` incluso nella release binaria di Flink.

- *SamzaApp.jar*: l'esecuzione in locale può avvenire come negli altri casi, invocando il *jar* da riga di comando. Per quanto riguarda l'esecuzione distribuita, invece, non c'è una modalità equivalente agli altri framework, che consente di inviare direttamente il *jar* al cluster; Samza delega la gestione del cluster a YARN, che richiede la creazione di un pacchetto *tar*, avente una determinata struttura interna, e l'utilizzo dello script `run-app.sh` messo a disposizione dal frameowrk (*Deployment - Run on YARN* 2022), per sottoporre un'applicazione al cluster; allo stato attuale, per eseguire l'applicazione in maniera distribuita, è necessario avviare manualmente il *jar* localmente a ogni macchina *worker* del cluster Samza.

I *jar* di cui abbiamo parlato richiedono alcuni parametri di avvio (Sconosciuto 2019), da specificare all'atto dell'invocazione da linea di comando, o tramite le modalità messe a disposizione da ciascun framework nelle procedure di deployment su cluster: il primo parametro è l'unico obbligatorio e indica quale applicazione di esempio mettere in esecuzione (`PLATE | FACE | OCR`); il secondo parametro indica se prendere i dati in input da RabbitMQ o da una cartella del filesystem condiviso dal cluster; il terzo parametro indica se inviare, o meno, in output i dati verso RabbitMQ. I parametri di connessione a RabbitMQ sono attualmente cablati nelle varie classi di tipo `Receiver` di ciascuna applicazione di esempio.

1.3 Quasi un framework

Dall'analisi che abbiamo presentato possiamo concludere che, allo stato attuale, RAM^{3}S consente di eseguire in maniera facilitata tre applicazioni di stream processing differenti, ma non è ancora in grado di consentire la definizione di una nuova applicazione senza dover intervenire su parte del codice del progetto stesso (ci riferiamo al codice delle applicazioni generiche).

Questo inconveniente colloca RAM^{3}S nella posizione di "quasi-framework" rispetto a una sua categorizzazione come strumento di sviluppo e alla luce della definizione di framework data nel capitolo introduttivo.

Bisogna invece trarre conclusioni differenti per la sola parte di supporto ai *message broker* poiché, come abbiamo potuto apprezzare, è indipendente dal codice dell'applicazione (soddisfa, quindi, il requisito di riutilizzabilità) e consente di disaccoppiare la realizzazione degli "adattatori" di lettura e scrittura di un certo *message broker* dai framework di stream processing sottostanti (offrendo dunque anche estensibilità).

Ci avviamo dunque alla trattazione del lavoro compiuto nell'ambito di questo progetto, che ha come obiettivo principale quello di consentire l'utilizzo di RAM^{3}S secondo l'intento originale, ossia di facilitare la creazione di nuove applicazioni di stream processing, cercando però di ottenere le qualità di riutilizzabilità ed estensibilità.

Capitolo 2

SPAF - Un framework per RAM^{3}S

Anziché operare una ristrutturazione e una conseguente estensione del codice esistente del progetto RAM^{3}S, si è preferito realizzare da zero un nuovo progetto. Questa scelta è stata dettata dalle seguenti motivazioni, documentate nel capitolo precedente:

- scarsa riutilizzabilità del codice esistente;
- interfaccia di programmazione esposta (API) non intuitiva e incompleta;
- mancanza di documentazione e difficile intelleggibilità del codice.

Si è scelto di dare a questo nuovo progetto una vita e un nome propri, e di progettarlo in maniera tale da essere considerabile come un framework vero e proprio. Il framework così realizzato costituirà dunque la base di supporto su cui realizzare il nuovo RAM^{3}S.

RAM^{3}S dunque sarà supportato da un nuovo Framework di Astrazione per lo Stream Processing, il cui nome deriva dalla sua stessa definizione in lingua inglese, ovvero: *Stream Processing Abastraction Framework* - **SPAF**[1].

Il nuovo RAM^{3}S, pertanto, non sarà un framework in sé per sé, ma un *toolchain* che utilizzerà il nuovo framework SPAF per realizzare il proprio intento originale: facilitare la creazione di applicazioni di stream processing e consentirne l'esecuzione sui vari framework disponibili sul mercato in maniera trasparente al programmatore.

Lo schema illustrato nella Figura 2.1 cerca di catturare la natura del nuovo RAM^{3}S, mettendola a confronto con quella della versione precedente.

[1]Il repository Git del progetto: https://github.com/nickshoe/spaf

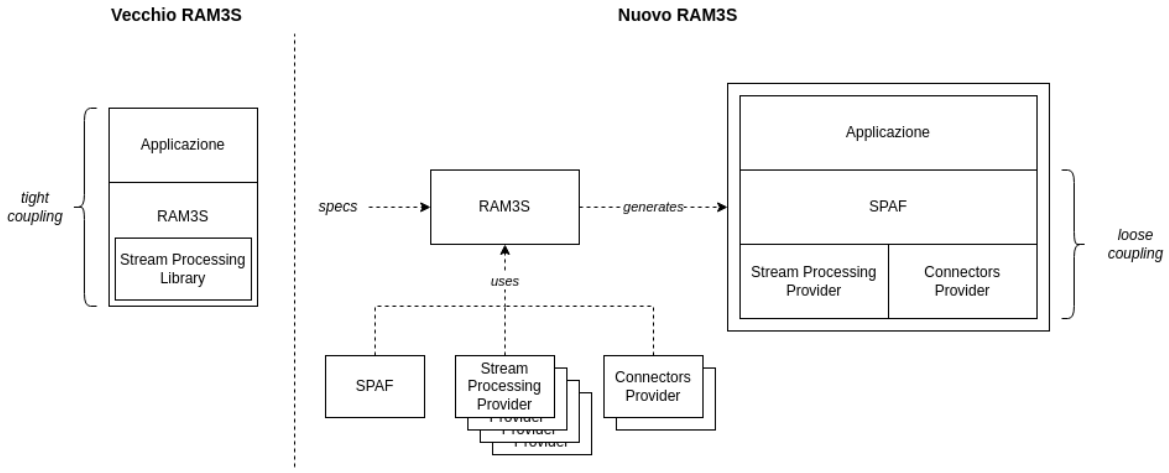

Figura 2.1: Vecchio e nuovo RAM^{3}S a confronto.

Prima di iniziare la fase realizzativa di questo progetto, si è pensato che fosse necessario stabilire un percorso logico costituito da tappe intermedie per il raggiungimento dell'obiettivo finale. Poiché si tratta in tutto e per tutto di un progetto software, tornano di estrema utilità gli isegnamenti dell'ingegneria del software che invita a considerare le classiche fasi di analisi dei requisiti, progettazione, implementazione e collaudo dell'applicazione (la fase di manutenzione è evidentemente al di fuori della portata del progetto di tesi).

Ecco dunque il processo a cui ci si è affidati per orientarsi nella realizzazione di SPAF:

1. Sviluppare la conoscenza del dominio del problema, identificando i concetti astratti dello stream processing e stabilendo le relazioni (di parentela, di composizione, di utilizzo, etc...) che intercorrono tra questi.

2. Chiarire qual è il risultato finale che si desidera ottenere all'atto pratico, pensado cioè a quelle che saranno le interfacce di utilizzo finale.

    - Definire le API *user-facing*, basate sui concetti identificati al punto 1..
    - Definire le API *provider-facing* (o SPI) e capire come realizzare il *glue code*[2] tra le API e le SPI, ispirandosi a framework di astrazione pre-esistenti (quali, ad esempio JPA, SLF4J).

[2]Il *glue code* è codice che serve solo ad "adattare" parti diverse di codice che altrimenti sarebbero incompatibili. *Glue code - Wikipedia* 2022

3. Per ciascun framework di stream processing (Apache Spark, Apache Storm, Apache Flink, Apache Samza):

   (a) imparare a utilizzare il framework, leggendo la documentazione e analizzando il codice delle applicazioni di esempio;

   (b) inferire quali sono le sezioni principali di un'applicazione scritta su tale framework;

   (c) realizzare un'applicazione base il cui codice renda evidente ed esplicita la struttura dedotta al punto (b);

   (d) implementare il *provider* estendendo l'applicaizone base e utilizzando le SPI e le API pensate al punto 2., cercando dunque di raccordare i concetti del nostro framework con quelli esposti dalle API del framework di stream processing; qualora ciò non risultasse possibile, ritornare al punto 2. (o, se necessario, al punto 1.) per adeguare o migliorare le interfacce.

Il processo appena descritto è trasversale alle fasi di analisi, progettazione e implementazione; si è trovato efficace adottare un processo altamente iterativo e non rigidamente sequenziale.

2.1 Requisiti

I requisiti di SPAF derivano direttamente dai requisiti originali di RAM^{3}S, ricordiamoli: facilitare la creazione di applicazioni di stream processing e consentirne l'esecuzione sui vari framework esistenti.

Questo progetto, per sua natura, non ha richiesto una fase di analisi particolarmente difficoltosa: le informazioni riguardanti il dominio del problema sono risultate abbondanti nel Web e spesso di buona qualità. Diversamente da quanto normalmente avviene per i progetti di applicazioni, qui ci si è potuti largamente ispirare ai sistemi esistenti e di cui di fatto si vuole realizzare un'astrazione (stiamo parlando dei framework di stream processing) e che ciò ha facilitato la fase di analisi. Le difficoltà maggiori si sono incontrate senz'altro nella fase realizzativa (di progettazione e di implementazione).

Sebbene i requisiti generali del progetto fossero manifesti e chiari fin dal principio, ci si è trovati comunque di fronte a requisiti di alto livello e che avrebbero evidentemente

lasciato ampio spazio di interpretazione e libertà decisionale per la loro realizzazione, dunque a un cospicuo sforzo creativo.

Di seguito, dunque, riportiamo l'interpretazione che si è data ai requisiti e le decisioni più importanti che si sono prese nella fase iniziale del progetto.

2.1.1 Facilitazione nella programmazione

Il primo requisito «facilitare la creazione di applicazioni di stream processing» è piuttosto autoesplicativo, se letto da uno sviluppatore: servono delle API semplici e comode da utilizzare. Pertanto, ci si è subito immedesimati nel ruolo del programmatore finale e sulle aspettative di semplicità di utilizzo che costui avrebbe potuto nutrire rispetto all'interfaccia di programmazione.

In estrema sintesi, per definire un'applicazione di stream processing, un programmatore deve:

- specificare da dove provengono i dati in ingresso;
- quale serie di trasformazioni si vuole operare sui dati;
- dove collocare i dati così elaborati.

Si è dunque immaginata un'API che rendesse evidente questa struttura e che esponesse in modo esplicito i concetti cardine dello Stream Processing, ma soprattutto che permettesse di definire l'applicazione scrivendo codice il più vicino possibile a una descrizione in linguaggio naturale (vedi Fluent API) e tramite strumenti linguistici in grado di alleggerire la stesura del codice (quali le *Functional Interface* di Java). Si è cercato, infine, di garantire al programmatore un po' di aiuto da parte dell'IDE, optando per gli strumenti linguistici in grado di garantire la type-safety a tempo di compilazione (vedi i *Generics* di Java).

2.1.2 Indipendenza dai framework

Il secondo requisito richiede la possibilità di eseguire l'applicazione definita con le API di SPAF sui vari framework esistenti in maniera trasparente.

In questo caso l'utente finale a cui dedicare il proprio sforzo creativo non è il programmatore di applicazioni, ma lo sviluppatore che vorrà adoperarsi per rendere disponibile un

certo framework di stream processing all'esecuzione di applicazioni così definite. Anche in questo caso, dunque, è necessario ideare un'interfaccia di programmazione, pensata però per i fornitori di servizi; questo genere di interfaccia viene chiamata in gergo *Service Provider Interface* (o SPI) e fa parte di un pattern di programmazione supportato nativamente da Java (vedi appendice).

L'indipendenza dai framework verrà dunque resa possibile adottando il pattern Java SPI, che consentirà ai vari vendor dei framework di stream processing di creare una libreria per SPAF di tipo "provider", per relizzare lo strato di traduzione dell'applicazione scritta con le API di SPAF alle API della libreria di stream processing proprietaria.

2.1.3 Indipendenza dai connettori

Un requisito meno evidente è quello di garantire anche un strato di astrazione delle sorgenti e delle destinazioni dati, o connettori, usate da un'applicazione SPAF. Dovrà infatti essere possibile integrare in maniera *pluggable* nuovi provider di connettori di tipo arbitrario (code di messaggi, file system, database), lasciando anche in questo caso l'onere della realizzazione delle librerie di raccordo ai programmatori intenzionati a fornire il supporto per un dato vendor, cioè a coloro che implementeranno lo strato SPI.

SPAF dovrà pertanto esporre un duplice strato di astrazione: uno per i framework di stream processing e uno per i supporti di input/output.

L'astrazione dei connettori, in realtà, riguarda anche il programmatore finale e la volontà di RAM^{3}S di semplificargli la vita. Le API di SPAF, anche in questo caso, dovranno sollevare il programmatore dai dettagli implementativi riguardanti l'utilizzo delle librerie di ciascun connettore e consentirgli di specificare sorgenti e destinazioni in modo dichiarativo.

2.1.4 Ipotesi semplificative

Il progetto RAM^{3}S è decisamente ambizioso, non solo perché nel mondo dello Stream Processing ci sono numerosi *player* (i vari framework Flink, Spark, etc.., ma anche framework astratti quali Apache Beam), ma soprattutto perché il dominio applicativo è sufficientemente articolato da richiedere uno sforzo progettuale e implementativo che non è stato possibile esaurire in un solo progetto di tesi. Per queste motivazioni si è ritenuto ragionevole limitare consapevolmente la portata di questa primissima versione di SPAF

a un insieme di funzionalità ristretto. Le limitazioni così imposte, però, non dovrebbero inficiare la possibilità di apprezzare, anche in questa prima versione, l'utilità di un framework di questo tipo.

Introduciamo pertanto un insieme di ipotesi semplificative rispetto all'utilizzo, dunque alla realizzazione, della prima versione di SPAF:

- saranno previste solo sorgenti e destinazioni di tipo code di messaggi, fornite da message broker noti quali Apache Kafka e RabbitMQ;
- gli operatori di trasformazione prevederanno un singolo stream in input e un singolo stream in output; così come le sorgenti prevederanno ciascuna un singolo stream in output, e le destinazioni un singolo stream in input; in sostanza, sarà possibile definire solo topologie di tipo "lineare";
- gli elementi che viaggeranno negli stream saranno esclusivamente coppie *chiave-valore*;
- non sarà previsto il supporto a *store* per la memorizzazione dei risultati intermedi di computazione dei singoli nodi, sarà cioè supportata solo la computazione di tipo *stateless*;
- sarà prevista la realizzazione di sole API di tipo *low-level*, in particolare dell'operatore più semplice concepibile, ossia di una funzione di processamento che, come anticipato, prevederà un unico stream di input e un unico stream di output e dove gli elementi verranno processati a uno a uno e senza possibilità di fare riferimento a elementi processati in precedenza (computazione *stateless*), ma dove la logica di trasformazione dei singoli elementi potrà essere arbitrariamente complessa;
- sarà prevista la possibilità di specificare la sola topologia *logica*, ovvero la definizione del processo di trasformazione da input a output, e non la topologia *fisica*, ovvero la possibilità di definire come distribuire sui nodi fisici i vari elementi computazionali (sorgenti, operatori di trasformazione, destinazioni) definiti nella topologia logica.

Aggirare i limiti imposti

Vediamo ora come alcuni limiti imposti dalle ipotesi semplificative non sia però così invalicabili. Elenchiamo dunque, per taluni limiti, alcuni *workaround* che ci consentono di "spremere" un po' più di potenza espressiva da questa prima versione del framework:

- **limite del singolo stream di output**: il fatto di poter disporre di un unico stream di output in un nodo di trasformazione non limita in realtà la possibilità di fornire in output anche elementi di tipo diverso in contemporanea; se, ad esempio, da un nodo di trasformazione si volessero produrre in output due elementi come risultato del processamento di un singolo elemento in input questo sarebbe realizzabile effettuando un "*multiplexing*" dei due elementi in un'unica tupla "contenitrice", considerando dunque l'output come composto da sole tuple "contenitrici", e prevedendo nel nodo trasformatore a valle un'apposita operazione di "*demultiplexing*";

- **limite di definizione della sola topologia logica**: è possibile fare un passo verso il controllo della distribuzione del calcolo andando a realizzare più applicazioni RAM^{3}S indipendenti e mettendo in connessione tra loro tali applicazioni tramite delle code di messaggi opportunamente predisposte; in tal modo, ogni topologia logica verrebbe mappata su una topologia fisica (definita in modo automatico, e peculiare, dal framework di esecuzione scelto per ciascuna applicazione RAM^{3}S);

- **limite della sola computazione *stateless***: seppure questa prima versione del framework non preveda un supporto formale e integrato di uno store (globale o locale) per la memorizzazione dei risultati intermedi dei singoli nodi di trasformazione, nulla vieta che nella logica di trasformazione definita nei nodi sia previsto l'accesso a risorse di memorizzazione esterne al framework (es. file, DB, cache di oggetti), a patto che queste risultino poi accessibili all'atto dell'esecuzione in contesto di cluster;

- **limite del supporto a connettori del solo tipo "coda di messaggi"**: questo limite può essere facilmente superato prevedendo la realizzazione di strumenti di utilità (ad esempio, a linea di comando) in grado di leggere dalla tipologia di sorgente desiderata (es. file di input) i dati che si intendono processare, e di pubblicarli su una coda di input debitamente predisposta; allo stesso modo, è possibile prevedere

la realizzazione di uno script in grado di consumare i messaggi dalla coda di output e di memorizzarli sulla destinazione desiderata (es. database).

Casi d'uso

Come in un normale progetto di applicazione software, anche in questo caso si presenta la necessità di definire un'interfaccia: non si tratta però di un'interfaccia utente (UI), pensata per essere utilizzata graficamente da utenti comuni, e solitamente rappresentata con la notazione UML degli *use case diagram*; si tratta invece di un'interfaccia di programmazione (API), poiché stiamo realizzando uno strumento rivolto agli utenti programmatori, e che può essere rappresentata con più facilità tramite esempi in pseudo-codice.

Come anticipato, e come presto vedremo nel dettaglio, l'interfaccia di programmazione è però duplice: vi sarà un'interfaccia di programmazione (API, Application Programming Interface) rivolta all'utente programmatore di applicazioni di stream processing, e un'altra interfaccia di programmazione rivolta agli implementatori degli stream processing provider (SPI, Service Provider Interface).

Qui sotto illustriamo il *boilerplate-code* che il programmatore di applicazioni di stream processing dovrebbe scrivere utilizzando le API di SPAF.

```
// 1. configurazione e ottenimento dell'ambiente di esecuzione
Config config = ConfigFactory.load();
Context context = StreamProcessing.createContextFactory().createContext(config);

// 2. definizione dell'input e dell'output
Source<String, String> source = StreamProcessing.createSource(config);
Sink<String, String> sink = StreamProcessing.createSink(config);

// 3. definizione del grafo di processamento (ovvero, della topologia logica)
Topology topology = new Topology()
   .setSource("<source-id>", source)
    // ...
   .addProcessor("<processor-id>", (String key, String value, Collector<String,
      String> collector) -> {
     // Logica di trasformazione
   }, "<predecessor-id>")
    // ...
   .setSink("<sink-id>", sink);

// 4. creazione dell'applicazione
Application application = new Application()
    .withName(config.getString("application.name"))
    .withTopology(topology);
```

```
// 5. lancio dell'applicazione
context.run(application);
```

N.B.: per i dettagli su come viene definito il contenuto delll'oggetto `config`, fare riferimento all'esempio di codice mostrato per l'entità *Context* nella Sezione 2.2.

Per ragioni di trattazione, non è altrettanto conveniente riportare un esempio di utilizzo delle SPI poiché il codice di utilizzo di queste dipende grandemente dal framework di stream processing che si intende fornire come provider di SPAF; si invita pertanto a consultare direttamente il codice sorgente di uno dei provider già implementati.

2.2 Progettazione

La fase di progettazione è molto delicata, le scelte che verranno prese in questa fase avranno delle ripercussioni sull'intero progetto.

Per affrontare più serenamente questa fase, si sono "invocati" i principi dello sviluppo software orientato agli oggetti, tra cui:

- il principio di singola responabilità (SRP): usato in maniera trasversale in tutto il progetto;
- il principio di sostituzione di Liskov (LSP): il cui utilizzo risulta evidente, poiché il progetto realizza proprio uno strato di astrazione verso dei sottotipi, rappresentati dai framework di stream processing, nei confronti dell'applicazione, che deve essere scritta una sola volta;
- il principio di inversione delle dipendenze (DIP): si è infatti cercato di introdurre delle interfacce ogni qualvolta è sorta la necessità di mettere in comunicazione tra loro due moduli del progetto.

Inoltre, ove se ne è riconosciuta l'occasione, ci si è affidati ai design pattern dell'ingegneria del software, in modo da contenere il più possibile la complessità del progetto.

Iniziamo dunque la trattazione della fase di progettazione partendo dei requisiti sopra citati.

Il nuovo framework dovrà:

- Tenere conto della struttura intrinseca di una generica applicazione di stream processing, in termini delle fasi principali di cui questa è composte (es. inizializzazione, definizione, esecuzione, ...); nella documentazione di Apache Flink si parla di "anatomia" di un'applicazione di Stream Processing (*Overview | Apache Flink* 2022), e riproponiamo qui lo stesso termine.

- Identificare i componenti alla base dell'applicazione di stream processing, ossia i concetti principali appartenenti al dominio dello stream processing; quello che in ingegneria del software viene chiamato modello.

- Esporre i concetti e la struttura così identificati in modo indipendente dai framework di stream processing e dai sistemi di input/output dati impiegati; esistono numerosi progetti che realizzano questo intento in altri ambiti della programmazione, quali la gestione della persistenza (vedi JPA), la gestione del *logging* (vedi SLF4J), l'interfacciamento con sistemi di messaggi (vedi JMS); tutte queste librerie realizzano in sostanza delle "facciate" uniformi verso i sistemi sottostanti, implementano cioè il design pattern noto con il nome *façade*.
  N.B.: abbiamo già rilevato che il requisito di indipendenza ha doppia natura, pertanto verranno esposte in maniera separata le *façade* per lo Stream Processing e per i Connettori.

Vediamo nel dettagio ciascuno di questi punti.

Anatomia di un'applicazione di Stream Processing

Prendiamo in prestito ancora dalla documentazione di Apache Flink, che sintetizza in maniera molto chiara la struttura di un'applicazione di stream processing (*Overview | Apache Flink* 2022):

1. Ottenere l'ambiente di esecuzione.

2. Caricare/creare i dati iniziali.

3. Specificare le trasformazioni su questi dati.

4. Specificare dove collocare i risultati di tali computazioni.

5. Scatenare l'esecuzione del programma.

Tale struttura è ricorrente e riesce nell'intento di mettere a fattor comune la struttura delle applicazioni definite anche negli altri framework considerati nel progetto.

Un'osservazione importante rispetto a questa struttura, e che è valida per tutti i framework, è che ciascuno di questi passi è fondamentale per definire un'applicazione di stream processing funzionante. In modo particolare, assume importanza cruciale il passo in cui si specifica dove i risultati della computazione debbono essere collocati (passo n. 4); senza specificare questo punto, i vari framework si rifiutano di mettere in esecuzione l'applicazione.
A tal proposito, la documentazione di Apache Spark è molto esplicita: *«If your application does not have any output operation [...] then nothing will get executed.» Spark Streaming - Spark 3.2.1 Documentation* 2022 e, ancora, *«Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations (similar to actions for RDDs).»* (*Spark Streaming - Spark 3.2.1 Documentation* 2022).

Sink come "messa a terra" dell'applicazione di Stream Processing

Un'analogia che spero risulti calzante è quella della "terra" nei circuiti elettrici/elettronici e della conseguente differenza di potenziale che si instaura nel circuito una volta collegato appunto il terminale di "terra" al supporto fisico che la realizza (es. il terminale negativo dell'alimentatore da banco): senza specificare una destinazione per i dati, che rappresenti una "depressione" in grado di accogliere il flusso di dati generato a monte, non è possibile instaurare una differenza di pressione (di potenziale) che porti all'emergere di un flusso di dati (di corrente) tra la sorgente e questa destinazione.

2.2.1 Modello di un'applicazione di Stream Processing

Questa è senz'altro la parte più delicata e più importante dell'intero progetto.

Come scritto nel libro "Design Patterns" (Gamma et al. 1994b (trad. dell'autore): «La parte difficile nella progettazione object-oriented è la decomposizione del sistema in oggetti».

Sempre seguendo gli insegnamenti del libro "Design Patterns", possiamo ricorrere a più approcci per modellare il sistema: un primo approccio è quello di derivare il modello dalla descrizione a parole del problema, concentrandosi soprattutto sui *nomi* e sui *verbi* che compaiono in questa, perché alcuni di questi rappresentano rispettivamente *entità* e *operazioni*; oltre a ciò, però, è necessario stabilire anche le *relazioni* che intercorrono tra le varie entità così identificate e le *responsabilità* che ciascun elemento del modello deve ricoprire. Non esiste un approccio migliore degli altri e soprattutto nulla vieta di usare più approcci in maniera combinata.

Per la modellazione del nostro sistema si è scelto di utilizzare senza dubbio gli approcci sopra elencati, basandosi soprattutto sulla documentazione disponibile dei vari sistemi e a volte sul codice sorgente, ma si è inoltre preferito integrare anche un approccio basato sull'identificazione dei concetti ricorrenti presenti nella documentazione e nel codice dei vari sistemi e nel metterli a fattor comune; questo ha consentito di confermare l'utilizzo e il significato di alcune entità, di meglio stabilire le relazioni che sussistono tra queste e di attribuire le responsabilità in modo più razionale.

Per fare un esempio di come si è impiegato il processo di modellazione appena descritto, si consideri la necessità di definire in un'applicazione di Stream Processing la sorgente da cui dovranno provenire i dati da elaborare e il dove si dovranno collocare gli esiti delle operazioni su questi eseguite: questo doppio requisito emerge in maniera evidente nella documentazione dei vari framework, come abbiamo già potuto constatare nella descrizione dell'anatomia di un'applicazione di stream processing fornita da Flink; per cui, si sono raccolti i vari *nomi* e *verbi* con i quali ciascuna documentazione fa riferimento a questo doppio requisito. in Flink si parla di `SourceFunction` e `SinkFunction`; in Samza di `InputDescriptor` e `OutputDescriptor`; in Storm è esplicita la modellazione della sorgente dati, che viene chiamata `Spout`, ma non c'è un entità esclusivamente dedicata al concetto di destinazione dei dati, il quale viene invece inteso come una delle varie trasformazioni eseguibili sui dati e dunque modellato con l'entità `Bolt`; analogamente a Storm, anche in Spark Streaming esiste un'entità specifica per le sorgenti dati chiamata `InputDStream`, ma la destinazione dei dati viene intesa anche qui come un'operazione da realizzare sullo stream specificandone la logica nel metodo `foreachRDD`. Per quanto riguarda le responsabilità da attribuire ai concetti di sorgente e destinazione dati, è necessario prendere delle scelte; come si è appena mostrato, i sistemi non sono concordi sul come intendere il con-

cetto di destinazione, ma in realtà non sono concordi nemmeno nello stabilire i compiti che i vari elementi debbono svolgere: alcuni considerano sorgenti e destinazioni come "entità attive" e prevedono un certo stato e delle certe operazioni per questo (vedi `Spout` di Storm), altri le considerano invece come "entità passive" e prevedono esclusivamente uno stato descrittivo (vedi `InputDescritpor` e `OutputDescriptor` di Samza); altri le considerano trasformazioni arbitrarie da specificare sullo stream (vedi `foreachRDD` di Spark); altri ancora, infine, considerano le sorgenti e le destinazioni di dati come operazioni arbitrarie ma reificandole come entità (vedi `SourceFunction` e `SinkFunction` di Flink). Poiché lo scopo di SPAF è sì quello di fornire un'astrazione dei concetti dello stream processing, ma di delegare la loro realizzazione ai framework sottostanti, si è usato come primo criterio di scelta il fatto di preferire la modellazione tramite *entità* invece che tramite *operazioni*; in seconda battua, si è preferita la modellazione con entità di tipo passivo, cioè puramente descrittive, a entità di tipo attivo, sempre per la ragione di delegare l'implementazione ai provider; infine, si è un po' scommesso sul fatto di trattare con la stessa "dignità" le destinazioni dei dati rispetto a quanto avviene per le sorgenti, quindi di intenderle entrambe delle *entità*, con la speranza di facilitare la comprensione del modello al programmatore finale, fornendo una visione più uniforme possibile.

L'atto di distinguere e operare una separazione tra concetti di **definizione** e i concetti di **esecuzione** di una stream processing application è un esercizio che è risultato ricorrente durante tutto il processo di progettazione; infatti, durante la progettazione ci si è concetrati soprattutto sulle API del sistema e sull'assicurarsi che tramite di esse si potesse dare la possibilità al programmatore finale di specificare tutte le informazioni necessarie per descrivere in maniera completa l'applicazione di stream processing. Tali informazioni verranno poi "consegnate" allo strato dei provider che, tramite lo strato SPI del sistema, dovranno a loro volta poter consultare in maniera completa e agevole la descrizione dell'applicazione contenuta in queste e dunque trasferirle alle API del framework di stream processing vero e proprio.

Un esempio che rende evidente di come sia possibile operare una netta distinzione tra concetto descrittivo e concetto implementativo è quella che sarà l'entità principe del sistema, la *Topologia*. Come anticipato, la topologia è definita come un grafo (diretto e aciclico) e alcuni sistemi operano questa distinzione concettuale anche al loro interno: Flink, ad esempio, utilizza il concetto di *Logical Graph* per riferirsi al grafo delle trasfor-

mazioni definito dal programmatore tramite le API; mentre, sempre Flink, usa il concetto di *Physical Graph* per fare riferimento al grafo dei nodi esecutori, cioè le entità che si occuperanno di mettere in atto le trasformazioni definite nel grafo logico (va notato, infatti, che i due grafici non necessariamente coincidono).

In SPAF accade la stessa cosa, ciò che viene definito tramite le API di SPAF ha carattere descrittivo, mentre ciò che viene realizzato implementando le SPI di SPAF ha carattere esecutivo. Poiché SPAF si pone a un livello di astrazione immediatamente soprastante i framework veri e propri, ne consegue che lo strato esecutivo di SPAF consiste nell'interfacciarsi con lo strato descrittivo del framework di stream processing sottostante.

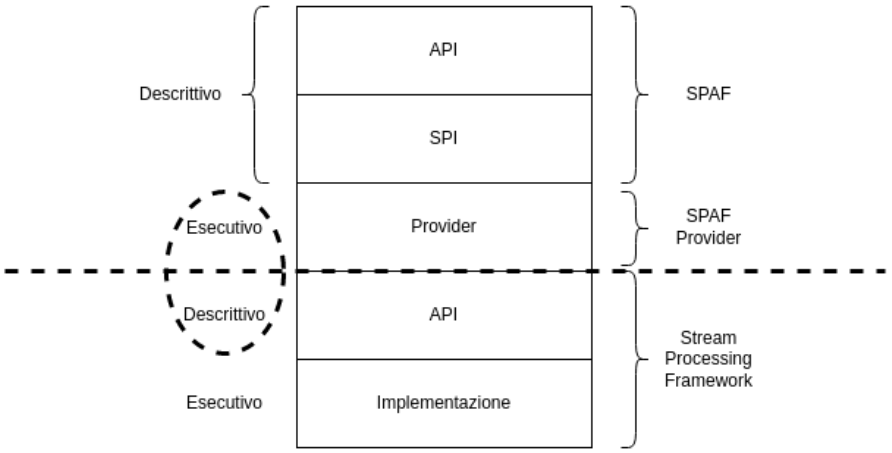

Figura 2.2: Architettura a livelli di SPAF

Idealmente, ciò che è presente nello strato di implementazione non dovrebbe essere mai visibile nello strato descrittivo; a volte, però, poter informare lo strato implementativo con consapevolezza potrebbe portare benefici. Questo è quello che avviene in parte in Apache Storm, dove le API consentono di specificare il grafo logico delle trasformazioni (ossia, la Topologia logica) e al contempo di dare dei "suggerimenti" allo strato implementativo rispetto a come dovrà avvenire l'esecuzione del grafo, in particolare è possibile specificare il grado di parallelismo desiderato per l'esecuzione di ciascun *Bolt*.

Dunque, pontezialmente è possibile dare visibilità lato API di concetti che in realtà apparterebbero esclusivamente al mondo dei provider; questa prima versione di SPAF non fornisce questa possibilità, ma si è pensato a uno sviluppo futuro che consenta al programmatore di informare lo strato provider con aspetti riguardanti la topologia fisica dell'applicazione (vedere la Sezione 4.1.6).

Ritornando alla modellazione del problema dello stream processing, seguendo il processo sopra descritto e applicandolo per tutti i framework considerati e per tutte le tematiche affrontate in questa prima versione del framework, si è giunti a un insieme di entità che costituiranno il modello per questa prima versione del framework.

La tabella sottostante consente una visione globale dei concetti esposti da ciascun framework, sottoforma di classi e di metodi, e di come questi siano stati mappati nel modello finale scelto per il nostro framework.

SPAF		Apache Samza	Apache Flink	Apache Storm	Apache Spark
Context	*local*	`LocalApplicationRunner`	`LocalStreamEnvironment`	`LocalCluster`	`JavaStreamingContext`
	remote	`RemoteApplicationRunner`	`RemoteStreamEnvironment`	`StormSubmitter`	`JavaStreamingContext`
Application		`TaskApplicationDescriptor`	`StreamExecutionEnvironment (superclass)`	`LocalCluster StormSubmitter`	`JavaStreamingContext`
Topology	*logical*	-	`StreamGraph`	`StormTopology`	`DStreamGraph?`
	physical	`StreamTask`	`JobGraph`	`Task`	`Job/Stage/Task?`
Source		`InputDescriptor`	`SourceFunction<T>`	`Spout`	`InputDStream`
Element		`Object`	`Tuple0, …, 25<…> DataStream<T>`	`Tuple`	`DStream<T>`
Processor		`Async]StreamTask interface` **`process()`** `method`	`ProcessFunction abstract class` **`processElement()`** `method`	`Bolt`	`Dstream methods`
Sink		`OutputDescriptor`	`SinkFunction<IN>`	`Bolt`	`DStream::foreachRDD`

Figura 2.3: Tabella di mappatura dei concetti

Tra poco illustreremo nel dettaglio il significato di ciascuna entità identificata nel modello, sul perché si sia deciso di utilizzarla e su quali responsabilità questa dovrà avere; prima di ciò, però, può essere utile avere un'idea generale di come le entità siano in relazione tra loro e che posto ciasuna di queste occupi nella tassonomia del sistema. Per tale scopo, aiutiamoci con il diagramma delle classi illustrato in Figura 2.4.

Per fornire idea di come queste entità interagiscano tra loro per realizzare un'applicazione di streaming, la cosa più efficace è illustrare esempi in pseudo-codice su come sarà previsto il loro utilizzo da parte del programmatore finale. Per ciascun concetto, verrà pertanto mostrato un esempio di pseudo-codice, e in certi casi anche un confronto con il codice che si sarebbe invece dovuto scrivere se si fosse utilizzato direttamente il provider.

Come anticipato, la definizione delle interfacce di programmazione ha seguito un processo iterativo in cui si sono alternati gli approcci *top-down* e *bottom-up*.

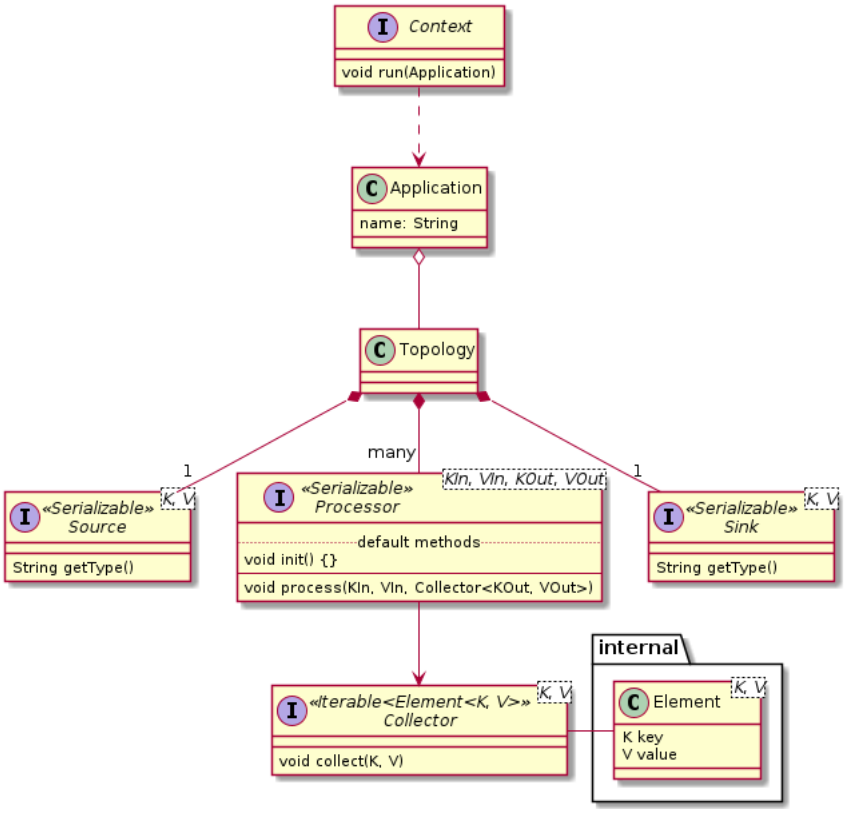

Figura 2.4: Diagramma delle classi dei concetti principali

Triangolazione delle interfacce di programmazione

Un'analogia che potrebbe rendere l'idea del processo che si è adottato per la definizione delle API e delle SPI è pensare a come potrebbe avvenire la triangolazione di un segnale radio: inizialmente si è cercato di determinare alcune direttrici che si potessero considerare ragionevolmente fisse, come ad esempio l'assunto che certe entità dovranno necessariamente essere presenti nelle API (ipotesi supportata dall'ispezione del codice dei framework sottostanti); quindi si è cercato di individuare, in maniera esplorativa e con l'intuito, quelle che sarebbero potute essere le "traiettorie" mancanti da tenere in considerazione (ossia le entità non ancora identificate, di non immediata concezione); si è reiterato il processo fino al raggiungimento di una convergenza definitiva sulle API (e SPI) finali.

Context

L'entità *Context* è senzaltro la prima entità che il programmatore di applicazioni di Stream Processing, cioè colui che utilizzerà le API, dovrà considerare, ma lo stesso vale anche per il programmatore che vuole realizzare un provider di un framework per lo Stream Processing, cioè colui che userà le SPI. Essa infatti rappresenta l'ambiente di esecuzione vero e proprio in cui verrà messa in opera l'applicazione definita in maniera astratta. È dunque un'entità che rappresenta un "ponte" tra il mondo astratto realizzato dal nostro framework e il mondo concreto messo a disposizione dai vari framework di Stream Processing.

Il metodo `run` della classe `Context` rappresenta il "punto d'ingresso" dell'applicazione di Stream Processing, in maniera equivalente a ciò che rappresenta il metodo `main` di una classe Java; tramite il metodo `run`, che accetta come parametro di ingresso una *Application*, si sottopone al provider di Stream Processing sottostante l'applicazione definita in termini delle API del nostro framework; sarà dunque il provider sottostante a "interpretare" l'applicazione così definita e a "tradurla" in una rappresentazione equivalente del motore di Stream Processing messo a disposizione. L'implementazione dell'interfaccia `Context` è dunque totalmente peculiare per ciascun provider, anche se, come vedremo nella Sezione 2.3, è possibile riconoscere uno schema ricorrente nell'implementazione del metodo `run`.

Come anticipato, per ragioni di trattazione si è trovato sconveniente includere esempi di codice per l'implementazione delle SPI. Si invita il lettore a fare direttamente riferimento al codice sorgente dei provider SPAF già implementati in questa prima versione (vedi progetti `spaf-spark-streaming`, `spaf-flink-streaming`, `spaf-storm` e `spaf-samza`).

Ogni framework di Stream Processing è dotato di un ambiente di esecuzione a sé e avente caratteristiche proprie; per questa ragione è stata predisposta la classe `Config` (vedere Gestione delle configurazioni), che consente ai programmatori di specificare le configurazioni del contesto di esecuzione in modo astratto (lato API) e di consultare e mettere in atto tali configurazioni in modo specifico al provider (lato SPI).

Caso d'uso Il codice sottostante mostra come avverrebbero la configurazione e l'ottenimento del contesto di esecuzione in un'applicazione scritta nativamente per Apache Flink.

```
Configuration envConfiguration = new Configuration();
```

```
/* env configs - START */
envConfiguration.set(RestOptions.PORT, webUIPort);
/* env configs - END */

/** Local execution **/
StreamExecutionEnvironment env = StreamExecutionEnvironment
    .createLocalEnvironmentWithWebUI(envConfiguration);
```

In SPAF, vorremmo invece poter specificare le configurazioni del contesto di esecuzione in maniera separata dal codice:

```
context {
   local = true

   flink {
      web-ui = true
      web-ui-port = 8081
   }
}
```

In SPAF, vorremmo inoltre ottenere il contesto di esecuzione in maniera indipendente dal framework sottostante:

```
// 1. configurazione e ottenimento dell'ambiente di esecuzione
Config config = ConfigFactory.load();
Context context = StreamProcessing.createContextFactory().createContext(config);

// ...

// 5. lancio dell'applicazione
context.run(application);
```

Topology

Una topologia definisce la logica computazionale di un'applicazione di stream processing, ovvero, come i dati in ingresso vengono trasformati nei dati in uscita (*Streams Architecture - Confluent Documentation* 2022).
Una topologia è un grafo composto da tre tipologie di nodi principali: nodi sorgente, nodi di processamento e nodi destinazione. Il grafo di una topologia deve essere di tipo diretto e aciclico (DAG):

- diretto: i collegamenti tra i nodi hanno un senso di percorrenza;
- aciclico: i collegamenti devono essere tali da non generare cicli.

Come anticipato nell'analisi dei requisiti, in questa prima versione di SPAF sarà possibile definire esclusivamente topologie di tipo "lineare". In Figura 2.5 i due tipi di tipologia a confronto.

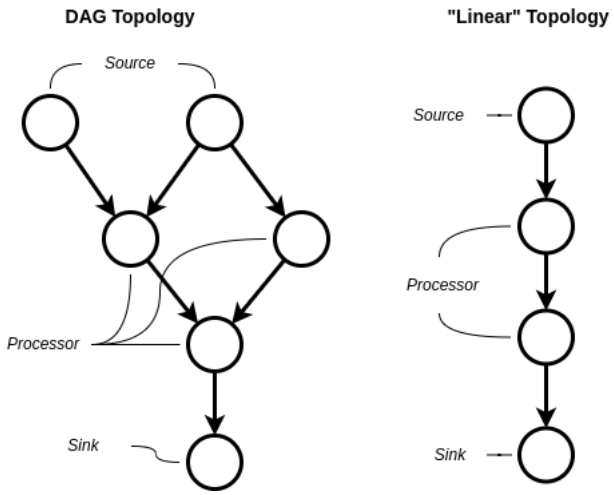

Figura 2.5: DAG vs Linear Topology

Caso d'uso In SPAF, vorremmo poter specificare la logica dei nodi *Processor* tramite lambda-function e disporre di un'API di tipo *fluent*[3] per comporre la topologia.

Ecco, in sostanza, come vorremmo poter specificare una topologia in SPAF:

```
// 3. definizione delle trasformazioni (ovvero, della topologia logica)
Topology topology = new Topology()
   .setSource("Source", source)
   .addProcessor("ToUppercase", (String key, String value, Collector<String,
       String> collector) -> {
      collector.collect(key, value.toUpperCase());
   }, "Source")
   .addProcessor("ToSneakCase", (String key, String value, Collector<String,
       String> collector) -> {
      collector.collect(key, String.join("-", value.split(" ")));
   }, "ToUppercase")
   .setSink("Sink", sink);
```

Per ottenere questo risultato, si è impiegato il design pattern denominato *Builder*: si tratta di un pattern di tipo "creazionale" che consente di costruire oggetti complessi, come una topologia, passo dopo passo (*Builder* 2022). Questo pattern, inoltre, consente di produrre differenti rappresentazioni dell'oggetto costruito utilizzando lo stesso codice

[3]Traducendo dalla pagina di Wikipedia (*Fluent interface - Wikipedia* 2022): «In ingegneria del software, un'interfaccia di tipo *fluent* è un'API orientata agli oggetti la cui progettazione si basa ampiamente sulla concatenazione di metodi»

di fabbricazione; questo aspetto può tornare utile in versioni successive di SPAF, dove potrebbe risultare conveniente cambiare il modo di rappresentare la topologia.

Application

Il concetto di *Applicazione* coincide sostanzialmente con il concetto di *Topologia* appena illustrato, in sostanza aggiunge solo delle informazioni descrittive o di contorno. In ogni caso, si è comunque preferito rendere esplicito il concetto di *Applicazione* per rendere manifesto il modello nella sua totalità e fornire al programmatore alle prime armi un lessico completo.

Concettualmente, un'applicazione di stream processing potrebbe definire anche più di una topologia, anche se tipicamente ne viene definita una sola (*Streams Architecture - Confluent Documentation* 2022). In questa prima versione di SPAF si è deciso di utilizzare una una relazione con cardinalità 1:1 tra *Application* e Topologia.

Caso d'uso Il codice per definire un'applicazione SPAF è molto semplice, ed è evidente come questa non faccia molto più che "impacchettare" una topologia per prepararla alla sua esecuzione:

```
// 4. creazione dell'applicazione
Application application = new Application()
    .withName(config.getString("application.name"))
    .withTopology(topology);
```

Source e Sink

Abbiamo già anticipato in più punti questi due termini, e ormai dovrebbe risultare chiaro che indicano rispettivamente una sorgente di dati e una destinazione per i dati in un'applicazione di Stream Processing. Ora diamo però una definizione più formale, prendendo in prestito la spiegazione fornita da un altro framework di stream processing, la cui integrazione però non è stata prevista in questa prima versione del progetto, Kafka Streams:

Source e Sink possono essere considerati come due nodi processori di tipo "speciale" in una topologia:

- **Source Processor**: Un *source processor* è un tipo speciale di nodo processore che non ha nessun nodo processore a monte. Produce uno stream dati in input alla

topologia, recuperando i dati dal sistema sottostante (es. coda di messaggi, file system, console), e inoltra i singoli elementi dello stream ai nodi processori a valle.

- **Sink Processor**: Un *sink processor* è un tipo speciale di nodo processore che non ha nessun nodo processore a valle. Riceve i singoli elementi dagli stream dati a monte e li colloca nel sistema sottostante (es. coda di messaggi, file system, console).

Cogliere l'astrazione suggerita da Kafka Streams, ovvero pensare a Source, Sink e Processor come entità derivanti tutte da una classe comune che possiamo indicare come *Nodo Processore*, consente al framework di realizzare certe operazioni sulla topologia in maniera più semplice: un esempio, molto importante e che sarà utile in futuro, è l'operazione di "visita" dell'intera topologia definita tramite le API di SPAF, con lo scopo di tradurre ogni singolo *Nodo Processore* di questa in un'entità o in un'operazione equivalente delle API del framework sottostante; maggiori dettagli si trovano nella sezione Occasioni di refactoring.

Meno evidente, ma di fondamentale importanza nella gestione di Source e Sink, è il tema riguardante la serializzazione dei dati in input/output dai sistemi sottostanti. Sappiamo ormai che all'interno della topologia SPAF viaggeranno solamente oggetti rappresentati da una coppia *chiave-valore*, dover però il tipo dato della *chiave* e il tipo dato del *valore* è arbitrariamente specificabile dal programmatore finale. All'atto pratico, dunque, ci si dovrà scontrare con il problema di convertire i dati grezzi provenienti dal sistema dati nei tipi dato specificati nell'applicazione, e gestire anche il problema inverso. Una prima soluzione fornita da SPAF a questo problema verrà illustrata nella sezione Implementazione; la tematica è però sufficientemente complessa da meritare delle ulteriori considerazioni, illustrate in e un'evoluzione di SPAF a riguardo.

Caso d'uso Vediamo ora come intendiamo semplificare e astrarre la definizione di Source e Sink tramite SPAF. Illustriamo prima un esempio di definizione di sorgenti e destinazioni in Apache Spark, in questo caso il sistema dati sottostante è il message broker Apache Kafka.

Ecco il codice di come definire una sorgente dati in Spark, a partire da una *topic* di Apache Kafka:

```
Map<String, Object> kafkaParams = new HashMap<>();
```

```
kafkaParams.put("bootstrap.servers", kafkaBootstrapServers);
kafkaParams.put("key.deserializer", StringDeserializer.class);
kafkaParams.put("value.deserializer", StringDeserializer.class);
kafkaParams.put("group.id", "simple-app-group");

Collection<String> topics = Collections.singletonList("inputTopic");
LocationStrategy locationStrategy = LocationStrategies.PreferConsistent();
ConsumerStrategy<String, String> consumerStrategy =
    ConsumerStrategies.Subscribe(topics, kafkaParams);

JavaInputDStream<ConsumerRecord<String, String>> inputStream =
    KafkaUtils.createDirectStream(
    streamingContext,
    locationStrategy,
    consumerStrategy
);
```

Poiché Spark, come anticipato, non espone il concetto di *Sink*, è necessario realizzare la funzione di *sinking* utilizzando l'operatore di basso livello `foreachRDD`. Qui sotto illustriamo solo una parte del codice necessario per definire una destinazione dati di tipo *topic* di Apache Kafka:

```
Properties kafkaProducerProps = new Properties();
kafkaProducerProps.put("bootstrap.servers", kafkaBootstrapServers);
kafkaProducerProps.put("key.serializer", StringSerializer.class);
kafkaProducerProps.put("value.serializer", StringSerializer.class);

KafkaProducer<String, String> producer = new
    KafkaProducer<>(kafkaProducerProps);
```

Oltre al codice appena mostrato, è necessario prevedere del codice di adattamento per gestire la serializzazione degli oggetti (perché questi verranno distribuiti ai nodi esecutori); è inoltre consigliabile implementare meccanismi affinché tali oggetti, ma soprattutto le connessioni da questi instaurate verso i sistemi dati, non vengano ricreati a ogni iterazione della logica di processamento.

In SPAF, invece, vorremmo gestire la definizione di source e sink, ed evitarci di gestire tutti gli aspetti summenzionati, semplicemente usando un file di configurazione di tipo testuale.

Ecco il contenuto del file di configurazione SPAF che contempla la stessa sorgente e la stessa destinazione dell'esempio precedente:

```
source {
   type = kafka
   zookeeper-connect = "zookeeper:2181"
```

```
   bootstrap-servers = "kafka:9092"
   key-deserializer = org.apache.kafka.common.serialization.StringDeserializer
   value-deserializer = org.apache.kafka.common.serialization.StringDeserializer
   topic = inputTopic
}

sink {
   type = kafka
   bootstrap-servers = "kafka:9092"
   key-serializer = org.apache.kafka.common.serialization.StringSerializer
   value-serializer = org.apache.kafka.common.serialization.StringSerializer
   topic = outputTopic
}
```

Serve inoltre ottenere una rappresentazione a oggetti nel codice dell'applicazione:

```
// 2. definizione dell'input e dell'output
Source<String, String> source = StreamProcessing.createSource(config);
Sink<String, String> sink = StreamProcessing.createSink(config);
```

In questo esempio abbiamo mostrato l'utilizzo di un sistema dati di tipo coda di messaggi. Lo scopo di SPAF è fornire un'astrazione anche al tipo di sistema dati impiegato dall'applicazione, è stata dunque prevista la possibilità di realizzare dei connettori di tipo *pluggable* che consentano lato API di definire nuovi tipi generici di sistemi dati, e lato SPI di integrare in SPAF i sistemi dati veri e propri e corrispondeti ai tipi definiti. L'architettura per la realizzazione di questo requisito è illustrata poco più avanti, nella Sezione 2.2.2.

Processor

Un processore è un nodo nella topologia, come mostrato nel diagramma della Sezione 2.2.1. Rappresenta un passo di elaborazione in una topologia, cioè è usato per trasformare i dati (*Streams Concepts - Confluent Documentation* 2022). Va da sé che questo concetto sia di cruciale importanza in un'applicazione di stream processing: serve a rappresentare la logica vera e propria che si intende eseguire sui dati.

I *processor* possono essere intesi come delle "scatole nere" aventi, in questa prima versione, un unico input e un unico output, e al cui interno avviene la trasformazione dei dati, definibile in maniera arbitraria.

Questa prima versione di SPAF consente di processare, a uno a uno, gli elementi in

ingresso a un processore, ma permette di produrre in output zero, uno o più elementi, calcolati sulla base del singolo elemento in input. Per gestire più agevolmente la variabilità della cardinalità degli elementi in uscita, si è introdotto il concetto di *Collector*, in realtà già adoperato nel codice di svariati framework (ad esempio, da Apache Flink[4]).

L'applicazione di *Face Recognition*, di cui si è effettuato il porting, ha fatto emergere la necessità di eseguire del codice di inizializzazione *una tantum* nei *Processor* di rilevamento dei volti (inizializzazione dell'`HaarDetector`) e di riconoscimento dei volti (inizializzazione e training di `EigenImages`, tramite dataset esterno). Per questa ragione l'interfaccia `Processor` prevede anche un metodo `init`, di cui viene fornita un'implementazione di default vuota, ma che è possibile ridefinire per specificare della logica di inizializzazione arbitraria.

```
@FunctionalInterface
public interface Processor<KIn, VIn, KOut, VOut> extends Serializable {
   default void init() {}

   void process(KIn key, VIn value, Collector<KOut, VOut> collector);
}
```

Disciplina di programmazione dei *Processor*

La possibilità di specificare logica arbitraria sia nel metodo `init` che nel metodo `process` fornisce certamente potere al programmatore, che però deve stare attento a non "cadere in tentazione" in almeno due modi:

- il mantenimento di attributi di classe, eventualmente inizializzati nel metodo `init`, introducono sostanzialmente dello stato nei *Processor*; questo di per sé non è un male, ma ciò non è paragonabile a un supporto completo per la computazione *stateful*, che non è prevista in questa prima versione del framework; attenzione pertanto a non abusare di questa possibilità, in particolare rispetto all'utilizzo della memoria;

[4]Codice sorgente della classe `Collector` nel repository ufficiale di Apache Flink: https://github.com/apache/flink/blob/955e5ff34082ff8a4a46bb74889612235458eb76/flink-core/src/main/java/org/apache/flink/util/Collector.java

- eseguire della logica molto costosa nel metodo `init` potrebbe portare a tempi di avvio anche molto lunghi per l'applicazione di stream processing e aumentare di conseguenza in maniera notevole i tempi di debug, dunque di sviluppo; attenzione pertanto a non specificare logica che impieghi tropppa CPU nella fase di inizializzazione.

Caso d'uso Mischiando l'utilizzo dell'annotazione `@FunctionalInterface`[5] e dei *default method* nelle interfacce Java, è possibile consentire al programmatore finale di definire l'implementazione dei processori in due modi distinti, a seconda delle esigenze e/o preferenze di scrittura:

- se non c'è bisogno di definire della logica per l'inizializzazione del processore, è possibile utilizzare una *lambda function* (che implicitamente definisce una classe anonima Java, che implementa l'interfaccia `Processor`) per definire il solo metodo `process`:

```
.addProcessor(ToSneakCase", (String key, String value,
    Collector<String, String> collector) -> {
    collector.collect(key, String.join("-", value.split(" ")));
}, "ToUppercase")
```

- se invece esiste della logica che ha senso eseguire *una tantum*, prima che il processore inizi a processare i singoli elementi dello stream, allora è conveniente definire una classe Java vera e propria, che implementi esplicitamente l'interfaccia `Processor`:

`process`:

```
public class MyCustomProcessor<KIn, VIn, KOut, VOut> implements
    Processor<KIn, VIn, KOut, VOut> {
    @Override
  public void init() {
       // logica di inizializzazione
  }

    @Override
  public void process(KIn key, VIn value, Collector<KOut, VOut>
      collector) {
     // logica di procesamento
```

[5]Documentazione ufficiale Java dell'annotazione `@FunctionalInterface`: https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html

```
    }
  }
```

N.B.: per rendere disponibili nel metodo `process` le risorse inizializzate nel metodo `init`, è possibile introdurre dei semplici attributi di classe.

L'utilizzo di un processore di questo tipo avviene invece creando un oggetto di tale classe e utilizzando quest'ultimo nella definizione della topologia:

```
.addProcessor("<processor-id>", new MyCustomProcessor(),
    "<predecessor-id>")
```

N.B: in questa modalità, si può apprezzare la compattezza del codice per la definizione della topologia.

Element

Questa entità rappresenta il solo tipo di oggetto in grado di viaggiare negli stream delle topologie definite con SPAF. I vari framework di stream processing sul mercato, invece, forniscono diverse modalità per specificare il tipo di oggetti da processare, ad esempio:

- Apache Storm mette a disposizione il concetto di tupla generica (vedi intefaccia `Tuple` (*Tuple (Storm 2.3.0 API)* 2022));
- Apache Flink prevede una famiglia di classi per rappresentare tuple composte da un certo numero di attributi (vedi classi `TupleXX`, con `XX` da `0` a `25`, nel package `org.apache.flink.api.java.tuple` (*Tuple (Flink : 1.15-SNAPSHOT API)* 2022), oppure tuple come POJO user-defined.

Nel caso di SPAF, è prevista la classe `Element<K, V>` che consente la gestione di sole tuple di tipo chiave-valore con tipi di dato arbitrari. Questa scelta ha evidentemente delle implicazioni pratiche per il programmatore finale, che sarà costretto a specificare sempre un tipo di dato anche per la chiave, ma che potrà al limite valorizzare anche con `null`. A parte ciò, in realtà, la classe `Element` è un po' nascosta al programmatore finale, mentre il suo utilizzo appare evidente nello strato SPI, sia della parte di stream processing che in quella dei connector; a tal proposito vedere le rispettive sotto-sezioni in Implementazione.

2.2.2 Façade per un'applicazione di Stream Processing

La scelta progettuale più importante, e anche la prima in ordine cronologico, è stata quella di identificare nel design pattern Façade lo strumento realizzativo cardine di tutto il framework. Il pattern Façade (in italiano "facciata") è risultato ideale per soddisfare i due requisiti principali del progetto: facilitazione e indipendenza nell'utilizzo di framework per lo stream processing.

Lo scopo del design pattern Façade è forire un'interfaccia unificata ad un insieme di concetti di un sottosistema (Gamma et al. 1994c). Una Façade definisce un'interfaccia di alto livello verso un sottosistema, e lo rende più facile da utlizzare; introduce, inoltre, un disaccoppiamento tra il sottosistema e i suoi clienti, semplificando così il porting di quest'ultimi ad altri sottosistemi.

Nel nostro progetto, i sottosistemi sono i framework di stream processing e i clienti di questi sono le applicazioni di stream processing scritte dai programmatori finali.

A grandi linee, questo pattern prevede uno o più oggetti "facciata", ciascuno dei quali fornisce una singola interfaccia semplificata verso le strutture più generali di un sottosistema. La Figura 2.6 rappresenta graficamente questo concetto.

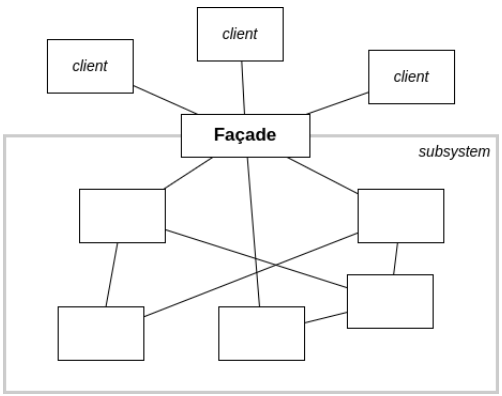

Figura 2.6: Pattern Façade (piccole integrazioni grafiche dell'autore) (Gamma et al. 1994c)

La maggior parte dei *client* di un sottosistema complesso, come un framework per lo stream processing, non ha bisogno di avere a che fare con dettagli di basso livello, come ad esempio il livello di parallelismo di ciascuna operazione della topologia, che nelle API di Apache Storm prende forma con il parametro `parallelism hint`, stante a indicare il numero di thread esecutori per un certo componente (Spout, Bolt). Le interfacce di basso livello sono sì potenti, ma spesso rendono più difficoltosa la creazione dei *client*.
Dunque, un'interfaccia unificata verso le funzionalità di un generico framework di stream

processing rende la vita più semplice alla maggior parte dei programmatori finali, lasciando eventualmente una "porta aperta" all'esposizione di funzionalità di più basso livello nel caso ciò si rendesse necessario (continuando l'esempio del parametro `parallelism hint`, consultare la Sezione 4.1.6 per un'idea su come fornire allo sviluppatore finale la possibilità di accedere a funzionalità di basso livello, riguardanti la topologia fisica).

In questo progetto si è potuta apprezzare la piena potenza del pattern Façade, in particolare la possibilità di usare questo pattern in maniera stratificata. Risulta ormai evidente che un'applicazione di stream processing ha in realtà bisogno di interfacciarsi con due sottosistemi, oltretutto dipendenti tra loro: il framework di stream processing e il sistema, o i sistemi, di input/output dati. Il pattern Façade è risultato adeguato anche per gestire l'astrazione di questa doppia natura delle applicazioni di stream processing, e modellare l'interazione tra i rispettivi sottosistemi. Il risultato finale della progettazione si concretizza perciò nell'applicazione duplice di questo pattern: una Façade per lo Stream Processing e una Façade per i Connettori.

Façade per lo Stream Processing

Per la realizzazione della "facciata" verso i framework di stream processing, si è presa ispirazione da una delle numerose implementazioni del pattern Façade già presenti nel mondo Java. Si propone dunque un parallelismo tra l'astrazione fornita da Java JNDI verso i sistemi di nomi e di directory e l'astrazione che realizzerà SPAF verso i sistemi di stream processing.

Il seguente diagramma mostra l'architettura di JNDI:

A seguire, dunque, l'architettura prevista per SPAF:

Nell'anatomia di un'applicazione di stream processing abbiamo visto quali siano le parti principali che la compongono e abbiamo inoltre identificato il modello dati sottostante. L'astrazione che stiamo realizzando dovrà pertanto tenerne conto di entrambe le cose e offrire delle API e delle SPI, nonché dell'eventuale codice di raccordo, per consentire agli sviluppatori rispettivamente di **descrivere** e di **tradurre** questi aspetti. Enumeriamo dunque le responsabilità principali che saranno affidate alla "facciata" verso un sistema di stream processing:

- Configurazione del framework e dell'applicazione:
  - prevedere lettura delle configurazioni da file di testo.

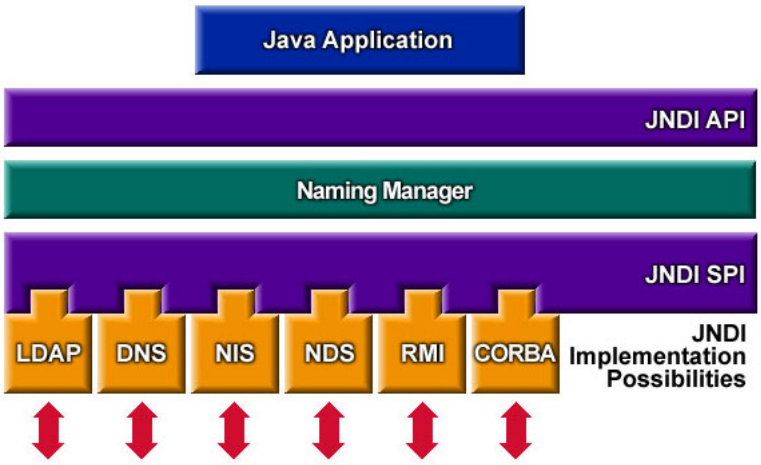

Figura 2.7: Architettura Java JNDI (*JNDI Overview* 2022)

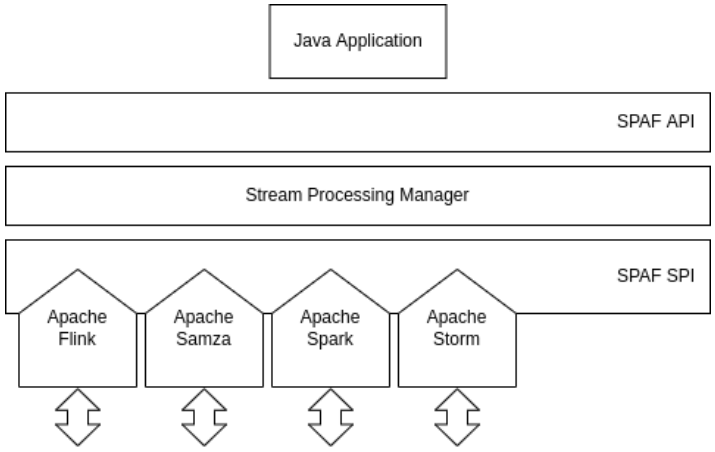

Figura 2.8: Architettura SPAF

- Definizione delle sorgenti e delle destinazioni:
  - consentire la definizione di *Source* e *Sink* usati dall'applicazione tramite file di configurazione.

    N.B.: questa responsabilità, come vedremo a breve, sarà poi delegata alla Façade per i Connettori, trattata nella Sezione 2.2.2.

- Definizione dei tipi di dato per *chiave* e *valore* che "viaggiano" all'interno della topologia:
  - prevedere la *type-safety* a tempo di compilazione, nel codice dei singoli *Processor*.

N.B.: questo aspetto riguarderà anche le *Source* e i *Sink*, verrà dunque considerato anche nella Façade per i Connettori.

- Definizione della Topologia:
  - consentire la definizione del grafo di processamento della topologia, ricordando la limitazione al solo supporto di topologie "lineari";
    N.B.: il supporto a topologie definite come DAG è previsto come sviluppo futuro di SPAF, vedere la sezione Topologie di tipo DAG.
  - prevedere delle API di basso livello per definire la logica computazionale
    N.B.: per una distinzione tra API di basso e alto livello, e considerazioni rispetto all'implementazione di API di alto livello in SPAF, vedere la sezione High-level API.
- Esecuzione dell'applicazione:
  - consentire di sottoporre l'applicazione SPAF al *runtime* del framework sottostante, per causarne l'esecuzione e per gestirne l'eventuale terminazione.

Façade per i Connettori

Questa parte di SPAF consente di astrarre rispetto ai sistemi di input/output dati, o più semplicemente "connettori", impiegabili in un'applicazione di stream processing. Questo risultato si ottiene applicando il pattern SPI Java in maniera doppia, esponendo di fatto due "facciate":

- un'interfaccia API/SPI rivolta al framework SPAF stesso, tramite il quale è possibile specificare nuove tipologie di connettori tramite la loro **descrizione** da rendere disponibili lato programmatore di applicazioni;
- un'interfaccia API/SPI rivolta ai sistemi di input/output dati, i quali potranno definire l'**implementazione** dei connettori specificatamente a un framework di stream processing.

Il diagramma sottostante tenta di cogliere la doppia incarnazione dell'astrazione per i connettori, sfruttando l'analogia della somma di due colori:

- in giallo viene rappresentato il *provider* per il descrittore del connettore SPAF per Apache Kafka
- in azzurro viene rappresentato il *provider* SPAF per Apache Flink
- in verde (somma di giallo e azzurro) viene rappresentato il *provider* per il connettore concreto Apache Kafka per il *provider* SPAF per Apache Flink.

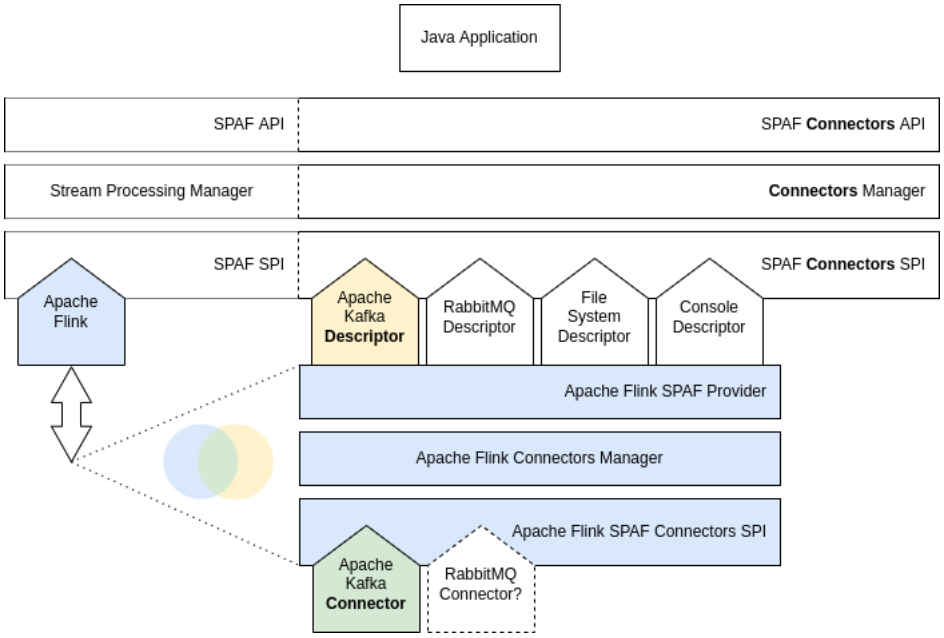

Figura 2.9: Doppio pattern SPI per connettori SPAF

Lo strato rivolto ai provider di connettori concreti richiede di declinare la loro implementazione per ciascun framework di stream processing. Questo perché ogni framework di stream processing, come visto nel modello, tratta a modo suo i concetti di sorgenti e destinazioni dati. Un'implicazione che potrebbe risultare non ovvia di questo design è che, potenzialmente, è necessario realizzare $n \times m$ librerie per fornire un supporto completo a m connettori su n framework di Stream Processing distinti. La soluzione a cui si è pensato rende però completamente indipendente la realizzazione di un provider di stream processing dall'eventuale realizzazione dei corrispondenti provider di connettori concreti (in figura, il bordo tratteggiato del connettore concreto RabbitMQ sta a significarne l'opzionalità). In questo modo è più facile rendere disponibile un nuovo provider di stream processing in SPAF, perché è possibile scegliere in modo completamente arbitrario quali tipi di connettori concreti realizzare. All'atto pratico, inoltre, questa doppia astrazione

consente di spostare la funzionalità di definizione e implementazione dei connettori in librerie separate, che possono essere collegate esplicitamente quando necessario. (*Spark Streaming - Spark 3.2.1 Documentation* 2022)

Un'ultima considerazione è che questo design consente di ottenere la caratteristica di **estensibilità** nel nostro framework SPAF.

Façade per Descrittori di Connettori Per realizzare questa parte di SPAF si è impiegato profusamente il pattern *Abstract Factory*, in entrambe le "facciate". Il pattern *Abstract Factory* di per sé ha lo scopo di fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti tra loro, senza specificare le loro classi concrete, ma può essere usato congiuntamente al pattern *Façade*, per fornire un'interfaccia di creazione per gli oggetti di un sottosistema in maniera indipendente dal sottosistema vero e proprio (Gamma et al. 1994d).

Il diagramma sottostante mostra come si è declinata la struttura di *Abstract Factory* per realizzare la "facciata" dei connettori rivolta verso SPAF.

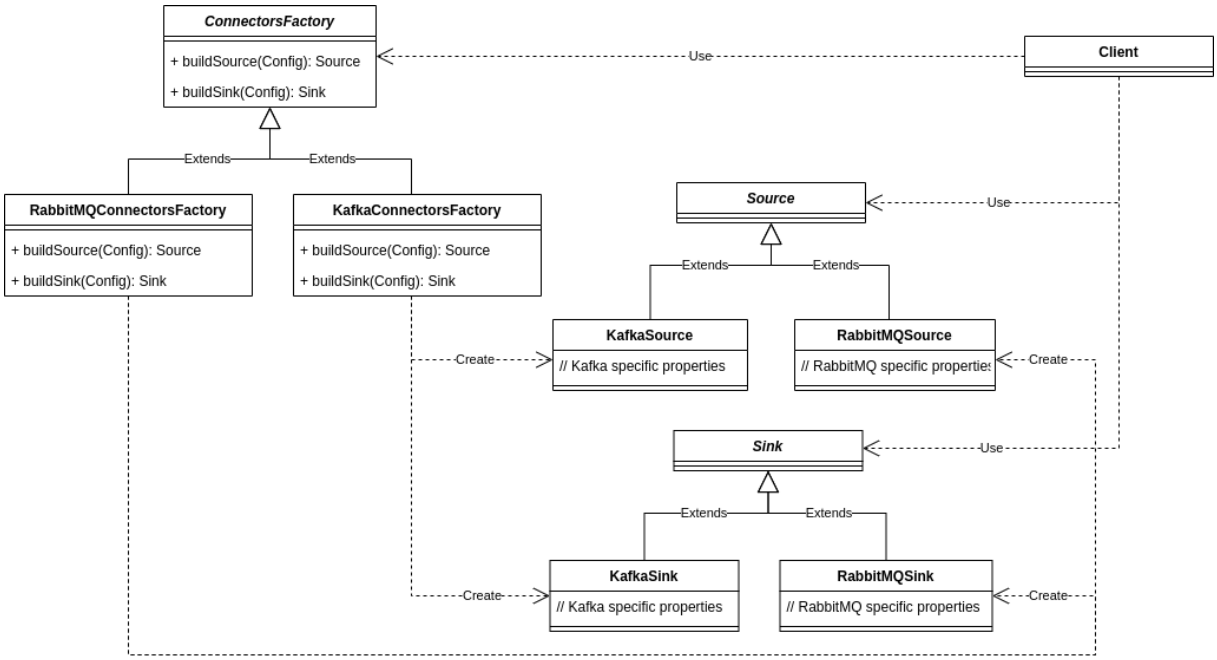

Figura 2.10: Diagramma UML pattern Abstract Factory descrittori di connettori

Questa prima versione del progetto prevede il solo supporto a connettori di tipo coda di messaggi, per tale ragione si sono realizzate le gerarchie di classi per supportare i due message broker Apache Kafka e RabbitMQ, ma l'integrazione di altri tipi di sistemi di

input/output è facilmente realizzabile andando a estendere/implementare le astrazioni `Source` e `Sink`: si può, ad esempio, integrare un nuovo connettore per lettura/scrittura su file system creando una nuova gerarchia di classi del tipo `LocalFileSystemSource`, `LocalFileSystemSink` e `LocalFileSystemConnectorsFactory`.

Façade per Implementazione di Connettori Mentre la "facciata" lato SPAF consente di specificare sorgenti e destinazioni in modo puramente dichiarativo e completamente slegato dalla effettiva realizzazione, la "facciata" lato *provider* di Stream Processing è invece fortemente dipendente dal framework considerato e da eventuali librerie di terze parti esterne a quest ultimo (es. libreria `kafka-clients`[6]). Fare riferimento alla sezione Provider di Connettori, per maggiori dettagli e per un esempio di provider di connettori.

2.3 Implementazione

In questa sezione illustreremo gli aspetti tecnici più rilevanti che sono emersi durante la scrittura del codice di questa prima versione di SPAF. Verrà illustrato come si è calato il pattern Java SPI nel progetto e da quali librerie esistenti si è preso spunto; dunque mostreremo a grandi linee come si implementa un provider per lo stream processing, usando come traccia il provider realizzato per Apache Flink; seguiremo questo stesso processo per illustrare l'implementazione di un provider per descrittori di connettori, quindi di un provider per connettori concreti, questa volta facendoci guidare da quanto realizzato per Apache Samza; mostreremo per completezza come si è messa in piedi la parte per la gestione delle configurazioni definibili dal programmatore finale per il contesto di esecuzione e dell'applicazione; prima di concludere, descriveremo un piccolo ostacolo tecnico che si è incontrato nella realizzazione del provider per Apache Spark, relativamente all'inizializzazione dei *Processor*; concluderemo dunque presentando delle possibili occasioni di refactoring del codice realizzato per questa prima versione di SPAF, che, se colte, ne migliorerebbero la struttura interna e faciliterebbero il lavoro di implementazione dei provider.

[6]Libreria Java ufficiale fornita da Apache Kafka per consentire l'interazione con il *message broker*: https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients

2.3.1 Struttura Pattern SPI

La struttura generale delle classi per il pattern SPI è stata dedotta da quella impiegata dalla libreria open-source SLF4J (Simple Logging Facade for Java), che realizza appunto una "facciata" verso le numerose librerie di logging Java. Tale struttura è stata semplificata e adattata alle esigenze del nostro framework SPAF, in particolare alla necessità di esporre una doppia facciata (una per lo stream processing e una per i connettori).

Per semplificare l'esposizione, tratteremo in questa sezione la parte del pattern SPI rivolta all'utente, con il relativo codice di gestione, e nelle sezioni successive le parti rivolte ai provider (di stream processing e di connettori).

Iniziamo dunque col mostrare, in Figura 2.11, il diagramma UML della parte del pattern SPI rivolta allo sviluppatore di applicazioni SPAF.

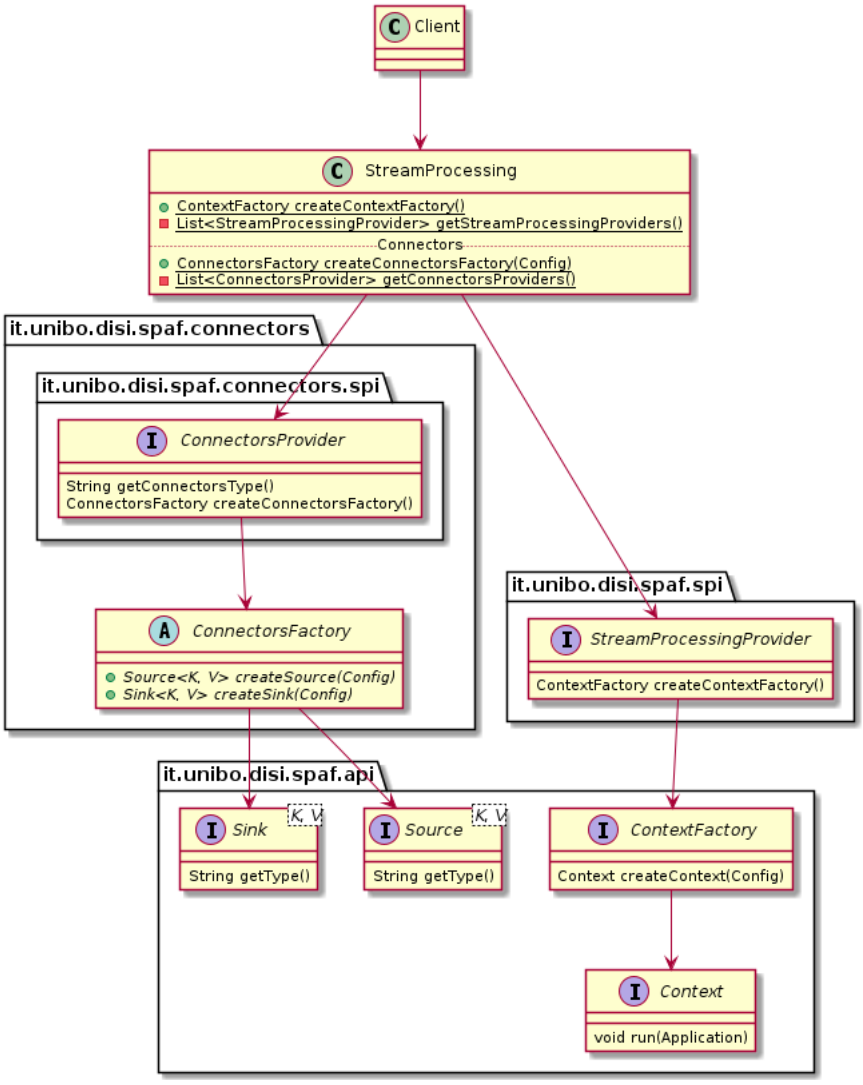

Figura 2.11: Diagramma delle classi pattern SPI (lato pubblico)

Lo sviluppatore di applicazioni è rappresentato dalla classe `Client`; questo può accedere alle funzionalità del framework tramite la classe principale `StreamProcessing`, la quale consente di recuperare le factory per la creazione del contesto di esecuzione e per la creazione dei connettori di sorgenti/destinazioni dati.

Di seguito, lo pseudo codice del metodo `StreamProcessing::createContextFactory`, che consente di ottenere una factory per il contesto:

```
1 recupera la lista dei provider di stream processing disponibili
2 if (non ci sono provider disponibili) then lancia un errore
3 crea la la context factory dal primo provider disponibile
4 if (si ottiene una context factory nulla) then lancia errore
5 return la context factory al client
```

Come si può vedere dal codice, al momento è possibile prevedere un unico provider di stream processing utilizzabile dall'applicazione

La logica del metodo `StreamProcessing::createConnectorsFactory` è leggermente diversa, perché in questo caso è ragionevole prevedere più factory di connettori, ciascuna delle quali è in grado di gestire la creazione di certi tipi di connettori:

```
1 recupera la lista dei provider di connectors disponibili
2 if (non ci sono provider disponibili) then lancia un errore
3 cerca un provider in grado di gestire i connettori specificati in
    Config
4 if (non ci sono provider per il tipo specificato) then lancia un
    errore
5 if (ci sono piu' provider per il tipo specificato) then lancia un
    errore
6 crea la connectors factory dal provider individuato
7 return la connectors factory al client
```

Una volta recuperati un oggetto `Context` e un oggetto `ConnectorsFactory`, il programmatore di applicazioni ha tutto ciò che gli serve per accedere alle API di definizione della topologia e delle sorgenti e destinazioni, nonché alle API per sottoporre l'applicazione all'ambiente di esecuzione.

2.3.2 Provider di Stream Processing

Spostiamo ora il focus sul codice lato provider, partendo dai provider di stream processing. Vediamo dunque quali classi un provider deve implementare per rendere disponibile un framework di stream processing in SPAF. Come anticipato, usiamo come filo conduttore quanto realizzato per rendere disponibile Apache Flink, in modalità streaming.

Anche in questo caso, aiutiamoci con il diagramma UML delle classi, mostrato in Figura 2.12, continuando con l'analogia dei colori introdotta nella Figura 2.9.

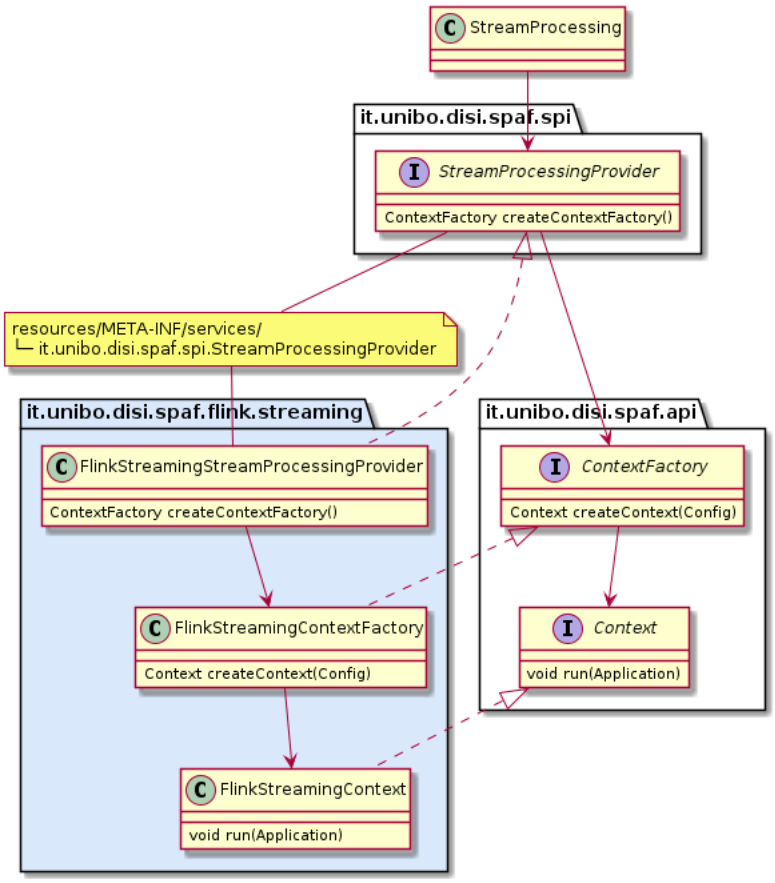

Figura 2.12: Diagramma delle classi del pattern SPI (lato provider)

La realizzazione di un provider con il pattern SPI richiede di definire una classe che implementi l'interfaccia provider, in questo caso `StreamProcessingProvider`, e di specificare un file di testo nel percorso `resources/META-INF/services` avente come nome l'identificativo dell'interfaccia provider, comprensivo del nome del package, e come contenuto l'identificativo della classe provider, anch'esso comprensivo del nome del package. Una volta fatto ciò, la JVM Java è in grado di riconoscere e caricare le classi del nostro provider.

Al netto dei dettagli implementativi del pattern SPI, che sono analoghi per qualsiasi provider, il codice vero e proprio del provider si concentrerà, nel nostro caso, nel metodo `run(Application)` della classe provider dell'interfaccia `Context` (in questo caso `FlinkStreamingContext`).

Vediamo ora qual è la struttura generale del codice da scrivere nel metodo `run`.

Struttura di un provider

La logica di raccordo da scrivere nel metodo `run` è completamente dipendente dal framework di stream processing integrato. Per sviluppare questa logica si è adottato un approccio bottom-up: si è prima creata un'applicazione di esempio nativa alla piattaforme da integrare; dunque si è estrapolato il codice così creato e lo si è reso generico; lo si è quindi inserirlo nel metodo `run`; infine si è realizzato il codice di raccordo tra il modello dati esposto da SPAF e il modello dati peculiare al framework integrato. Si è replicato lo stesso approccio per la realizzazione di tutti e quattro i provider previsti dal progetto.

Ricordiamo che il metodo `run` costituisce sostanzialmente un "ponte" tra l'applicazione SPAF e il framework di stream processing, è qui che avviene il "passaggio di consegne" tra API e SPI. Il metodo `run` riceve come unico parametro di ingresso un oggetto `Application`, che rappresenta l'applicazione SPAF. Dall'oggetto `Application` dovrà dunque essere possibile recuperare tutti gli oggetti definiti dal programmatore dell'applicazione (`Topology`, `Source`, `Sink`, `Processor`, etc...).

Il processo sopra citato ha portato a identificare uno schema ricorrente per la realizzazione del codice del metodo `run`, che può essere sintetizzato come segue:

1. configurazione e ottenimento dell'ambiente di esecuzione;
2. mappatura della *Source* SPAF in oggetto rappresentante uno stream di input nel provider;
3. mappatura della *Topology* SPAF in topologia logica equivalente nel provider;
4. mappatura del *Sink* RAM^{3}S in oggetto, o operazione equivalente, per realizzare uno stream di output nel provider;
5. lancio dell'applicazione e attesa della sua terminazione.

Lo schema appena mostrato ci porta finalmente alla considerazione centrale nella realizzazione di un provider: la topologia definita tramite le API dovrà essere "percorribile" dal codice del provider, per consentire la "mappatura" di ciascuna operazione di trasformazione definita con le API di SPAF, nonché di ciascuna sorgente e destinazione, nella rispettiva rappresentazione delle API del vendor sottostante. La Figura 2.13 da un'idea visiva di questo processo.

Mappatura della topologia L'operazione fondamentale di mappatura della topologia SPAF nella topologia equivalente del framework di stream processing integrato può essere rappresentata con il diagramma in Figura 2.13.

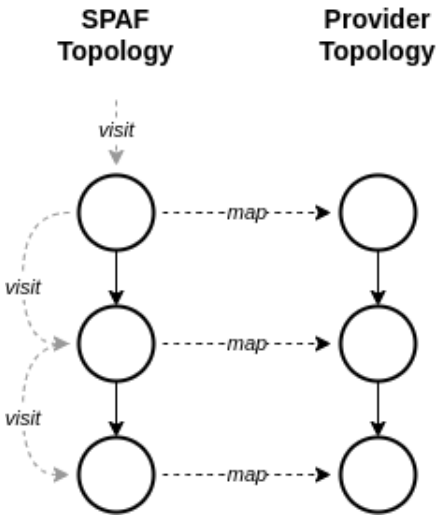

Figura 2.13: Visita della topologia

In sostanza, la topologia SPAF viene visitata nodo per nodo; in base al tipo di nodo incontrato nella topologia SPAF deve essere scelta la strategia di mapping adeguata; tale strategia deve quindi essere applicata per creare il nodo equivalente; i nodi così creati vanno inseriti nella topologia del provider in modo che preservino l'ordine di esecuzione della topologia originale. Poiché questa prima versione del framework prevede solo topologie di tipo "lineare" e con un solo nodo di ingresso e un solo nodo di uscita, l'algoritmo di visita è risultato estremamente semplice da implementare: in sostanza si può implementare con un ciclo `for` che percorra in maniera ordinata i nodi della topologia originale.

In realtà esiste un design pattern appositamente studiato per risolvere questo genere di problemi, si tratta del pattern *Visitor* ed è stato identificato come candidato ideale per applicare un possibile refactoring a questa parte del framework (vedere la Sezione 2.3.5).

Per quanto riguarda l'operazione concreta di *mapping*, va sottolineato che ogni provider richiede del codice a sé. Prendendo come esempio il *mapping* dei nodi *Processor* di SPAF, per il provider che si è realizzato per Apache Spark, si è optato per l'utilizzo del metodo `transform` della classe `DStream` (*Spark Streaming - Spark 3.2.1 Documentation* 2022). Questo metodo consente di specificare trasformazioni di più basso livello possibile in Spark, poiché permette di accedere agli oggetti di classe `RDD`, e in particolare ai relativi metodi di trasformazione. Tra questi ultimi metodi, si è scelto di usare il meto-

do `RDD::flatMap` poiché consente di restituire zero o più elementi per ciascun elemento elaborato, che è esattamente la stessa semantica messa a disposizione da un *Processor* SPAF.

> **Topologia del provider come topologia fisica di SPAF**
>
> La topologia di SPAF definibile dal programmatore di applicazioni è certamente di tipo *logico*. La topologia costruita a run-time dal provider potrebbe essere intesa invece come una topologia SPAF di tipo *fisico*, d'altra parte rappresenta un artefatto eseguibile dal punto di vista di SPAF. È chiaro però che dal punto di vista del provider quest'ultima topologia è invece di tipo logico, poiché il framework di stream processing che dovrà eseguirla creerà a sua volta una rappresentazione di più basso livello, e che considerà finalmente topologia *fisica*. Si instaura dunque un **approccio a livelli**, dove le topologie si "inanellano" tra un livello e l'altro, e hanno in certi casi una doppia natura.

Provider per Apache Flink

Il codice sottostante mostra (a meno di semplificazioni per necessità espositive) l'implementazione del metodo `Context::run` da parte del provider per Apache Flink. Tramite i commenti è possibile identificare lo schema ricorrente sopra citato per la realizzione del metodo `run`.

```
@Override
public void run(Application application) {
    // 1. Configuring and obtaining the execution context
    this.env = FlinkExecutionEnvironmentFactory.build(config);

    // 2. Mapping SPAF Source to Flink Source
    Source<?, ?> source = application.getTopology().getSource();
    DataStream<Element<Object, Object>> stream = /* build from source */;

    // 3. Mapping SPAF Topology to Flink Streaming topology
    for (Processor<> processor : application.getTopology().getProcessors()) {
        // IN and OUT are of type Element<Object, Object>
        ProcessFunction<IN, OUT> processFunction = /* build from processor */;

        stream = stream.process(processFunction);
    }

    // 4. Mapping SPAF Sink to Spark Streaming output
```

```
    Sink<?, ?> sink = application.getTopology().getSink();
    FlinkStreamingConnectors.setupStreamOutput(stream, sink);

    // 5. Application launch
    try {
        this.env.execute(application.getName());
    } catch (Exception e) { /* ... */ }
}
```

Le operazioni di *mapping* vere e proprie avvengono nelle seguenti linee:

- mapping di `Source`:

  ```
  DataStream<Element<Object, Object>> stream = /*build from source */;
  ```

- mapping di `Processor`:

  ```
  ProcessFunction<IN, OUT> processFunction = /*build from processor */;
  ```

- mapping di `Sink`:

  ```
  FlinkStreamingConnectors.setupStreamOutput(stream, sink);
  ```

Vediamo ora, a grandi linee, come è stata realizzata l'operazione di mapping per i `Processor` SPAF.
Per le operazioni di mapping di `Source` e `Sink`, invece, si rimanda alla sezione Connettori Concreti.

Mapping di `Processor` In Figura 2.14 la rappresentazione come diagramma UML dell'operazione concreta di *mapping* di un `Processor` SPAF in una `ProcessFunction` di Apache Flink.

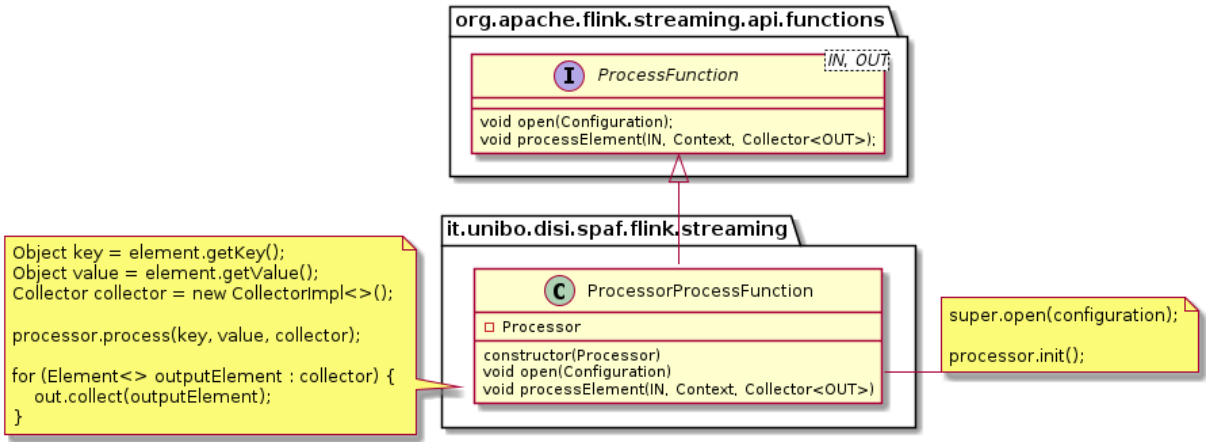

Figura 2.14: Apache Flink provider - Diagramma delle classi per mapping di `Processor`

L'operazione di *mapping* di *Processor* su Apache Flink è risultata particolarmente immediata; le operazioni `init` e `process` dell'interfaccia `Processor` di SPAF trovano infatti una corrispondenza quasi perfetta rispettivamente nelle operazioni `open` e `processElement` dell'interfaccia `ProcessFunction` di Apache Flink. Per amor del vero, però, non si tratta di una coincidenza inaspettata; infatti, come anticipato nella parte di analisi dei requisiti, si è effettuata una profonda ispezione del codice sorgente dei vari framework di stream processing, per cogliere spunti per la realizzazione di SPAF e cercare una convergenza tra le API di SPAF e quelle dei framework integrati.

Nel diagramma della Figura 2.14 è stato riportato anche lo pseudo-codice dei metodi `open` e `processElement` che mette in atto il *mapping* vero e proprio; come si può vedere, il codice è molto semplice.

Utilizzo della classe `Element` Cogliamo l'occasione per illustrare anche come venga utilizzata la classe `Element`, di cui abbiamo anticipato l'esistenza nella sezione di progettazione.

Nel codice del metodo `processElement` risulta finalmente chiaro come la classe `Element` si occupi di "impacchettare" ogni singola tupla che viaggia all'interno di uno stream. La classe `Element` richiede in realtà di specificare due parametri tipo: uno per il tipo di dato rappresentante la chiave, uno per il tipo di dato rappresentante il valore; per ragioni grafiche, si è omesso questo dettaglio nel box dello pseudo-codice, ma indichiamo ora che a livello di provider tali parametri tipo vengono sempre valorizzati entrambi a `Object`. A livello di provider, pertanto, si gestiranno sempre riferimenti a oggetti di tipo `Element<Object, Object>`, questo perché non è stato ancora prevsito un supporto alla *type-safety* a tempo di esecuzione. Realizzare in Java questo aspetto è tutt'altro che scontato: infatti, i *Generics* Java possono aiutarci solamente a tempo di compilazione, poiché il compilatore Java mette in atto la cosiddetta *type erasure*, di cui si riporta la spiegazione fornita nel libro "Java - Tecniche avanzate di programmazione" (H. M. Deitel e P. J. Deitel 2006): «quando una classe generica viene compilata, il compilatore esegue la cancellazione sui parametri tipo della classe e sostituisce ciascuno di essi con il rispettivo limite superiore», e ancora «La visibilità del parametro tipo di una classe generica è la classe stessa.».

I limiti imposti dal funzionamento del compilatore Java possono però essere aggirati, è possibile cioè prevedere il supporto alla *type-safety* anche a tempo di esecuzione; di

questo fatto, ne è la prova Apache Flink che fornisce appunto un supporto completo alla *type-safety*, e la sfrutta non soltanto per facilitare la scrittura di applicazioni, ma anche per ottimizzare le performance a tempo di esecuzione. Per maggiori dettagli a riguardo, vedere la Sezione 3.1.2 del capitolo Considerazioni Avanzate.

Superamento dei problemi su cluster Riportiamo un ultimo dettaglio relativo all'implementazione del provider per Apache Flink.

Ciascun framework di stream processing mette a disposizione due modalità di esecuzione: locale e cluster. Per consentire l'esecuzione di un'applicazione SPAF su un cluster Apache Flink, si è reso necessario effettuare l'override della versione di Kryo usata da Flink (2.24.0) ad una versione immediatamente successiva ($>=$ 3.x e $<$ 5.x), perché nella versione originale era presente un bug che portava alla seguente eccezione nel tentativo di deserializzare l'attributo `value`, di tipo `byte[]`, dell'oggetto `Element<Object, Object>`:

```
java.lang.IllegalAccessError: tried to access field
    com.esotericsoftware.kryo.io.Input.inputStream from class
    org.apache.flink.api.java.typeutils.runtime.NoFetchingInput
```

Grazie alle indicazioni contenute nei commenti a una domanda su StackOverflow (Szczur 2022), si è capito che si trattasse di un problema risolto[7] nella versione di Kryo immediatamente successiva alla 2.24.0 e si è dunque approfondita l'indagine.

Si è dunque appurato che l'eccezione `IllegalAccessError` venisse lanciata perché la classe `NoFetchingInput` di Flink, sotto-classe di `Input` di Kryo, tentava di accedere all'attributo `inputStream` della sua super-classe, anche se questo attributo era dichiarato senza modificatore di accesso e dunque considerato di tipo *package* (o *package-private*), accessibile cioè solo dalle classi dello stesso package della classe `Input`. Il problema alla libreria Kryo è stato sistemato cambiando[8] il modificatore di accesso all'attributo `inputStream` da *package* a *protected*, consentendo così l'accesso all'attributo anche alle sottoclassi come appunto `NoFetchingInput` di Flink.

[7] Voce relativa al problema, contenuta nel sistema di tracciamento di Apache Flink: https://issues.apache.org/jira/browse/FLINK-3291?focusedCommentId=15304234&page=com.atlassian.jira.plugin.system.issuetabpanels%3Acomment-tabpanel#comment-15304234

[8] Commit in cui è avvenuta la risoluzione del problema di Kryo: https://github.com/EsotericSoftware/kryo/commit/642972d532e832aabfe8c9be515b8b2560f093ce

2.3.3 Provider di Connettori

Questa parte, all'atto implementativo, è risultata un progetto nel progetto; illustreremo brevemente il perché.

Come indicato nei requisiti del nuovo RAM^{3}S, e come raccontato nella Sezione 1.1.1, questo progetto doveva prevedere il supporto a entrambi i message broker RabbitMQ e Apache Kafka in ciascuno dei framework di Stream Processing integrati. Seguendo l'intenzione esposta in fase di progettazione di fornire un supporto modulare e *pluggable* delle sorgenti e destinazioni dati, in fase di implementazione ciò si è "incarnato" in tanti sotto-progetti quante sono le combinazioni di message broker e framework di stream processing supportati.

La tabella illustrata in Figura 2.15 enumera tutte le combinazioni possiblii; di fatto, tutte realizzate in questa prima versione di SPAF.

		Stream Processing frameworks			
		Apache Spark	Apache Storm	Apache Flink	Apache Samza
Message Brokers	RabbitMQ	✔	✔	✔	✔
	Apache Kafka	✔	✔	✔	✔

Figura 2.15: Tabella dei provider di connettori concreti

Questa tabella ci aiuta anche a orientarci meglio nella spiegazione delle prossime due sezioni, se la interpretiamo in questo modo:

- le righe rappresentano i provider di descrittori di connettori, di cui parleremo subito dopo questa introduzione;
- le colonne rappresentano i provider di framework di stream processing, di cui abbiamo abbiamo appena finito di parlare;
- gli incroci rappresentano i provider di connettori concreti, che tratteremo subito dopo la sezione sui descrittori.

Descrittori di Connettori

Il diagramma UML mostrato in Figura 2.16 rappresenta le classi principali da implementare, lato provider, per rendere disponibili nuove tipologie di connettori di input/output.

Useremo come traccia il provider realizzato per descrivere sorgenti e destinazioni di tipo *topic* di Apache Kafka (anche qui, continuiamo ad aiutarci con l'analogia dei colori introdotta nella Figura 2.9).

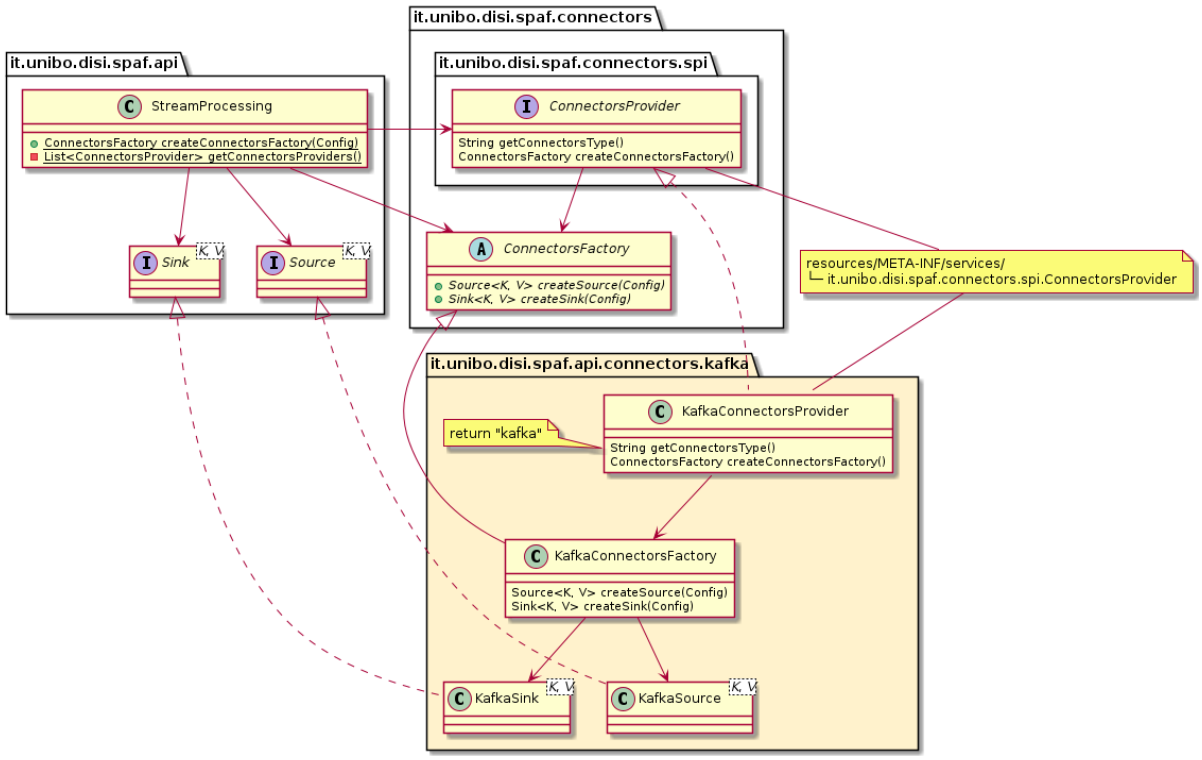

Figura 2.16: Diagramma delle classi di un provider di descrittori di connettori

Il pattern *Abstract Factory* (Gamma et al. 1994d) ha trovato largo impiego in questa gerarchia di classi. La vera modularità del framework rispetto ai tipi di connettori si ottiene però tramite la restituzione della factory concreta (in questo esempio, inclusa nel package `it.unibo.spaf.api.connectors.kafka`) in base a quanto specificato nel file di configurazione relativamente al tipo di `Source`/`Sink` che l'applicazione vuole usare.

Per aiutarci nella compresione di come agisce questo meccanismo, prendiamo come esempio la seguente configurazione di una `Source`:

```
source {
   type = kafka
   zookeeper-connect = "zookeeper:2181"
   bootstrap-servers = "kafka:9092"
   key-deserializer = org.apache.kafka.common.serialization.StringDeserializer
   value-deserializer = org.apache.kafka.common.serialization.StringDeserializer
   topic = inputTopic
}
```

Il metodo `createConnectorsFactory`, della classe principale `StreamProcessing`, accetta in input l'oggetto `Config` contenente tutte le informazioni specificate nel file di configurazione; è proprio questo metodo che individua, se presente, il `ConnectorsProvider` necessario alla costruzione del `Source` o del `Sink` dichiarato nel file di configurazione: la risoluzione avviene usando come chiave il parametro `type` specificato nel file, e cercando corrispondenza con il valore restituito dal metodo `getConnectorsType()` di ciascun provider. Una volta individuato il provider, la classe `StreamProcessing` può recuperare la classe factory e dunque istanziare gli oggetti descrittori che implementano le interfacce `Source` e `Sink`.

Ciascun provider di descrittori di connettori ha il compito di gestire la costruzione dei corrispettivi oggetti concreti `Source` e `Sink`. Nel caso del provider per Apache Kafka, come mostrato in Figura 2.17, le classi concrete si chiamano `KafkaSource` e `KafkaSink` e contengono, ciascuna, le informazioni di configurazioni tipiche che il provider dovrà inizializzare.

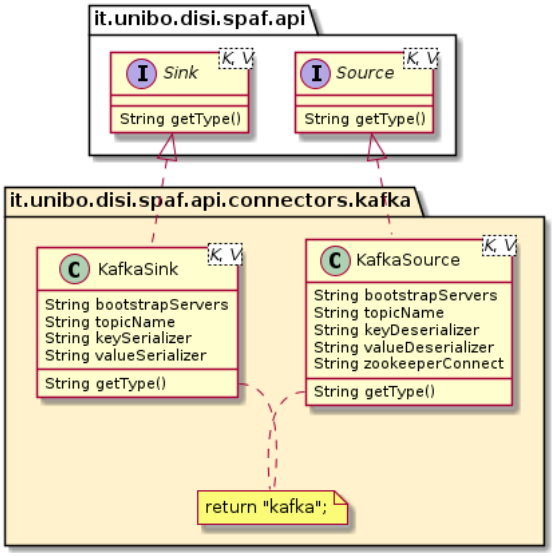

Figura 2.17: Descrittori per Kafak Source e Sink

Connettori Concreti

Arriviamo finalmente alla parte di codice che si occupa di effettuare il *mapping* vero e proprio tra `Source` e `Sink` definite in SPAF, cioè dei rispettivi descrittori di connettori, e gli oggetti/operazioni di input/output messe a disposizione dal framework di stream processing integrato.

Anche in questo caso, facciamoci aiutare da un esempio concreto. Continueremo dunque l'esposizione di quanto realizzato per intergrare Apache Kafka nel provider di Apache Flink.

Cerchiamo di spiegare il diagramma UML mostrato in Figura 2.18 in base ai compiti di ciascun moduilo: il provider di connettori concreti (package verde) riceve in ingresso i descrittori dei connettori specificati nell'applicazione SPAF (package giallo), li traduce in sorgenti e destinazioni native al framework di stream processing preso in cosiderazione (package bianco) e li rende disponibili al provider di stream processing (package azzurro).

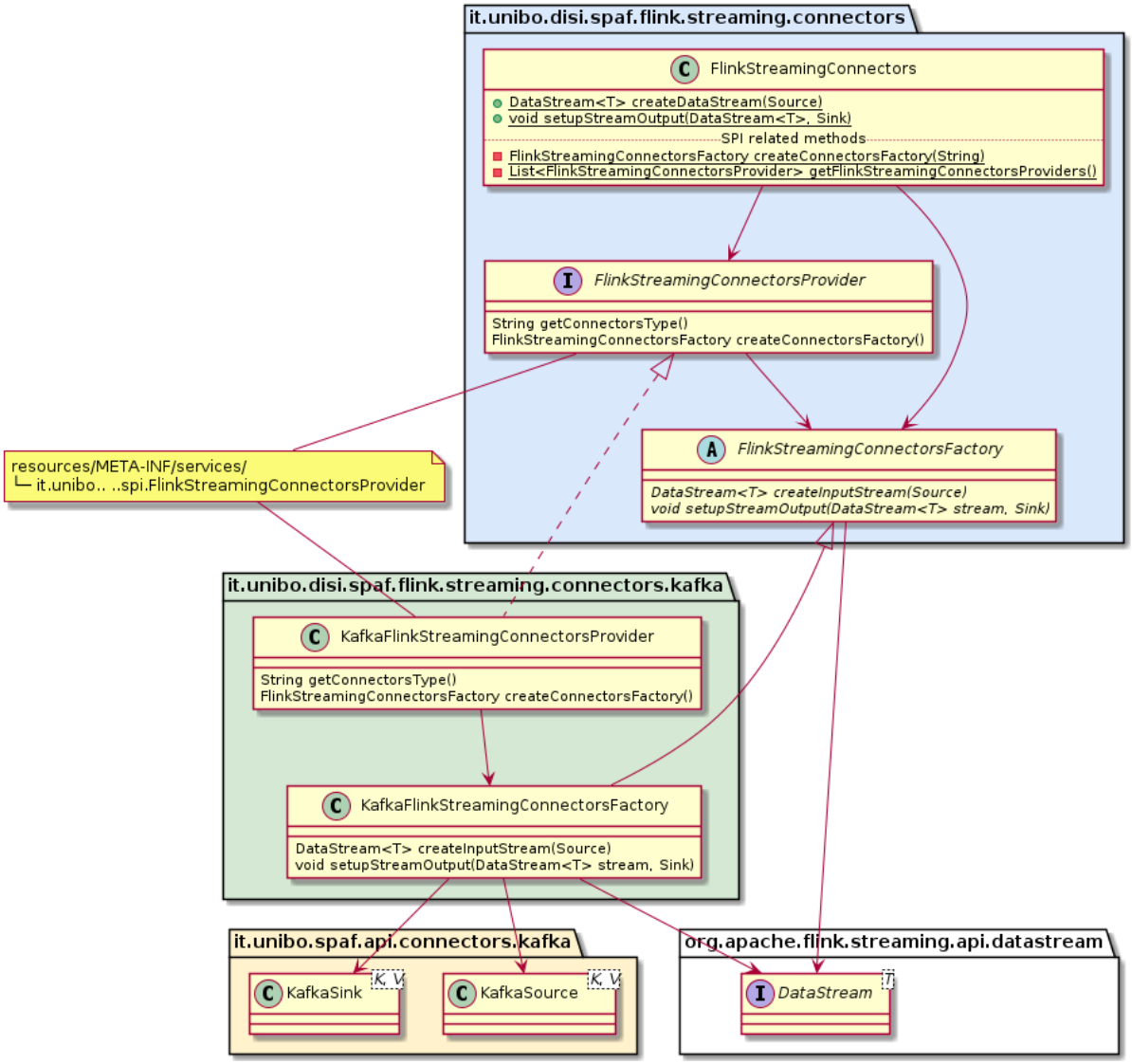

Figura 2.18: Provider di connettori concreti per Apache Flink

Per dare un'idea della natura del codice da scrivere per realizzare il *mapping* vero e proprio, prendiamo in esame l'implementazione del metodo `createInputStream` della

classe `KafkaFlinkStreamingConnectorsFactory`; ricordiamo che lo scopo, in questo caso, è tradurre un'oggetto `KafkaSource` di SPAF in un oggetto `DataStream` di Apache Flink.

```
@Override
public DataStream<Element<Object, Object>> createInputStream(
    KafkaSource<?, ?> source
) {
    KafkaRecordDeserializationSchema<> deserializer = buildDeserializer(
        source.getTopicName(),
        source.getKeyDeserializer(),
        source.getValueDeserializer()
    );

    // org.apache.flink.connector.kafka.source.KafkaSource
    KafkaSource<Element<Object, Object>> kafkaSource = KafkaSource
        .<Element<Object, Object>>builder()
        .setBootstrapServers(source.getBootstrapServers())
        .setTopics(source.getTopicName())
        .setDeserializer(deserializer)
        .build();

    DataStreamSource<Element<Object, Object>> sourceStream =
        environment.fromSource(
        kafkaSource,
        "kafka-source"
    );

    return sourceStream;
}
```

N.B.: anche in questo caso sono state apportate delle semplificazioni al codice, per favorirne l'esposizione.

L'oggetto `DataStream` restituito dal metodo `createInputStream` verrà dunque usato nel codice del metodo `run` del provider di stream processing; su questo oggetto verranno quindi mappate, a una a una, le operazioni di trasformazione definite tramite i *Processor* SPAF. Abbiamo dunque svelato ciò che avviene al posto del commento di codice `/* build from source */` mostrato nel corpo del metodo `run`.

L'analisi del codice del metodo `setupStreamOutput` viene lasciata al lettore[9].

[9]Fare riferimento al codice della classe `KafkaFlinkStreamingConnectorsFactory` del pacchetto `spark-flink-streaming-connectors-kafka`.

2.3.4 Gestione delle configurazioni

Per gestire agevolmente le varie configurazioni di un'applicazione SPAF, sia lato utente che lato codice di supporto, si è scelto di integrare una libreria di terze parti che fornisse un supporto *type-safe*. L'alternativa sarebbe stata quella di usare la modalità "standard" Java, che prevede l'utilizzo di file `.properties` e della classe `Properties`, rinunciando però alla *type-safety* (o dovendo implementare noi tale aspetto, che però è al di fuori dell'intento principale del progetto).

La libreria che si è scelto si chiama "Config", un progetto Open-Source reso disponibile su GitHub[10] sotto licenza Apache 2.0, sviluppato della compagnia Lightbend, e pubblicato sul repository *Maven Central*.

Oltre al vantaggio della *type-safety*, la libreria consente anche di definire il contenuto del file di configurazione in maniera innestata, potendo così organizzare le configurazioni in base al concetto di appartenenza (ovvero, *Context*, *Sink*, *Source*, etc...).

Abbiamo già visto, nelle sezioni precedenti, il formato delle configurazioni di Source, Sink e Context. L'ultima parte dell'applicazione che possiamo configurare tramite file è l'applicazione stessa, cioè possiamo definire un'insieme di parametri personalizzati che potranno essere utilizzati modificare il suo comportamento senza dover mettere mano al codice. Con ogni probabilità, i parametri di configurazione personalizzato definiti per l'applicazione verranno utlizzati all'interno dei *Processor*. Vediamo un esempio di parametro personalizzato, introdotto per l'applicazione di *face recognition* (vedere progetto `spaf-esamples-face`) e che consente di indicare il percorso su file-system dove è collocato il dataset per il training della libreria di riconoscimento volti.

```
...

application {
    name = "Face Detection"
    dataset-path = "/tmp/training-faces/"
}

...
```

Il formato del file appena illustrato, e supportato dalla libreria *Config*, si chiama HOCON e sta per *Human-Optimized Config Object Notation*. Il formato HOCON, in sostanza, è un *superset* di JSON; in aggiunta a quest ultimo, consente infatti di: specficare

[10]Repository del codice della libreria Config: https://github.com/lightbend/config

commenti (con `#` o `//`); omettere le graffe `{}` dell'oggetto radice; utilizzare `=` in maniera equivalente a `:`; e tutto un altro insieme di accorgimenti che semplificano la scrittura e la lettura di un file JSON da parte dei programmatori.

Ecco, per completezza, come viene utilizzato il parametro sopra specificato nel *Processor* che effettua il riconoscimento dei volti:

```
public class FaceRecognitionProcessor implements Processor<...> {
   private final Config config;

   public FaceRecognitionProcessor(Config config) {
      super();
      this.config = config;
   }

   @Override
   public void init() {
      VFSGroupDataset<FImage> dataset;
      try {
         String path = config.getString("application.dataset-path");

         dataset = new VFSGroupDataset<FImage>(path, ...);
      } catch (FileSystemException e) { /* ... */ }
        ...
```

Per quanto riguarda le configurazioni del contesto di esecuzione, cioè del framework di stream processing sottostante, sarebbe possibile pensare a un insieme di configurazioni comuni a tutti i provider, in modo da renderne ancor meno difficoltoso l'utilizzo. Un esempio di configurazione fattorizzabile è `local`, che rappresenta un booleano per indicare la volontà di eseguire l'applicazione sulla JVM locale. L'insieme di configurazioni comuni a tutti i provider così individuate si potrebbero dunque porre, per convenzione, direttamente sotto la chiave `context`, senza dunque specificare anche il suffisso relativo al provider utilizzato.

La classe `OneTimeInitProcessor`

L'esecuzione della logica di inizializzazione una tantum dei *Processor*, in certi framework, non è risultata affatto banale.

Nel provider realizzato per Apache Spark, ad esempio, si è dovuto tenere conto che l'inizializzazione dei *Processor* non potesse avvenire direttamente nel "driver program", cioè nel processo Java principale che di fatto esegue il `main` della nostra applicazione, perché un *Processor* così inizializzato avrebbe potuto contenere riferimenti ad oggetti non

serializzabili. La necessità di serializzare gli oggetti deriva dal fatto che il "driver program" di Apache Spark si occupa di distribuire il calcolo ai cosiddetti "worker", i quali non eseguono necessariamente sulla stessa JVM; per assolvere a questo compito, pertanto, ha bisogno di trasmettere gli oggetti Java coinvolti, che debbono quindi risultare serializzabili.

La casistica appena descritta si è verificata per il *Processor* di riconoscimento volti, il quale istanzia oggetti di classe `EingenImages` durante la fase di inizializzazione, e questi stessi oggetti non sono serializzabili poiché la classe `EingenImages` non implementa l'interfaccia `Serializable`.

Oltre all'aspetto della serializzazione, nel provider di Apache Spark si è anche dovuto ovviare a una presunta assenza di un API che consentisse di eseguire del codice arbitrario prima di mettere in esecuzione nel "worker" l'operazione di trasformazione definita con `RDD::flatMap` (non si è riuscita a individuare nella documentazione e nel codice un'API per questo scopo). Di conseguenza, un "worker" esegue indistinamente il codice di trasformazione definito in `RDD::flatMap` a ogni iterazione.

Per superare questi due problemi, si è deciso di spostare l'inizializzazione dei *Processor* lato "worker" e fare in modo che questa avvenisse una volta sola, alla prima iterazione della logica specificata in `RDD::flatMap`, cioè alla primissima esecuzione del *Processor*. Per fare ciò, si è pensato di effettuare il *wrapping* dell'oggetto di tipo `Processor`, definito dal programmatore finale, in un oggetto di classe `OneTimeInitProcessor`, appositamente sviluppata per risolvere il problema dell'assenza di un API per l'inizializzazione.

Ecco il codice della classe `OneTimeInitProcessor`:

```
public class OneTimeInitProcessor<...> implements Processor<...> {

   private static final long serialVersionUID = 1L;

   private boolean alreadyInitialized = false;
   private final Processor<KIn, VIn, KOut, VOut> processor;

   public OneTimeInitProcessor(Processor<KIn, VIn, KOut, VOut> processor) {
      super();
      this.processor = processor;
   }

   @Override
   public void init() {
      if (!this.alreadyInitialized) {
         this.processor.init();

         this.alreadyInitialized = true;
```

```
      }
   }

   @Override
   public void process(...) {
      this.processor.process(...);
   }
}
```

Di seguito, dunque, il codice interessato del provider per Apache Spark:

```
stream = stream.transform(rdd -> { // codice eseguito nel "driver"
    return rdd.flatMap(inputElement -> { // codice eseguito nei "worker"
        // recupero del Processor, trasmesso tramite serializzazione
        OneTimeInitProcessor<...> processor = processorWrapperVar.value();

        processor.init();

        Object key = inputElement.getKey();
        Object value = inputElement.getValue();
        Collector<Object, Object> collector = new CollectorImpl<>();

        processor.process(key, value, collector);

        return collector.iterator();
    });
});
```

La classe `OneTimeInitProcessor` appartiene al package `it.unibo.disi.spaf.internals` e viene pertanto messa a disposizione, dal pacchetto "core" di SPAF, a tutti i provider che ne avessero bisogno.

2.3.5 Occasioni di refactoring

L'obiettivo di questa prima versione di SPAF era quello di fornire un "motore" a RAM^{3}S che sostituisse in maniera equivalente quello precedente. Un requisito che ha condizionato il tempo a disposizione per la progettazione, e per la scrittura del codice, è stato quello di realizzare il supporto a tutti i framework di stream processing supportati dalla versione precedente di RAM^{3}S. Questa prima versione di SPAF ha raggiunto questo obiettivo, ma presenta ampio spazio di miglioramento in alcune sue parti. Di seguito, dunque, illustriamo alcune occasioni di reingegnerizzazione del codice che si sono presentate durante la fase di implementazione, ma che sono maturate nella testa dello sviluppatore di SPAF solo in un secondo momento.

Gli interventi di refactoring più significativi, e a maggior valore aggiunto, riguarderebbero senz'altro la rappresentazione a oggetti del concetto di *Topologia* e l'insieme di operazioni necessarie alla sua conversione nelle topologie equivalenti dei framework di stream processing veri e propri.

Topologia e DAG corrispondente

Il primo intervento di reignegnerizzazione che proponiamo è la realizzazione di una struttura a oggetti per la topologia che sia in grado di trattare ogni singolo nodo tramite un'interfaccia comune, sia questo una *Source*, una *Sink* o un *Processor*. In sostanza, sarebbe opportuno realizzare una rappresentazione del DAG corrispondente alla topologia adottando un solo tipo di oggetto come nodo del grafo. Questa idea è, tra l'altro, già realizzata dalla libreria di stream processing di Apache Kafka, Kafka Streams, non considerata in questo progetto.

Realizzando un'entità astratta che rappresenti un nodo generico della topologia, e facendo in modo che *Source*, *Sink* e *Processor* vengano considerati dalla topologia come tale entità, sarebbe possibile utilizzare il polimorfismo e impiegare algoritmi molto semplici ed eleganti per la "visita" del DAG. In Figura 2.19 una rappresentazione di questa astrazione.

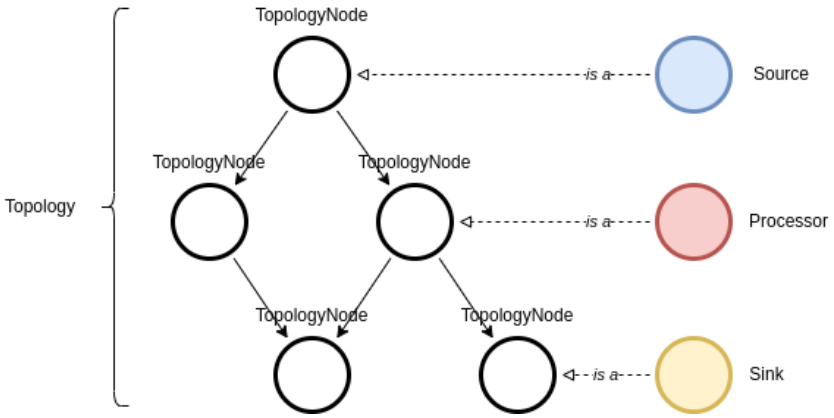

Figura 2.19: Topologia come struttura di nodi

Un nodo generico della topologia potrebbe essere rappresentato dalla classe astratta `TopologyNode`, la quale potrebbe mantenere un riferimento ai nodi immediatamente successivi a questa (anche detti "successori"), ed eventualmente anche ai nodi precedenti (o "predecessori"). Nella Figura 2.20 è rappresentata un ipotesi di quella che potrebbe essere la nuova gerarchia delle classi riguardanti la topologia.

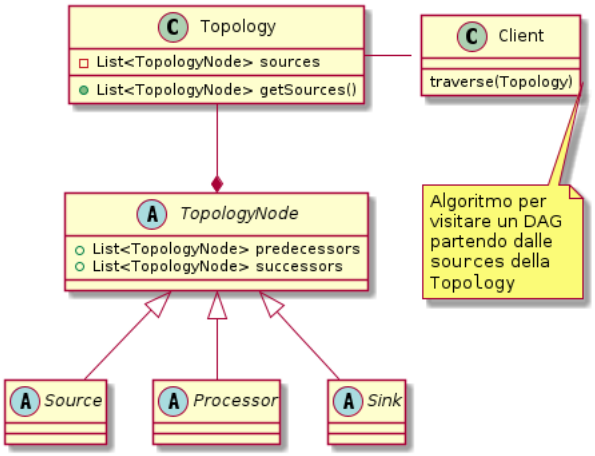

Figura 2.20: `TopologyNode`

Il diagramma mostra come una classe cliente potrebbe fornire un metodo per la visita completa della topologia, tale metodo dovrebbe implementare un algoritmo per la visita di un DAG. Un tale algoritmo sarebbe relativamente semplice da ideare, anche se, di fatto, il problema che stiamo tentando di risolvere è già noto in letteratura come il problema del *Topological sorting* (*Topological sorting - Wikipedia* 2022), per il quale esistono già degli algoritmi a tempo lineare (ad esempio, il Depth-first search (*Topological sorting - Wikipedia* 2022). Come ormai risulterà chiaro, l'operazione di visita della topologia è proprio quella che consentirebbe di eseguire il compito di conversione messo in atto da SPAF.

Questo intervento di reingegnerizzazione preparerebbe il terreno a un altro intervento di refactoring, che riguarda il codice che effettua il *mapping* di una topologia SPAF nella topologia equivalente di un framework di stream processing vero e proprio, e che ora illustreremo.

Topologia e pattern *Visitor*

Nella proposta di refactoring precedente abbiamo discusso di come migliorare la rappresentazione a oggetti della topologia e di come centralizzare la logica per la visita della topologia così definita in un ipotetico metodo `traverse(Topology)`, che implementerebbe l'algoritmo di visita.

Ora discutiamo invece di quello che vorremmo ottenere in seguito alla visita completa della topologia: vorremmo ottenere una rappresentazione equivalente della topologia in termini del framework di stream processing utilizzato. Possiamo fare un'analogia con i

compilatori di linguaggi di programmazione: se pensiamo che il DAG della topologia sia l'analogo dell'*Abstract Syntax Tree* di un programma scritto in un certo linguaggio, allora l'operazione di *mapping* della topologia sarebbe l'analogo della compilazione dell'AST in linguaggio macchina. Vogliamo dunque "compilare" una topologia SPAF in un insieme di operazioni di più basso livello, definite dal framework sottostante. La compilazione è solo una delle possibili operazioni eseguibili su un AST, allo stesso modo il *mapping* della topologia non è potenzialmente l'unica operazione eseguibile su una topologia; in generale ci riferisce all'esecuzione di queste operazioni con il termine **valutazione** di una struttura a oggetti (AST, DAG, o di altro tipo).

Quello che esporremo ora, dunque, è una proposta per migliorare il codice per la valutazione della topologia; in particolare, per l'operazione di *mapping*.

Un design pattern che è stato ideato esattamente per risolvere questo tipo di problema è il pattern *Visitor*. Anticipiamo subito che, nonostante il nome potrebbe suggerirlo, questo pattern non ha la responsabilità di visitare egli stesso la struttura a oggetti sul quale è pensato per lavorare. Se pensiamo a una topologia come a un "museo", questo pattern non realizza dunque anche il concetto di "guida turistica", ma solo quello del "visitatore" e, soprattutto, delle reazioni che costui potrebbe avere di fronte a ciascuna opera d'arte (ossia, le valutazioni). La "guida turistica" per il nostro "visitatore" verrebbe invece realizzata dal metodo `traverse` precedentemente ipotizzato.

Chiariamo ora in maniera formale quale sia lo scopo del pattern *Visitor*, aiutandoci con la definizione fornita nel libro "Design Patterns" (Gamma et al. 1994e) (trad. e grassetti dell'autore):

> [...] rappresentare un'**operazione** da eseguire sugli **elementi** di una **struttura** a oggetti.

Se caliamo questa definizione sul nostro problema, otteniamo questa definizione: «rappresentare il l'operazione di **mappatura** da eseguire sui **nodi** della **topologia**.»

Il pattern *Visitor* ci consente di raggiungere questo scopo senza cambiare le classi degli elementi su cui opera, idealmente senza mai modificare il codice di *Source*, *Sink* e *Processor*. Questo pattern consente, cioè, di specificare nuove operazioni da eseguire sulla topologia in maniera separata le une dalle altre, e mantenere l'indipendenza della struttura della topologia stessa da tali operazioni. Nel nostro caso, al momento, abbiamo necessità di definire la sola operazione di mappatura della topologia, ma in futuro

potremmo prevederne delle altre: ad esempio, la stampa a video della struttura della topologia per scopi di documentazione. In realtà, l'operazione di mappatura non è unica ma dovrà essercene una per ciascun framework di stream processing integrato, anche se per il momento non è prevista la convivenza di due operazioni di *mapping* per la stessa applicazione.

Possiamo ottenere la separazione tra le varie operazioni da eseguire sulla topologia, e al contempo garantire l'indipendenza della struttura della topologia da tali operazioni, raggruppando la logica delle operazioni in oggetti distinti, chiamati **visitatori**, e passando questi oggetti ai **nodi** della nostra **topologia**.

Il diagramma UML mostrato in Figura 2.21 mostra come si potrebbe utilizzare il pattern *Visitor* in SPAF.

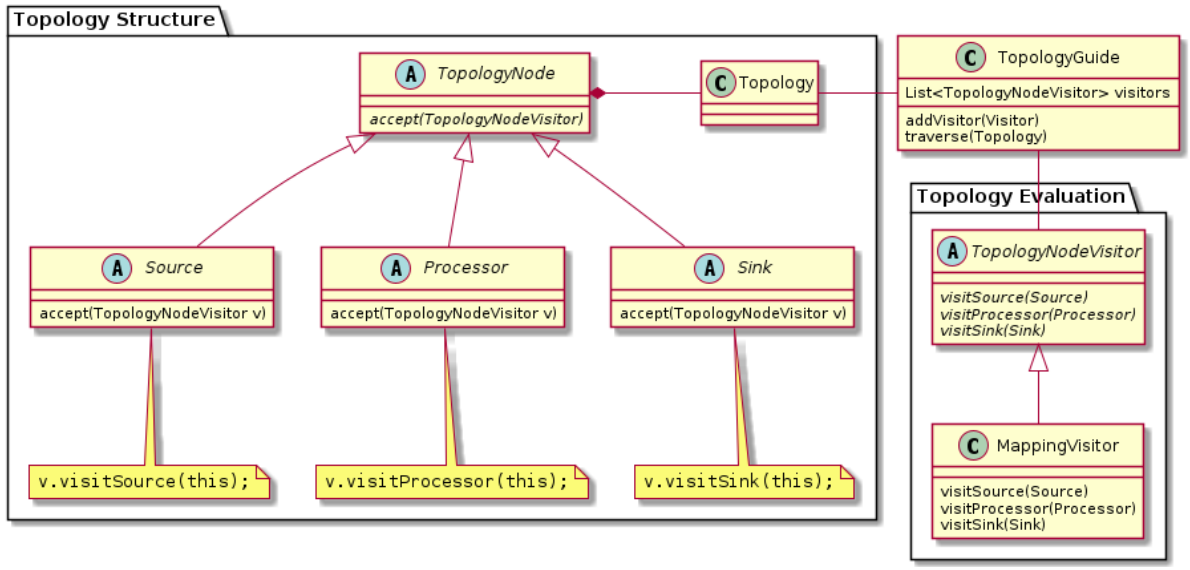

Figura 2.21: Pattern Visitor per la valutazione della topologia

Dal diagramma UML è possibile notare come, con il pattern *Visitor*, si definiscono due gerarchie di classi: una per la struttura degli oggetti da valutare e una per le operazioni di valutazione della struttura.design-pattterns-visitor

La responsabilità di visitare la topologia è stata delegata alla classe `TopologyGuide` (il cui nome è stato scelto in base all'analogia del museo appena proposta). Questa classe, prima di iniziare a visitare la topologia, dovrà essere caricata con gli oggetti visitatori che si vogliono eseguire, utilizzando il metodo `addVisitor`; per avviare la valutazione di una topologia è dunque necessario invocare il metodo `traverse(Topology)` passando la topologia stessa come argomento; l'algoritmo di visita implementato nel metodo `traverse`

consentirà di disporre, a ogni passo, del nodo della topologia attualmente visitabile e di passare a questo l'insieme di visitatori precedentemente specificato; ciò avviene tramite il metodo `accept(TopologyNodeVisitor)` della classe `TopologyNode`, o tramite la sua implementazione concreta definita nella classe del nodo attualmente visitato che si occuperà di invocare il giusto metodo di visita nell'oggetto visitatore, passando il nodo stesso come argomento.

La gestione dell'esito della visita completa di una topologia è completamente a carico dell'oggetto visitatore, che potrà accumulare uno stato mano a mano che visita ciascun nodo della topologia. Nel caso di SPAF, sarà necessario definire un visitatore per ciascun framework di stream processing integrato; tale visitatore dovrà memorizzare al suo interno la struttura a oggetti della topologia "fisica" del framework corrispondente alla topologia SPAF visitata; sarà infine necessario prevedere anche dei metodi per estrarre tale topologia "fisica" dal nodo visitatore e consegnarla finalmente al framework, per metterla in esecuzione.

2.4 Collaudo

Siamo finalmente arrivati alla fase del collaudo del nostro framework di astrazione per lo stream processing.

In generale, la fase di collaudo consiste nel sottoporre l'applicazione all'utente finale, per verificare se quanto realizzato soddisfa le sue aspettative in termini di funzionalità e performance. Il collaudo di SPAF non fa eccezzione, però va tenuto conto delle particolarità del nostro progetto: non è rivolto a un utente "normale", ma a un utente programmatore; non c'è un'interfaccia grafica con cui interagire, bensì un'interfaccia di programmazione da utilizzare; le performance da valutare riguardano tanto l'esecuzione di un'applicazione SPAF su un certo framework, quanto il beneficio in termini di ore/uomo risparmiate per diventare operativi nella creazione di un'applicazione SPAF invece di un'applicazione "nativa", nonché la ridotta complessità e la minor lunghezza del codice da scrivere in SPAF rispetto alla realizzazione di una stessa applicazione direttamente su uno dei framework integrati.

Raccontiamo dunque questi aspetti del collaudo, usando come traccia la realizzazione dell'applicazione di esempio per il riconoscimento dei volti.

2.4.1 Sviluppo di un'applicazione

Per creare un'applicazione in SPAF è necessario seguire un insieme di passi che è perlopiù indipendente dalla logica specifica dell'applicazione creata. Nel nostro caso, la logica dell'applicazione che andremo a realizzare era già definita nel progetto pre-esistente che impiegava il vecchio RAM^{3}S. Il codice specifico di tale applicazione sarà facilmente distinguibile dal codice che realizza, invece, i passaggi fondamentali per la creazione di una qualsiasi applicazione SPAF.

L'applicazione SPAF che realizzeremo sarà eseguibile su uno qualsiasi dei quattro framework di stream processing, e su entrambi i message broker, integrati in questa prima versione del progetto; nel nostro esempio, faremo riferimento all'utilizzo di Apache Flink (Streaming) come provider di stream processing e ad Apache Kafka come provider di connettori.

Ecco dunque i passi per creare un'applicazione SPAF per il riconoscimento dei volti:

1. Creare un nuovo progetto Maven: `spaf-examples-face`

2. Aggiungere le seguenti dipendenze:

   - pacchetto contenente le API di SPAF:

     ```
     <dependency>
         <groupId>it.unibo.disi</groupId>
         <artifactId>spaf-api</artifactId>
         <version>${spaf-api.version}</version>
     </dependency>
     ```

   - pacchetto contenente il provider SPAF per Apache Flink (Streaming):

     ```
     <dependency>
         <groupId>it.unibo.disi</groupId>
         <artifactId>spaf-flink-streaming</artifactId>
         <version>${spaf-flink-streaming.version}</version>
     </dependency>
     ```

   - pacchetti per consentire il debug sulla macchina locale (opzionali):

     ```
     <!-- Apache Flink (for development) -->
     <dependency>
         <groupId>org.apache.flink</groupId>
         <artifactId>flink-streaming-java_${scala.version}</artifactId>
         <version>${flink.version}</version>
     </dependency>
     ```

```
<!-- The flink-clients dependency is only necessary to invoke the
    Flink program locally. -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_${scala.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
<!-- This depdendency is only necessary for local development,
    opt-out in release profile -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-runtime-web_${scala.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
```

- pacchetto contenente il provider per connettori SPAF ad Apache Kafka, relativamente ad Apache Flink (Streaming):

```
<dependency>
    <groupId>it.unibo.disi</groupId>
    <artifactId>spaf-flink-streaming-connectors-kafka</artifactId>
    <version>
        ${spaf-flink-streaming-connectors-kafka.version}
    </version>
</dependency>
```

3. Creare il package principale dell'applicazione:

   `it.unibo.disi.spaf.examples.face`

4. Creare il file di configurazione per l'applicazione:

   `src/main/resources/application.conf`

```
context {
    flink {
       local = true
       web-ui = true
       web-ui-port = 10081
    }
  }

  application {
    name = "Face Recognition"
      dataset-path = "/tmp/training-faces/"
  }

  source {
    type = kafka
    bootstrap-servers = "kafka:9092"
```

```
    topic = FACES
  }

  sink {
    type = kafka
    bootstrap-servers = "kafka:9092"
    topic = RECOGNIZED_FACES
  }
```

5. Creare la classe principale dell'applicazione (es. `FaceRecognition`) e aggiungere il *boilerplate code* al metodo `main`:

```
// 1. configurazione e ottenimento dell'ambiente di esecuzione
Config config = ConfigFactory.load();
Context context =
    StreamProcessing.createContextFactory().createContext(config);

// 2. definizione dell'input e dell'output
Source<String, String> source = StreamProcessing.createSource(config);
Sink<String, String> sink = StreamProcessing.createSink(config);

// 3. definizione delle trasformazioni (ovvero, della topologia logica)
Topology topology = new Topology()
    .setSource("Source", source)
    .addProcessor(/* processor-id */, /* processor-impl */, /*
        predecessor-id */)
    .setSink("Sink", sink);

// 4. creazione dell'applicazione
Application application = new Application()
    .withName(config.getString("application.name"))
    .withTopology(topology);

// 5. lancio dell'applicazione
context.run(application);
```

6. Definire la logica specifica all'applicazione che si sta creando (per ragioni espositive, si riporta solo l'utilizzo dei vari *Processor* realizzati per l'applicazione di riconoscimento volti; per il codice completo fare riferimento al pacchetto `spaf-examples-face`)

```
// 3. definizione delle trasformazioni (ovvero, della topologia logica)
Topology topology = new Topology()
    .setSource("Source", source)
    .addProcessor("FaceDetector", new FaceDetectionProcessor(), "Source")
    .addProcessor("FaceRecognizer", new
        FaceRecognitionProcessor(config), "FaceDetector")
    .addProcessor("FaceMarker", new PersonFaceMarkingProcessor(),
        "FaceRecognizer")
```

```
    .setSink("Sink", sink);
```

Come si può facilmente dedurre dal codice, questa applicazione è composta da tre fasi principali:

1. rilevamento dei volti nell'immagine;
2. riconoscimento dei volti rilevati nell'immagine;
3. marcatura condizionale dei volti nell'immagine.

Ciascuna di queste fasi è stata realizzata con un *Processor* dedicato.

A questo punto è possibile avviare l'applicazione con l'IDE, eventualmente utilizzando la modalità di *debug*.
N.B.: poiché l'applicazione prevede la connessione a un broker Apache Kafka, questo dev'essere già avviato[11].

Una volta avviata l'applicazione, questa rimarrà in attesa di messaggi in arrivo sulla *topic* Kafka specificata come *Source*. L'applicazione si aspetta di ricevere in input dei messaggi aventi il seguente formato:

- chiave: valorizzata con il nome del file immagine;
- valore: array di byte rappresentante l'immagine vera e propria.

Lo stesso formato viene impiegato dall'applicazione per produrre i messaggi in output, dove evidentemente l'immagine restituita sarà stata modificata rispetto a quella originale.

Per facilitare il collaudo dell'applicazione, la vecchia versione di RAM^{3}S prevedeva due piccoli progetti accessori che consentissero di produrre e consumare, su code di tipo Kafka o RabbitMQ, dei messaggi del formato sopra specificato. Il codice di tali progetti è stato rifattorizzato e reso più generico in modo che si adattasse alle caratteristiche del nuovo framework.

Per prima cosa, è conveniente avviare il "consumatore" di messaggi, per il quale è necessario specificare il percorso di una cartella dove collocare le immagini elaborate:

[11] Istruzioni ufficiali su come installare e avviare Apache Kafka: https://kafka.apache.org/quickstart

```
$ java -jar kafka-image-receiver.jar <bootstrap server> <images dir> <topic
    name>
```

Arriviamo dunque al passo finale, cioè l'avvio del "produttore" di messaggi che metterà all'opera la nostra applicazione di riconoscimento volti; per questo progetto è necessario specificare il percorso di una cartella contenente le immagini da inviare:

```
$ java -jar kafka-image-sender.jar <bootstrap server> <images dir> <topic name>
```

Se tutto va bene, sarà possibile osservare nei log dell'applicazione l'attività svolta dai *Processor* in risposta a ciascun messaggio, e si inizieranno a materializzare le immagini risultanti nella cartella di destinazione specificata.

Se si desiderasse eseguire la nostra applicazione su un altro framework di stream processing, sarebbe sufficiente cambiare le dipendenze specificate nel file `pom.xml` del progetto, puntando al provider di stream processing desiderato (e, di conseguenza, a quello dei connettori), e definire nel file `application.conf` le configurazioni peculiare del framework scelto. Dunque, avviare nuovamente l'applicazione.

2.4.2 Esecuzione dell'applicazione su cluster

Durante lo sviluppo di un'applicazione SPAF abbiamo senz'altro necessità di metterla in esecuzione, ma sempre in ambiente locale (nel nostro IDE) e con scopi perlopiù di debug. Al termine dello sviluppo della nostra applicazione, possiamo comunque optare per la sua esecuzione in modalità locale; questa modalità di esecuzione ha perfettamente senso nel caso in cui la quantità di dati da elaborare fosse ridotta. Certo è che, nella modalità locale, non possiamo apprezzare il vero motivo per cui sono nati i framework di stream processing, cioè non tanto quello di offrire un paradigma di programmazione basato sul concetto di stream e di tipo reattivo[12], ma quanto quello di facilitare grandemente la distribuzione dell'elaborazione dei dati su più nodi di calcolo, in modo da gestire grosse quantità di dati, in arrivo nel sistema come un flusso continuo.

Vediamo perciò ora come possiamo mettere in esecuzione un'applicazione SPAF su un cluster di nodi, ottenendo così la distribuzione del calcolo. A seconda del framework di stream processing scelto per la nostra applicazione, la procedura di deployment su cluster

[12]Esistono framework e librerie appositamente dedicate alla programmazione di tipo reattivo (nel concetrato), tra cui RxJava: https://github.com/ReactiveX/RxJava

cambia leggermente. Inoltre, è necessario predisporre manualmente il cluster secondo le modalità specificate da ciascun framework. Un possibile miglioramento potrebbe essere quello di prevedere in RAM^{3}S la possibilità di semplificare questi compiti, fornendo delle procedure semplificate per mettere in piedi un cluster del framework desiderato e per effettuare in questo il deployment dell'applicazione.

In questa sezione vedremo come effettuare manualmente entrambe le operazioni, prendendo come riferimento il framework Apache Flink ma concentrandoci prevalentemente sugli aspetti legati a SPAF.

Per prima cosa, è necessario predisporre un cluster di Apache Flink. È possibile realizzare un cluster di Apache Flink in più modi, tra cui la modalità manuale, tramite l'utilizzo di Kubernetes oppure utilizzando Apache YARN. Vedremo, in estrema sintesi, come realizzare manualmente un cluster di Apache Flink[13]:

1. Predisporre un insieme di macchine aventi accesso SSH senza password, e aventi la stessa struttura della directory. In questo modo sarà possibile eseguire gli script in maniera indipendente da ciascun nodo.

2. Scaricare la distribuzione binaria di Apache Flink[14], decomprimere il pacchetto e collocare la directory così ottenuta nello stesso percorso del file system in tutti i nodi del cluster.

3. Scegliere la macchina che dovrà svolgere il ruolo di nodo coordinatore (detta *JobManager*) e specificare l'indirizzo IP di questa macchina nel file `conf/flink-conf.yaml`, di ciascun nodo del cluster, tramite il parametro `jobmanager.rpc.address`.

4. Specificare ora gli IP (o gli hostname) di tutti i nodi esecutori (detti *TaskManager*) nel file `conf/slaves`, in ogni nodo del cluster.

5. Avviare il cluster eseguendo lo script del file `bin/start-cluster.sh` dal nodo coordinatore.

[13]Guida *quick start* ufficiale di Apache Flink: https://nightlies.apache.org/flink/flink-docs-release-1.1/quickstart/setup_quickstart.html

[14]Indirizzo ufficiale per scaricare la distribuzione binaria di Apache Flink: https://flink.apache.org/downloads.html

A questo punto dobbiamo "impacchettare" la nostra applicazione SPAF, in modo da poterla sottoporre al cluster. Questo si traduce nel realizzare un pacchetto *jar* dell'applicazione, contenente anche tutte le librerie di terze parti sulle quali si basa il codice dell'applicazione. Un pacchetto *jar* di questo tipo viene chiamato *uber*-jar (dal tedesco *über*, «sùper»). È importante però escludere da tale pacchetto le librerie corrispondenti al framework di stream processing vero e proprio, poiché il codice di queste sarà ovviamente disponibile nell'ambiente di esecuzione realizzato da ciascun nodo del cluster.

Per ottenere un *uber*-jar di questo tipo, si può impiegare il plugin Maven denominato `maven-shade-plugin`; ricordarsi di escludere dal pacchetto finale i jar relativi al framework di stream processing, questo si può ottenere specificando le relative dipendenze con `scope` di tipo `provided`. Per un esempio completo di file `pom.xml` in grado di produrre un pacchetto di questo tipo, fare riferimento al codice contenuto nel progetto `spaf-examples`.

Una volta predisposto il progetto in questo modo, è possibile produrre il pacchetto uber-jar con il seguente comando `mvn clean package`.

Al termine dell'esecuzione del comando, dovremmo trovare il pacchetto dell'applicazione all'interno della directory `target` del progetto; attenzione a considerare il file giusto, verranno infatti creati più file con estensione `.jar` e quello che serve a noi è quello avente dimensione maggiore (perché contiene anche le librerie di terze parti).

Siamo finalmente pronti a mettere in esecuzione la nostra applicazione SPAF sul cluster. Questo ultimo passaggio dipendente fortemente dal framework di stream processing impiegato.

Per sottoporre la nostra applicazione al cluster Apache Flink ci sono due possibilità: utilizzare uno script messo a disposizione dal framework stesso, oppure utilizzare l'applicazione web resa anch'essa disponibile dal framework. Illustreremo l'utilizzo dello script. Sottoporre un'applicazione al cluster Flink significa effettuare l'upload del file jar corrispondente sui vari nodi del cluster, dunque mettere il jar in secuzione. Queste due operazioni vengono svolte per noi dalla CLI (Command Line Interface) messa a disposizione

da Flink.[15]

Assicuriamoci di eseguire il seguente comando dalla macchina rappresentante il nodo coordinatore (*JobManager*):

```
$ ./bin/flink run --target remote my-spaf-application-uber.jar
```

N.B.: Lo script considererà le configurazioni contenute nel file `conf/flink-conf.yaml`.

Se tutto è andato per il verso giusto, dovremmo iniziare a vedere in console numerose linee di log indicanti l'evoluzione dello stato dell'applicazione. Dopo alcuni istanti, l'applicazione dovrebbe essere attiva e in attesa di ricevere dati dalla *Source* specificata.

2.4.3 Performance

Abbiamo anticipato che il collaudo di un progetto come SPAF non può non considerare la valutazione di due tipi di performance: le performance di tipo "classico", relative cioè all'esecuzione del codice, quindi determinate da SPAF, ma che meritano comunque delle riflessioni; e le performance nell'utilizzo di SPAF da parte del programmatore di applicazioni, cioè quelle relative alla scrittura di codice.

Sono senz'altro di maggior interesse le performance del secondo tipo, poiché costituiscono il banco di prova oggettivo per valutare se RAM^{3}S mantiene davvero la promessa di semplificare la scrittura di applicazioni di stream processing. Prima, però, tratteremo le performance relative all'esecuzione di un'applicazione SPAF e cerchermo di capire qual è l'approccio più sensato per la loro valutazione poiché, come vedremo a breve, la natura di un progetto SPAF è tale da indurre più componenti indipendenti nel costo totale di esecuzione. Concluderemo il capitolo con l'analisi delle performance di scrittura del codice, aiutandoci con alcune analogie nella nostra argomentazione.

Esecuzione del codice

Un'applicazione SPAF è essenzialmente composta da più strati di astrazione, che possiamo riassumere così:

[15] Guida ufficiale alla CLI di Apache Flink: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/deployment/cli/

- strato applicativo: è il codice dell'applicazione SPAF, definito dal programmatore finale, che consiste principalmente nella definizione dei *Processor* e nella definizione della *Topology*;
- strato interpretativo: è il codice del framework SPAF, composto dal *glue-code* contenuto nel pacchetto principale (`spaf-api`) e, soprattutto, dal codice del provider di stream processing impiegato (es. `spaf-spark-streaming`), che effettua l'operazione di mappatura dell'applicazione sul framework sottostante;
- strato esecutivo: è il codice del framework di stream processing vero e proprio, reso utilizzabile dal provider SPAF, che mette in esecuzione l'applicazione tradotta.

Scelto un framework di stream processing, una data applicazione SPAF messa in moto su questo avrà un certo costo totale di esecuzione; tale costo è determinato dal tempo di CPU impiegato e dalla quantità di memoria richiesta per eseguire l'applicazione. Ogni strato applicativo summenzionato avrà un proprio peso nella determinazione del costo totale, e anche in questo caso bisognerà tenere conto di entrambi gli aspetti di CPU e memoria.

Suddividiamo dunque in componenti il costo totale di esecuzione, in base agli strati applicativi individuati e ai fattori di CPU e RAM.

Costo dello strato applicativo Il costo di esecuzione di questo strato è totalmente dipendente dalla logica dell'applicazione. Il programmatore finale ha totale libertà nel definire le operazioni di elaborazione all'interno dei *Processor*, e può utilizzare una qualunque libreria di terze parti a supporto di queste. La componente di costo imputabile a questo strato è dunque difficilmente predicibile; non si può dire a priori granché nemmeno su come verranno impiegati CPU e RAM per eseguire la logica dell'applicazione.

Per queste ragioni, escluderemo dalla nostra analisi delle performance di SPAF il costo di esecuzione dello strato applicativo.

Costo dello strato esecutivo Anticipiamo la trattazione della componente di costo dello strato esecutivo, perché anche questa componente è praticamente indipendente dal nostro framework. Infatti, si tratterebbe di valutare le performance di ogni singolo framework di stream processing in sé per sé, il che è senz'altro interessante, ma esula dal nostro intento di capire quale ruolo gioca SPAF nelle performance e in quale misura.

D'altra parte, esistono già *benchmark* a riguardo[16], dove vengono messi a confronto i vari framework e vengono confrontate le loro prestazioni in termini di *latenza*, *throughput* e *completezza* (ossia la capacità di elaborare tutte le informazioni previste, senza perdite dovute a errori o a limitatezza delle risorse di calcolo; una dimensione che noi ignoriamo). In realtà, un altro confronto interessante si può fare tra le prestazioni dei vari framework per come vengono usati da RAM^{3}S, sulla base di una stessa applicazione; un confronto di questo tipo potrebbe aiutare il progettista nella scelta del framework di stream processing migliore da impiegare per risolvere il proprio problema, ed è infatti uno degli scopi per cui era stato creato originariamente RAM^{3}S; esperimenti empirici a riguardo sono già stati realizzati in lavori di tesi precedenti a questo, si invita pertanto il lettore a fare rifermento ai rispettivi elaborati (tra questi, vedere la tesi di Berni 2021).

Per le ragioni appena esposte, dunque, escluderemo dall'analisi anche le componenti di costo imputabili ai framework di stream processing veri e propri.

Costo dello strato interpretativo Arriviamo dunque all'unica componente di costo che sembra essere imputabile al framework SPAF in sé, quella dello strato interpretativo.

Lo strato interpretativo ha un unico obiettivo: tradurre l'applicazione SPAF in un'applicazione del framework sottostante. Questa traduzione, come illustrato nella sezione Implementazione, avviene nei singoli provider e in maniera peculiare al framework integrato. Per analizzare il costo dell'operazione di traduzione, però, possiamo ignorare i costi associati a tali peculiarità; questo perché sarebbe un costo che avremmo comunque dovuto pagare nel caso avessimo realizzato un'applicazione utilizzando direttamente il framework di stream processing sottostante. Infatti, ci stiamo riferendo alla parte di codice del provider che usa direttamente le API dal framework scelto, e che non fa né più né meno di quello che avrebbe fatto il programmatore se avesse utilizzato lui stesso tali API, però lo fa in maniera automatizzata.

Non ci rimane dunque che considerare i costi dell'unica operazione rimasta, cioè quella della visita della topologia definita nell'applicazione SPAF.

Riportiamo, per l'ultima volta, qui sotto il codice che attualmente si occupa di "visitare" la topologia (aiutiamoci sempre con il provider di Apache Flink):

...

[16]Diapositive sul confronto delle prestazioni di alcune piattaforme di streaming big data: https://www.slideshare.net/HadoopSummit/performance-comparison-of-streaming-big-data-platforms

```
// 3. Mapping SPAF Topology to Flink Streaming topology
// (A)
for (Processor<> processor : application.getTopology().getProcessors()) {
    // (B)
    ProcessFunction<IN, OUT> processFunction = /* build from processor */;
    // (C)
    stream = stream.process(processFunction);
}

...
```

Per valutare il costo di questo codice, determiniamo la sua complessità temporale, per avere un'idea di quelli che saranno i costi di CPU, e la sua complessità spaziale, per i costi di utilizzo della memoria.

Nell'algoritmo di visita della topologia appaiono tre operazioni fondamentali:

- `(A)`: iterazione della topologia, per recuperare il prossimo elemento di cui effettuare la traduzione;
- `(B)`: traduzione dell'elemento nella rappresentazione del framework sottostante;
- `(C)`: ricostruzione della topologia sul framework sottostante.

Le componenti di costo associate alle operazioni `(B)` e `(C)` avvengono con tempo costante, poiché nel primo caso si tratta della sola costruzione degli oggetti contenenti il codice della logica applicativa, che invece inciderà nel costo solo a *runtime*, mentre nel secondo caso si tratta di operazioni di assegnazione di riferimenti tra oggetti.

La componente di costo associata all'operazione `(A)` è invece completamente determinata da SPAF e dipende dalla struttura a oggetti rappresentante la topologia e dall'algoritmo di visita impiegato. Dato che questa prima versione di SPAF consente di definire solo topologie di tipo "lineare", la struttura dati che rappresenta la topologia è sostanzialmente una lista di oggetti e l'operazione per la visita della topologia coincide con l'operazione di iterazione su di una lista di oggetti, come mostrato esplicitamente nel codice. Dunque, la complessità temporale dell'operazione `(A)` è lineare nel numero dei nodi che compongono la topologia, perciò $O(N)$. Per quanto riguarda la complessità spaziale, invece, dobbiamo tenere in conto che ci troveremo ad avere una doppia rappresentazione della stessa topologia; per cui possiamo approssimare la complessità spaziale a $O(2N)$,

anche se in realtà gran parte degli oggetti partecipa a entrambe le rappresentazioni grazie all'utilizzo dei riferimenti e senza necessariamente presentare dei duplicati.

Un'ultima considerazione sui costi relativi a questo strato di codice è che debbono essere pagati una volta soltanto all'avvio dell'applicazione; una volta che SPAF ha fatto il suo lavoro di traduzione, l'applicazione è diventata a tutti gli effetti nativa rispetto al framework sottostante, e a quel punto entrano in gioco i costi summenzionati specifici alla logica applicativa e al framework utilizzato.

Possiamo dunque concludere che il costo di esecuzione imputabile propriamente al framework SPAF è estremamente contenuto, e che viene ulteriormente ridimensionato se calato nel contesto in cui un'applicazione di stream processing è pensata per funzionare: l'elaborazione di grandissime quantità di dati, i quali vanno a "gonfiare" solo i costi dello strato applicativo e di quello esecutivo, in maniera proporzionale al loro volume.

Scrittura del codice

RAM^{3}S si concentra sulle performance di scrittura del codice sia dal punto di vista quantitativo che dal punto di vista qualitativo:

- performance di tipo quantitativo:
  - numero linee di codice: considerare il codice da scrivere in termini di numero di righe e raffrontarlo con il codice che si sarebbe dovuto scrivere per realizzare la stessa applicazione direttamente in uno dei framework integrati;
- performance di tipo qualitativo:
  - tipologia di linguaggio: considerare quale tipo di linguaggio di programmazione utilizzare per realizzare i vari aspetti di un'applicazione di stream processing, mettendo a confronto cosa offre SPAF rispetto ai framework sottostanti.

Entrambi questi fattori concorrono nel determinare la dimensione a cui siamo sostanzialmente interessati:

- **tempo di scrittura del codice**: considerare il tempo necessario a un programmatore per realizzare un'applicazione di stream processing, raffrontando la complessità in cui impegnarsi per la creazione di una aplicazione SPAF rispetto a quella per creare un'applicazione "nativa".

Il tempo di scrittura di codice è evidentemente collegato al numero di linee di codice da scrivere, ma non è determinato unicamente da questo, tutt'altro. Il compito del programmatore è quello di descrivere tramite il codice la soluzione a un problema, è evidente che non è sufficiente digitare meno lettere sulla tastiera per asserire che una soluzione è migliore di un'altra. La vera difficoltà sta infatti nel capire cosa scrivere per risolvere il problema. Una chiara rappresentazione di quelli che sono i concetti principali del dominio del problema da risolvere è senz'altro un ingrediente fondamentale per capire l'istanza del problema e sviluppare di conseguenza una soluzione. RAM^{3}S tenta di rendere più accessibile lo sviluppo di applicazioni di stream processing, normalizzando l'insieme di concetti che ruotano attorno a questo problema e fornendone una rappresentazione unificata.

Un altro aspetto legato alla difficoltà di scrittura del codice, dunque al tempo necessario, è il linguaggio di programmazione da usare, cioè lo strumento. Lo sviluppatore dovrà a un certo punto descrivere la soluzione che ormai ha intuito: c'è una apprezzabile differenza nel scrivere certe parti di tale soluzione in modo dichiarativo invece che in modo imperativo. Nella maggior parte dei framework di stream processing è necessario utilizzare prevalentemente l'approccio imperativo (Java puro) per descrivere la propria soluzione al problema; in RAM^{3}S, invece, si è messo a disposizione del programmatore anche l'approccio dichiarativo per descrivere certe parti della soluzione (quali, ad esempio, la definizione di *Source* e *Sink*). Non ha senso parlare di quale sia il migliore dei due approcci: la programmazione dichiarativa e imperativa sono equivalenti in termini di potenza espressiva, però è più facile usare una o l'altra a seconda del problema che si intende risolvere; un problema di strumenti, appunto.

Latenza e Throughput Proponiamo ora una possibile interpretazione delle prestazioni di scrittura del codice in termini delle due metriche ricorrenti solitamente adottate quando si parla di performance: *latenza* e *throughput*.

La definizione di *latenza* nell'ambito della comunicazione su reti è la seguente (trad. dell'autore): «Latenza è il tempo che un pacchetto dati impiega per viaggiare da una sorgente a una destinazione.» (*Understanding latency - Web Performance | MDN* 2022) Se caliamo questa definizione sul problema di realizzare un'applicazione di stream processing, potremmo riformulare come segue: «Latenza è il tempo che impiega un programmatore per risolvere per la prima volta un problema di stream processing (sorgente) scrivendo la relativa soluzione in codice (destinazione), tramite SPAF».

Le argomentazioni appena presentate dovrebbero sostenere l'importanza di questa metrica e come questa sia collegata agli aspetti quantitativi e qualitativi sopra esposti.

Rimanendo sempre nell'ambito delle reti di comunicazione, il *throughput* è definito come segue (trad. dell'autore): «Throughput è il tasso di consegna (con successo) dei messaggi su un canale di comunicazione». (*Throughput - Wikipedia* 2022)
Anche per questa metrica, riformuliamo in termini del nostro contesto: «Throughput è il tasso di realizzazione di applicazioni di stream processing, tramite SPAF».
Questa definizione merita invece una riflessione: non ha molto senso mettere a confronto il numero di applicazioni scritte per unità di tempo, in base al framework utilizzato. Si sa, però, quanto il tempo sia la risorsa più preziosa nella realizzazione di progetti di qualsiasi tipo; i progetti software non sono un'eccezione a questa regola, anzi. Pertanto, ha più senso considerare come incremento di *throughput* il tempo risparmiato nello scrivere la singola applicazione di stream processing, perché tale tempo può essere reinvestito nello sviluppo della stessa applicazione, per migliorarne il codice e per renderla più robusta (magari scrivendo qualche test in più). Il maggior *thorughput* si traduce pertanto in maggiore qualità dell'applicazione finale.

Curva di apprendimento Un'altra dimensione di fondamentale importanza, che va tenuta in conto nella scelta di un qualsiasi strumento di sviluppo, è la cosiddetta "curva di apprendimento", che mette in relazione il livello di conoscenza e il tempo investito nell'apprendere una cosa nuova. Poiché SPAF incarna il lavoro di ricerca e di normalizzazione dei concetti relativi allo stream processing, può semplificare la vita al programmatore inesperto che deve avventurarsi per la prima volta nel mondo dello stream processing, e rendere tale viaggio meno insidioso. Azzardando un'analogia, l'ultima, potremmo dire che SPAF può svolgere il ruolo della "guida" che accompagna il programmatore nella scoperta dei concetti dello stream processing, fornendo un percorso logico che ne faciliti la comprensione e rendendo così la curva di apprendimento dello stream processing meno scoraggiante.

Figura 2.22: «**74.** *Inf.* XXXIV [...] Dante e Virgilio passano attraverso una caverna e un cunicolo, scavato da un ruscello, per tornare in superficie.» (Doré 2013)

Capitolo 3

Considerazioni Avanzate

In questo capitolo riporteremo una serie di considerazioni avanzate emerse durante la fase realizzativa del progetto. Possiamo dividere l'insieme di considerazioni che tra poco presenteremo in due macro categorie: considerazioni sugli aspetti peculiari dei framework di stream processing messi a confronto tra loro, e come questi aspetti si relazionino con SPAF, e considerazioni sulle possibili modalità di utilizzo di SPAF in maniera avanzata.

3.1 Aspetti peculiari dei framework

In questa sezione illustreremo una serie di aspetti riguardanti, in modo trasversale, tutti i framework di stream processing considerati in questa prima versione del progetto. Tali aspetti riguardano la modalità di esecuzione delle applicazioni sui framework, o meglio delle relative topologie, la gestione della *type-safety* e della serializzazione e infine le tipologie di API esposte dai framework.

3.1.1 Modalità di esecuzione della topologia

Seppure i framework presi in cosiderazione si occupano tutti di fornire strumenti per la realizzazione di applicazioni di stream processing, questi non sono tutti concordi sulla modalità con cui tali strumenti debbano essere messi in atto. Ci riferiamo in particolar modo a come i vari framework interpretino il concetto fondamentale di *topologia* e le conseguenze che derivano da queste differenze interpretative.

In questa sezione ci concentreremo sulle differenze nella modalità di esecuzione della topologia tra i vari framework e scopriremo come, in certi casi, queste differenze siano a

tal punto notevoli da presupporre un'interpretazione profondamente diversa del concetto stesso di topologia da parte di taluni framework.

Topologie in Apache Spark

Innanzitutto, ricordiamo che Apache Spark realizza le funzionalità di stream processing basandosi sul concetto di RDD, pertanto questo concetto lo ritroveremo nelle cosiderazioni che stiamo per illustrare.

In Apache Spark il concetto di topologia è associato direttamente al concetto di *Logical Execution Plan*, ma per dare una definizione di questo concetto dobbiamo prima introdurre il concetto di *RDD Lineage* (in italiano, *lineage* si può tradurre con «casata» o «lignaggio» (*lineage - Traduzione del vocabolo e dei suoi composti, e discussioni del forum.* 2022)).

Un "lignaggio" di RDD è il grafo di tutti gli RDD da cui discende un certo RDD. Questo grafo di discendenza è il risultato dell'applicazione di operazioni di trasformazione a un RDD. Pertanto, un "lignaggio" di RDD corrisponde al grafo delle trasformazioni da eseguire in risposta all'invocazione di un'operazione di *sinking*. Nella sezione Modello di un'applicazione di Stream Processing abbiamo visto che in Apache Spark un'operazione di *sinking* è realizzabile con il metodo `foreachRDD`.

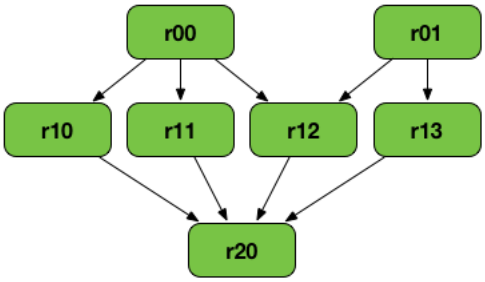

Figura 3.1: RDD Lineage (Laskowski 2021a)

Un *Logical Execution Plan* rappresenta, appunto, il piano di esecuzione di una topologia logica: il suo inizio coincide con la creazione degli RDD "progenitori" e termina con gli RDD risultanti dalle operazioni di *sinking*. Un *Logical Execution Plan* può dunque essere inteso come il **DAG logico** corrispondente alla topologia definita dall'applicazione.

Per eseguire un'applicazione, Apache Spark materializza un DAG logico in un **DAG fisico**, chiamato *Physical Execution Plan*, composto invece da un grafo di più *Stage*. L'operazione di materializzazione viene innescata da un'operazione di *sinking*, che in Apache

Spark viene indicata con il nome generico *Action*. Apache Spark introduce un ulteriore concetto, che serve a racchiudere i piani di esecuzione così determinati, il *Job*. C'è dunque una corrispondenza univoca tra un'operazione di *sinking* (o *Action*) e un *Job*.

Dunque, per creare un *Job*, Apache Spark trasforma un piano di esecuzione logico (definito dai "lignaggi" di RDD) in un piano di esecuzione fisico (definito da *stage*). Ciascuna delle operazioni definite nel DAG logico deve essere quindi essere pianificata in termini di *stage*.

Uno *Stage* è un'astrazione dei passi elementari che un piano di esecuzione fisico deve compiere per calcolare un certo RDD del DAG logico. Ciascun RDD è, in realtà, partizionato; ossia, l'insieme dei dati che un singolo RDD rappresenta è suddiviso in maniera logica, con lo scopo di consentire la distribuzione del calcolo. L'operazione di mappatura tra un RDD e uno Stage tiene dunque conto di questo partizionamento, e ciò si traduce nell'assegnazione di un *Task* a sé per ogni partizione di un RDD.

In poche parole, un *Job* Apache Spark è una computazione divisa in più *Stage* (Laskowski 2021a), ciascuno dei quali è a sua volta composto da più *Task*.

Il diagramma sottostante riassume la corrispondenza che c'è tra i concetti del DAG logico e quelli del DAG fisico, ossia tra *Action* e *Job* e tra RDD e *Stage*; e come quest'ultima corrispondenza venga declinata a basso livello tra partizioni e *Task*.

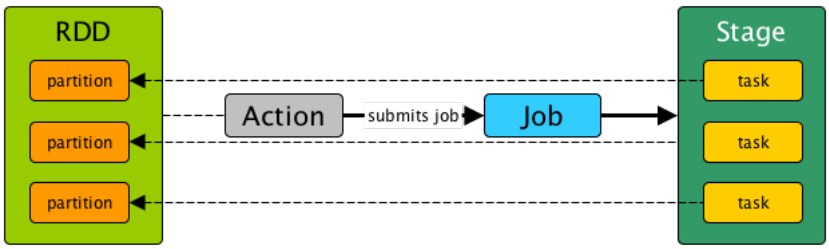

Figura 3.2: Mapping tra RDD e Stage (Laskowski 2021b)

Se si considera che in una singola applicazione Apache Spark possono essere definite più operazioni di *sinking*, ciò significa che una singola applicazione può essere tradotta nell'esecuzione di più *Job* distinti. Il livello di parallelismo dell'esecuzione di una topologia logica non è determinato dunque solo dagli *Stage*, e dai rispettivi *Task*, ma anche dalla potenziale creazione di più *Job* per la stessa applicazione. Questo ci porta a concludere che il modello di esecuzione di Apache Spark è altamente distribuito, e che la topologia

logica definita nell'applicazione viene di fatto decomposta per essere eseguita in maniera distribuita, e coordinata, sui nodi del cluster.

Non abbiamo parlato per nulla di coordinamento nella distribuzione del calcolo, nella nostra trattazione siamo però interessati agli aspetti di alto livello; l'intento della nostra analisi è limitato al capire il grado di distribuzione del calcolo per l'esecuzione di una topologia.

Topologie in Apache Storm

Mentre in Apache Spark non si ha evidenza, lato programmatore di applicazioni, come avvenga la distribuzione del calcolo (nel capitolo precedente abbiamo fatto riferimento a documentazione di terze parti, e ci siamo dovuti avvenutare negli *internals* di Apache Spark), in Apache Storm si capisce fin da subito che l'esecuzione della topologia avviene in maniera distribuita. Non solo.

In Apache Storm è evidente che il programmatore di applicazioni può addirittura specificare degli aspetti relativi all'esecuzione distribuita dell'applicazione, può cioè intervenire anche sulla topologia fisica in maniera esplicita.

Tutto ciò è spiegato nella documentazione di Apache Storm: *«Each spout or bolt executes as many tasks across the cluster. Each task corresponds to one thread of execution [...]»* (*Concepts* 2022); ma se ne ha anche un'evidenza diretta nelle API: i metodi `setSpout` e `setBolt`, in alcune delle loro versioni, consentono di specificare un parametro per "suggerire" al framework il grado di parallelismo desidearto per l'esecuzione della rispettiva operazione:

> `parallelismHint` - the number of tasks that should be assigned to execute this bolt. Each task will run on a thread in a process somewhere around the cluster. (*TopologyBuilder (Storm 2.3.0 API)* 2022)

Una prima dimensione che determina il grado di distribuzione del calcolo in Apache Storm è dunque il *Task*, menzionato nella documentazione appena illustrata.

La Figura 3.3 illustra la corrispondenza che c'è tra un nodo della topologia logica, ovvero gli *Spout* o *Bolt* rappresentati con rettangoli, e i corrispondenti nodi della topologia fisica, ovvero i *Task* rappresentati con cerchi.

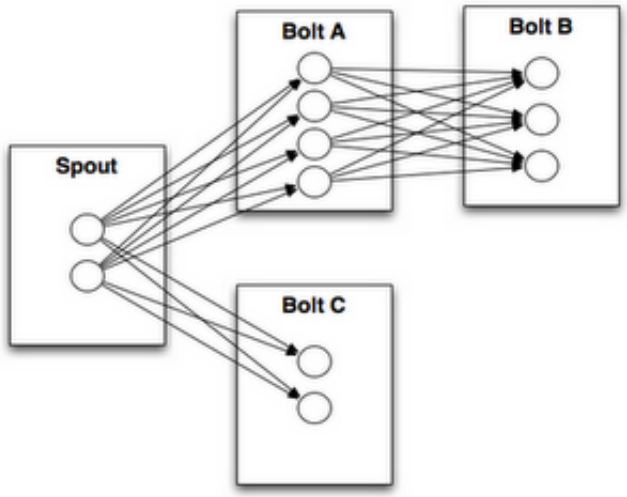

Figura 3.3: Corrispondenza tra *Spout*/*Bolt* e *Task* (Marz 2011)

Un *Task* di Apache Storm corrisponde a un unico thread di esecuzione. Ogni *Spout* e *Bolt* di un'applicazione Storm viene eseguito come più task distribuiti nel cluster; il numero di *Task* per ciascun *Spout* e *Bolt*, come abbiamo visto, è definibile tramite il parametro `parallelismHint` summenzionato.

I *Task* di cui stiamo parlando, in quanto thread, dovranno però essere eseguiti da dei processi di esecuzione veri e propri, dei processi per così dire "contenitori" di *Task*. Questi processi in Apache Storm vengono realizzati dal concetto di *Worker*.

Un *Worker* di Apache Storm corrisponde a una JVM "fisica", cioè a un processo pesante, in esecuzione su di una certa macchina del cluster. Il compito di ciascun *Worker* è eseguire un sottoinsieme dei task creati per una topologia. Anche il numero di *Worker* può essere definito dal programmatore finale, tramite il parametro di configurazione `Config.TOPOLOGY_WORKERS` legato alla topologia.

Possiamo dunque concludere che anche Apache Storm fornisce un modello di esecuzione altamente distribuito, paragonabile a quello di Apache Spark. A differenza di Apache Spark, però, in un'applicazione Storm il programmatore finale ha la possibilità di interessarsi di aspetti riguardanti la topologia fisica (in Storm costituita dall'insieme dei *Task*, eseguiti dai vari *Worker*); questo consente potenzialmente di determinare una distribuzione del calcolo più conveniente per l'applicazione rispetto a quella determinata in maniera automatica dal framework (possibilità comunque esistente anche in Storm), e di ottenere di conseguenza performance maggiori oppure un minor impiego di risorse.

Anche nel caso di Storm ci siamo disinteressati degli aspetti di coordinamento del calcolo distribuito; vale la pena però indicare che, in Storm, il programmatore finale ha visibilità anche su alcuni aspetti di questo tema; può, ad esempio, specificare come lo stream di dati in ingresso a un *Bolt* debba essere partizionato tra i relativi task[1].

Topologie in Apache Flink

Anche nel caso di Apache Flink, come fatto per Apache Spark, prima di affrontare i concetti cruciali per il calcolo distribuito, abbiamo bisogno di introdurre un po' di terminologia legata ai concetti più astratti. Per tale compito, faremo riferimento alla documentazione ufficiale di Apache Flink[2].

Partiamo anche in questo caso dal concetto di topologia logica, corrispondente alla topologia definita dall'applicazione SPAF.

In Apache Flink la topologia logica trova perfetta corrispondenza nel concetto di *Logical Graph*, cioè un grafo diretto (il solito DAG) dove i nodi vengono detti *Operator* e gli archi tra questi definiscono le operazioni di input/output e corrispondono agli stream di dati. Dunque, una topologia SPAF corrisponde a un *Logical Graph* Flink.

Un *Operator* Flink rappresenta una certa operazione da eseguire all'interno della topologia. Tale operazione viene normalmente specificata tramite una *Function* di Apache Flink, che consente di incapsulare la logica di trasformazione dell'applicazione, ma anche tramite i concetti di *Source* e *Sink*, che vengono intesi come tipi speciali di operatori, che si occupano rispettivamente dell'immissione e dell'emissione dei dati dalla topologia.

L'ultimo concetto astratto, molto semplice, che vale la pena introdurre è la cosiddetta *Operator Chain* che altro non è che la sequenza di due o più *Operator* consecutivi di una topologia, connessi in modo tale che gli elementi dello stream flusicano da un *Operator* all'altro in maniera praticamente trasparente al framework (nella sezione Type-Safety e serializzazione vedremo cosa comporta l'eventuale intervento di Flink nel passaggio dei dati da un *Operator* all'altro).

[1] A tal proposito, vedasi la sezione "Stream groupings" della documentazione ufficiale: https://storm.apache.org/releases/2.3.0/Concepts.html

[2] Documentazione ufficiale di Apache Flink - Glossario: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/glossary/

Veniamo dunque ora ai concetti di Apache Flink relativi all'esecuzione di un *Logical Graph*. Anche in Apache Flink la distribuzione del calcolo di un'applicazione ruota attorno ai concetti di *Job* e *Task*.

Un *Job* Flink è la rappresentazione a tempo di esecuzione di un *Logical Graph*. Questa rappresentazione viene anche detta *Physical Graph* (o *Dataflow [graph]*), e viene creata all'atto dell'invio dell'applicazione al cluster. I nodi di un *Physical Graph* sono i *Task* Flink e gli archi indicano, anche in questo caso, le relazioni di input/output tra i nodi.

Un *Task* è l'unità di esecuzione di base del *runtime* di Flink. I *Task* incapsulano (ossia, eseguono) esattamente un'unica istanza parallela di un *Operator* o di una *Operator Chain*. Per istanza parallela si intende il fatto che per ciascun *Operator*, o per ciascuna *Operator Chain*, possono esistere in realtà più *Task* paralleli detti *Sub-Task*, ciascuno dei quali si occupa del processamento di una singola partizione dello stream in input.

Il diagramma sottostante esprime graficamente i concetti e le relazioni appena esposte (*Flink Architecture | Apache Flink* 2022).

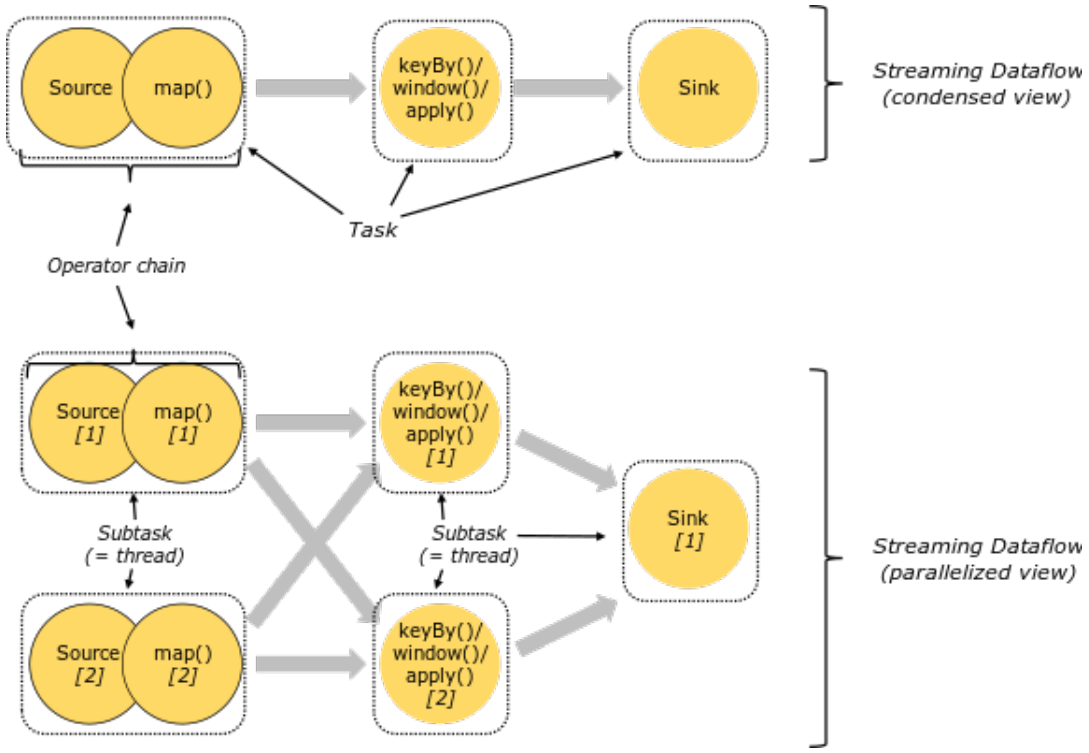

Figura 3.4: Decomposizione di un *Logical Graph* in un *Physical Graph*

Senza grandi sorprese, dunque, anche Apache Flink fornisce un supporto altamente distribuito all'esecuzione di una topologia logica, paragonabile a quello offerto da Apache

Spark e Apache Storm.

Concludiamo riportando una funzionalità di Apache Flink degna di nota e che riguarda sempre le topologie. Il concetto di topologia inteso da Flink è molto simile a quello visto in Apache Storm, tant'è che Flink mette addirittura a disposizione una libreria che consente di eseguire topologie precedentemente definite con le API di Storm direttamente su un cluster Flink, senza dover modificare il codice (Sax 2015).

> *Flink ships with a Storm compatibility package that allows users to:*
>
> - *Run **unmodified** Storm topologies using Apache Flink benefiting from superior performance.*
> - ***Embed** Storm code (spouts and bolts) as operators inside Flink DataStream programs.*

Topologie in Apache Samza

La documentazione di Apache Samza è stata molto utile nel capire qual è l'approccio di questo framework all'esecuzione di un'applicazione di stream processing, e come questo approccio si confronti rispetto a quelli degli altri framework.

La distribuzione del calcolo di un'applicazione di stream processing in Samza è ottenuta dividendo l'elaborazione in *Task* indipendenti, che possono essere parallelizzati. Questi *Task* vengono eseguiti dai cosiddetti *Container*. È possibile eseguire più *Task* in un *Container*, o solo un *Task* per *Container*; in ogni caso, va considerato che un *Container* usa solo un thread, che corrisponde esattamente a una CPU (*Samza - Spark Streaming* 2022).

In Spark Streaming, Storm e Flink, si costruisce un intero grafo di elaborazione per una singola applicazione, ciò che abbiamo indicato con topologia logica; dunque si considera questa topologia come un'unità. In seguito all'invio della topologia sul cluster, questa viene trasformata in una topologia fisica la quale, a seconda del framework, realizza in maniera distinta la distribuzione del calcolo delle operazioni definite nel grafo logico. In questi framework, la comunicazione tra i nodi del grafo di processamento viene realizzata dai rispettivi ambienti di esecuzione.

Samza è completamente diverso. Ogni *Job* Samza processa completamente ciascun messaggio. In Samza non c'è il concetto di topologia, dunque non c'è nemmeno il relativo supporto da parte del framework. L'output di un *Task* di processamento non può che

essere reindirizzato immediatamente a un message broker. In sostanza, il supporto da parte del framework per la comunicazione tra i nodi della topologia, di cui parlavamo prima, in Samza non c'è; questo perché non ha senso parlare di topologie in Samza. Come vedremo presto, però, nell'idea di Samza, il supporto alla comunicazione tra *Task* viene invece delegato al message broker impiegato (ad esempio, Apache Kafka); in Samza, pertanto, l'idea che l'output di una "topologia" costituisca l'input di un'altra "topologia" è una scelta di design.

In altre parole, Apache Samza esegue un DAG di operatori come singolo task su un'unica JVM; mappa cioè una topologia logica in una topologia fisica costituita da un unico nodo (ossia, un unico thread) di esecuzione. Da ora in poi, per chiarezza, faremo riferimento a una "topologia" Samza (le parentesi, quì, sono significative) con il termine **sub-topologia**.

Il diagramma sottostante cerca di rappresentare graficamente l'architettura appena descritta.

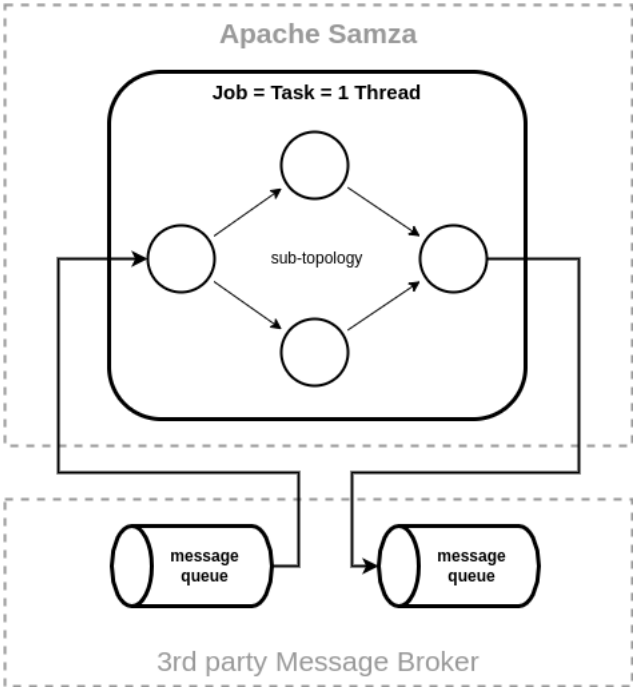

Figura 3.5: Esecuzione *runtime* di Apache Samza

L'unità fondamentale per il calcolo distribuito in Apache Samza è dunque il *Task*. L'esecuzione di un'applicazione Samza può portare alla creazione di più *Task* identici,

eseguiti in parallelo nel cluster e che svolgono tutti lo stesso compito: calcolare l'intera topologia logica (cioè, la sub-topologia) definita nell'applicazione.

Il fatto che Apache Samza non fornisca un supporto "nativo" alle topologie si è manifestato in tutta la sua chiarezza all'atto di implementare il provider SPAF per questo stesso framework. Nel momento in cui ci si è trovati a scegliere quali entità/operazioni delle API di Samza utilizzare per effettuare il mapping di un *Processor* SPAF, ci si è accorti che non esisteva nulla di equivalente in termini semantici. L'unica scelta possibile è stata quella di realizzare uno *StreamTask*, appunto.

Per questa ragione, è stato necessario realizzare del codice che effettuasse una mappatura "manuale" della topologia logica SPAF direttamente su una sub-topologia di Apache Samza. Ecco dunque il codice del metodo `process` della classe `TopologyStreamTask`, sottoclasse di `StreamTask`, che esegue l'operazione appena descritta:

```
@Override
public void process(
    IncomingMessageEnvelope envelope,
    MessageCollector collector
) {
    Object key = envelope.getKey();
    Object value = envelope.getMessage();
    List<Element<Object, Object>> elemsToProcess =
        Collections.singletonList(new Element<>(key, value));

    /**
     * "Emulazione" della topologia fisica,
     * tramite esecuzione di tutti i Processor
     */
    for (Processor<Object, Object, Object, Object> proc : processors) {
        Collector<Object, Object> procCollector = new CollectorImpl<>();

        for (Element<Object, Object> e : elemsToProcess) {
            proc.process(e.getKey(), e.getValue(), procCollector);
        }

        elemsToProcess = procCollector.getCollectedElements();
    }

    List<OutgoingMessageEnvelope> outElements =
        mapToOutElements(elemsToProcess);
    collector.send(outElements);
}
```

Per realizzare "nativamente" in Samza una topologia che abbia un grado di distribuzio-

ne del calcolo confrontabile con gli altri framework è necessario collegare tra loro i singoli *Job* Samza, cioè le singole sub-topologie, tramite dei canali di comunicazione esterni, utilizzando come supporto di realizzazione dei canali un broker di messaggi (come Apache Kafka). Come vedremo nella sezione Realizzazione di Super-Topologie, SPAF rende molto semplice tale operazione. Per il momento ci basti sapere che una super-topologia è una topologia di topologie logiche, messe in comunicazione tra loro sfruttando il supporto di un message broker; poiché in Samza le topologie logiche vengono dette sub-topologie, con questo approccio realizzeremmo una super-topologia di sub-topologie. Elidendo "algebricamente" i prefissi "sub-" e "super-" otteniamo quindi una topologia, avente la stessa natura e grado di distribuzione del calcolo confrontabile alle topologie degli altri framework.

3.1.2 *Type-safety* e serializzazione

Quando si parla di *Type-safety* nei framework di stream processing non ci si riferisce solo ai "classici" temi che riguardano la scrittura e la compilazione del codice, come il grado di assitenza alla scrittura del codice da parte dell'IDE o la rilevazione di alcuni errori da parte del compilatore, ma si considerano anche le implicazioni che questa caratteristica, solitamente desiderabile, può avere nell'esecuzione distribuita di un'applicazione di stream processing.

In questa sezione scopriremo come le caratteristhce di *Type-safety*, relative a un framework di stream processing, determinino la gestione della serializzazione degli oggetti che viaggiano nelle topologie, nell'ambiente di esecuzione corrispondente.

Nella sezione Progettazione abbiamo anticipato la necessità di gestire la serializzazione degli oggetti da immettere in una topologia SPAF, nel caso questi provengano da un sistema esterno quale un broker di messaggi (ad esempio, Apache Kafka), e della esigenza duale, nel caso gli oggetti debbano essere emessi dalla topologia per essere trasmessi su un canale di messaggi. Nella sezione Implementazione abbiamo visto come questo problema venga risolto, in certi casi, dalla libreria client del sistema di messaggi.

La serializzazione non è però un aspetto che riguarda soltanto l'input e l'output della nostra applicazione, ovvero i *Source* e i *Sink* della nostra topologia.

Nella sezione precedente abbiamo visto come quasi tutti i framework, dietro le quinte,

mettano in atto dei meccanismi per la distribuzione del calcolo di una topologia logica. Nel momento in cui una parte della logica di processamento della topologia viene spostata su un'altra JVM sorge la necessità di effettuare la serializzazione (e deserializzazione) degli oggetti che vengono processati all'interno dalle varie componenti della topologia.

Quindi, in uno scenario in cui la nostra applicazione venga eseguita in modo distribuito, si configura implicitamente la necessità di gestire la serializzazione degli oggetti che vengono scambiati dai nodi di calcolo derivanti dalla topologia logica della nostra applicazione. Abbiamo visto come ciascun framework traduca in maniera diversa la topologia logica della nostra applicazione in una topologia fisica equivalente, in grado di essere eseguita nel cluster. Per semplificare la trattazione di questo argomento, però, ipotizzeremo che tutti i framework di stream processing effettuino la conversione della topologia logica allo stesso modo, almeno per quanto riguarda gli aspetti riguardanti la serializzazione e deserializzazione degli oggetti che viaggiano all'interno della topologia.

In sostanza, ipotizzeremo quanto segue: ogni *Processor* della topologia SPAF verrà mappato su un'unità di calcolo indipendente del framework sottostante, che indicheremo con il termine generico *Task* (il nome scelto vuole richiamare le varie declinazioni di task viste in precedenza per ciascun framework).

Ora che abbiamo normalizzato la trattazione al concetto astratto di *Task*, andiamo a vedere come i framework di stream processing affrontano concretamente il problema della serializzazione/deserializzazione degli oggetti in viaggio tra un *Task* e l'altro. Prenderemo in esame due framework che adottano approcci diametralmente opposti: Apache Storm e Apache Flink.

Type-safety **e serializzazione in Apache Storm**

Rispetto alla gestione dei tipi degli oggetti che viaggiano nelle topologie, Apache Storm adotta un approccio semplificativo: in Apache Storm non c'è la gestione statica dei tipi, tutto si basa su tipizzazione dinamica.

Il team di Apache Storm motiva questa scelta con ragioni riguardanti l'aumento di complessità del codice lato API, cioè del codice che deve scrivere lo sviluppatore di applicazioni.

Riporto qui sotto il paragrafo della documentazione ufficiale di Apache Storm (*Serialization* 2022) (grassetti miei):

> *Adding static typing to tuple fields would* ***add large amount of complexity to Storm's API****. Hadoop, for example, statically types its keys and values but requires a huge amount of annotations on the part of the user. Hadoop's API is a burden to use and* ***the 'type safety' isn't worth it****. Dynamic typing is simply easier to use.*

All'atto pratico, un programmatore di applicazioni Apache Storm non si troverà dunque a dover dichiarare il tipo per i campi in una Tupla; potrà semplicemente valorizzare i campi della tupla con oggetti Java di tipo arbitrario, sarà poi Storm a determinare in modo dinamico come deve avvenire la loro serializzazione.

In realtà, Apache Storm delega l'operazione di serializzazione degli oggetti a una libreria di terze parti utilizzata anche da altri framework di stream processing per lo stesso scopo. La libreria di serializzazione usata da Apache Stork si chiama Kryo[3] e si tratta di un progetto Open-source il cui scopo è, appunto, fornire per Java (trad. dell'autore) «un framework di serializzazione veloce ed efficiente di grafi di oggetti binari» (*kryo/README.md at master - EsotericSoftware/kryo* 2022).

La libreria Kryo è in grado di gestire autonomamente la serializzazione e deserializzazione degli oggetti nella maggior parte dei casi; ci sono casi però in cui la libreria non è in grado di effettuare tali operazioni. Per gestire questi casi particolari, la libreria supporta la definizione da parte del programmatore di serializzatori personalizzati, e questa funzionalità viene esposta anche da Apache Storm.

Nel caso in cui Kryo non riesca a svolgere la serializzazione, vuoi perché non sono stati definiti serializzatori personalizzati, oppure perché questi non contemplano tutti i casi, allora Apache Storm ripiega sull'utilizzo della serializzazione standard fornita dalla JVM. Si è però scoperto, grazie sempre alla documentazione Apache Storm (*Serialization* 2022), che in realtà il processo di serializzazione standard di Java è estremamente costoso, sia in termini di CPU che di memoria, a causa delle grandi dimensioni della rappresentazione serializzata dell'oggetto.

Il team di Apache Storm si raccomanda pertanto di fare affidamento sulla serializzazione standard Java solo per scopi di prototipazione delle topologie.

[3]Repository del progetto Kryo: https://github.com/EsotericSoftware/kryo

Quest'ultima considerazione è dunque valida anche per la prima versione del nostro framework, poiché adotta lo stesso approccio di Apache Storm alla *type-safety* (tutto è un `Object`), ma non delega la serializzazione degli oggetti a framework esterni alla JVM.

Type-safety e serializzazione in Apache Flink

Se Apache Storm invitava il programmatore finale a non preoccuparsi di specificare i tipi di oggetto costituenti le tuple che viaggiano nella topologia, poiché alla fine tutto viene gestito da una potente libreria di serializzazione sottostante, in Apache Flink il programmatore di applicazioni è incoraggiato a specificare i tipi degli oggetti in ogni occasione possibile fornita dalle API, poiché questo rende la scrittura del codice e il debug più agevoli e ha anche importanti risvolti sulle performance di serializzazione degli oggetti da parte del framework. L'utilizzo della *type-safety* in Apache Flink è dunque desiderabile e non risulta un onere gravoso per il programmatore finale, perché è il framework stesso a occuparsi di tutta la "magia" necessaria affinché tutto funzioni.

Se riprendiamo dunque le due riflessioni chiave riportate nella sezione precedente riguardo le motivazioni di Apache Storm nel non usare la *type-safety*, possiamo dire che Apache Flink ha risposto a ciascuna in questi modi:

- *«Adding static typing to tuple fields would* ***add large amount of complexity to Storm's API [...]****»* (*Serialization* 2022): il team di Apache Flink ha sviluppato un sofisticato sistema di gestione dei tipi, che risiede nel "cuore" del framework; questo sistema tiene conto delle limitazioni imposte da Java a tal proposito (come la *type erasure* operata dal compilatore) e fa tutto il necessario affinché il programmatore finale non debba far nulla lato API (niente annotation, o meta-programmazione di altro tipo). Dunque, Apache Flink risolve questo problema spostando l'onere dal programmatore di applicazioni al programmatore del framework; la complessità non appare nelle API, ma è gestita nel codice implementativo del framework.

- *«Hadoop's API is a burden to use and* ***the 'type safety' isn't worth it [...]****»* (*Serialization* 2022): è quindi certo che la gestione completa dei tipi non è cosa affatto facile da codificare; la scelta di realizzare questo strato deve pertanto avere dei risvolti positivi e che ripaghino gli sforzi. Grazie alla conoscenza pressoché completa dei tipi di oggetto che viaggiano nelle topologie, il framework Apache Flink

può mettere in atto delle ottimizzazioni importanti riguardo la serializzazione e la deserializzazione di questi oggetti; il guadagno è dunque in termini di prestazioni, ed è tale da giustificare lo sforzo sulla *type-safety*.

Abbiamo visto come Apache Flink motiva le proprie scelte e quale approccio impiega, a grandi linee, per superare i problemi di natura tecnica. Diamo ora qualche dettaglio rispetto a come Apache Flink realizza il supporto alla *type-safety* e alla serializzazione.

La gestione dei tipi di dato e della serializzazione in Apache Flink è piuttosto sofisticata e non si è trovato qualcosa di equivalente negli altri framework presi in analisi. La funzionalità, forse la più notevole, messa in atto il framework è l'estrazione dei tipi generici dal codice specificato dal programmatore di applicazioni, che si scontra con le limitazioni imposte da Java (ricordiamo ancora la *type erasure*, operata dal compilatore) e le supera tramite artifici tecnici. Oltre a ciò, Apache Flink fornisce al programmatore finale dei propri descrittori per i tipi di tuple e realizza la funzionalità di serializzazione/deserializzazione con un proprio framework sottostante e con caratteristiche avanzate.

Apache Flink gestisce diverse categorie di tipi di dato (*Overview | Apache Flink* 2022), elenchiamo quelle più significative per la nostra trattazione:

1. *Tuple* Java: sono i descrittori proprietari a cui abbiamo appena accennato, le tuple sono tipi composti che contegono un numero fisso di campi di vario tipo; vengono messe a disposizione classi da `Tuple1` a `Tuple25`; ogni campo di una tupla può essere a sua volta uno qualunque dei sette tipi qui elencati, è dunque possibile creare tuple innestate.

2. POJO Java: questa categoria di oggetti è molto importante in Apache Flink, perché porta ad alte performance nella lora serializzazione/deserializzazione; vengono trattati come tali tutti gli oggetti che soddisfano i seguenti requisiti:

   - la classe dev'essere `public`;
   - deve esistere un costruttore `public` senza argomenti;
   - tutti gli attributi devono essere `public`, oppure essere accessibili tramite funzioni *getter* e *setter*;
   - il tipo di ciascun attributo dev'essere supportato da uno dei serializzatori inclusi nel framework (vedremo tra poco cosa sono).

3. Tipi primitivi: sono i classici tipi scalari Java, come `Integer`, `String` e `Double`.

4. Classi Java generali: Flink è in grado di gestire la serializzazione della maggior parte delle classi appartenenti alle API Java, o di terze parti; non sono supportate classi che contengono campi che non possono essere serializzati (come socket, puntatori a file, etc.).

5. *Value* di Flink: questo tipo di oggetti descrive la propria serializzazione e deserializzazione in maniera manuale; quando il framework incontra questi oggetti, non li sottopone a uno dei serializzatori interni ma delega le operazioni ai metodi `read` e `write` dell'interfaccia `org.apache.flink.types.Value`, implementata dagli oggetti stessi.

Diamo ora qualche informazione su come Apache Flink aggiri i limiti imposti da Java per quanto riguarda la conoscenza dei tipi generici.

Tramite la *type erasure*, il compilatore Java si sbarazza di molte informazioni sui tipi generici dopo la compilazione. Ciò significa che a runtime un'oggetto non può più conoscere i tipi generici specificati dal programmatore. Per esempio, istanze di `ArrayList<String>` e `ArrayList<Integer>` saranno indistinguibili alla JVM.

Apache Flink cerca di ricostruire queste informazioni nel momento in cui l'applicazione viene preparata all'esecuzione, e le memorizza in oggetti di classe `TypeInformation` (una classe degli *internals* di Flink). Gli oggetti `TypeInformation` vengono dunque associati agli operatori della topologia e agli oggetti che viaggiano al suo interno. In questo modo Flink può dispporre di tali informazioni in maniera contestuale, all'atto dell'esecuzione della topologia.

Flink opera dunque una vera e propria inferenza per determinare i tipi. Nella maggior parte dei casi Flink riesce a recuperare tutte le informazioni da sé, ma questo meccanismo ha dei limiti e a volte Flink ha bisogno della "cooperazione" del programmatore finale, che può "suggerire" a Flink le informazioni mancanti usando le API del framework [4]).

[4]Per maggiori informazioni, vedere le sezioni "Type Hints" e "Creating a TypeInformation or TypeSerializer" della documentazione ufficiale di Flink: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/datastream/fault-tolerance/serialization/types_serialization/

Concludiamo fornendo qualche dettaglio in più rispetto al framework di serializzazione proprietario di Apache Flink.

Flink gestisce in maniera distinta la serializzazione/deserializzazione degli oggetti, a seconda delle categorie sopra elencate. In pratica, Flink dispone di più strategie di serializzazione e sceglie la più conveniente in base alle informazioni contenute nell'istanza `TypeInformation` dell'oggetto che si trova a gestire:

- Per i POJO Java, viene impiegato il cosiddetto `PojoSerializer`, un serializzatore proprietario molto efficiente; Flink suggerisce pertanto che gli oggetti ricadano prevalentemente in questa categoria. Nei casi di POJO in cui il serializzatore proprietario non sia utilizzabile, anche Flink (come Storm) adotta la libreria Kryo, che offre comunque buone performance di serializzazione/deserializzazione.

- Tutte le classi che non vengono identificate da Flink come POJO ricadono nella categoria di classi generali Java. Flink tratta queste classi come se fossero delle *black box* e non è pertanto in grado di accedere al loro contenuto. Queste classi vengono sempre serializzate/deserializzate con la libreria Kryo. Non tutti i tipi di oggetti ricadenti in questa categoria sono gestiti senza problemi da Kryo (e quindi da Flink); in questi casi, però, è possibile registrare serializzatori Kryo aggiuntivi [5].

- Gli oggetti di tipo *Value* di Flink, come anticipato, definiscono la loro stessa logica di serializzazione; in questi casi, dunque, Flink non usa serializzatori di altro tipo o librerie come Kryo, ma delega completamente l'operazione alla classe dell'oggetto. Ricadono in questa categoria i tipi di dato primitivi, per i quali Flink fornisce già i corrispettivi oggetti *wrapper* (`ByteValue`, `IntValue`, `DoubleValue`, `StringValue`, `BooleanValue`, etc...).

Possiamo dunque concludere che, maggiori informazioni ha il framework sui tipi di dato scambiati durante l'esecuzione distribuita dell'applicazione, maggiore è la possibilità di effettuare ottimizzazioni.

Disporre di uno strato di supporto alla *type-safety* direttamente all'interno del framework consente di migliorare la serializzazione/deserializzazione dei dati, adottando la modalità più efficiente, e risparmiare (nella maggior parte dei casi) al programmatore

[5] Per maggiori informazioni, si rimanda alla documentazione ufficiale della libreria Kryo: https://github.com/EsotericSoftware/kryo#implementing-a-serializer

finale l'onere di occuparsi dei vari aspetti legati alla serializzazione, come la scelta del framework e le peculiarità nel suo utilizzo.

3.1.3 API *low-level* vs *high-level*

In questa ultima sezione, riguardante gli aspetti peculiari dei framework, illustreremo una caratteristica riguardante la modalità di programmazione di un'applicazione di stream-processing.

Praticamente ogni framework preso in analisi espone due tipi di API: una API di basso livello (o *Low-level* API) e una API di alto livello (o *High-level API*).

Entrambe le API sono pensate per essere utilizzate dal programmatore finale, ma hanno scopi e ragion d'essere differenti.

Le *Low-level* API sono quelle usate sostanzialmente dai provider di stream processing realizzati per il nostro framework; infatti, queste interfacce di programmazione consentono di specificare operazioni di trasformazione totalmente arbitrarie sui dati; l'intera logica implementativa è lasciata al programmatore finale, che può al limite decomporla in più nodi di processamento della topologia; i framework di stream processing vedono i nodi di processamento così definiti come delle *black box*, non mettono in atto nulla rispetto alla logica in essi contenuta, al più si limitano a considerare il tipo di oggetti in ingresso e in uscita a questi nodi e a gestire la loro serializzazione.

Le *High-level* API, diversamente, offrono al programmatore di applicazioni un insieme piuttosto definito di operazioni di trasformazione sugli *stream* di dati, con lo scopo di facilitare la scrittura di applicazioni di stream processing; queste operazioni sono più o meno comuni a tutti i framework, ma non sempre hanno lo stesso nome. In ogni caso, tutti i framework forniscono operazioni per mappare, partizionare, filtrare, confrontare e aggregare i dati che fluiscono negli stream. Citiamone giusto alcune che hanno lo stesso nome su tutti i framework:

- ***Map*(*function***): si ottiene uno stream avente lo stesso numero di elementi di quello originale, la funzione passata come argomento trasforma e restituisce un singolo elemento.

- ***FlatMap*(*function***): si ottiene uno stream avente un numero di elementi potenzialmente diverso da quello originale, la funzione passata come argomento può

trasformare e restituire zero, uno o più elementi per ogni elemento processato.

- ***Filter(function)***: si ottiene uno stream avente un numero di elementi potenzialmente diverso da quello originale, la funzione passata come argomento verifica le caratteristche di ciascun elemento e restituisce un booleano per indicare se tale elemento deve essere mantenuto o scartato, lo stream risultante contiene solo gli elementi mantenuti.

Quello che in generale si è rilevato è che le API di basso livello sono state le prime, cronologicamente, a essere state messe a disposizione da ciascun framework. L'aspetto più interessante, però, è il fatto che in ogni framework le API di alto livello siano state realizzate sulla base delle API di basso livello: ogni operazione logica di alto livello corrisponde a un nodo di processamento "preconfezionato" di basso livello.

Vediamo ora, a scopo esemplificativo, il parallelismo che c'è tra le API di alto e basso livello nel caso di Apache Samza. La tabella mostrata nella Figura 3.6 elenca le entità/operazioni principali esposte da ciascuna API, mettendole in relazione ai concetti generali dello stream processing identificati in questo progetto.

		Apache Samza	
		Low-level API	High-level API
Context	*local*	`LocalApplicationRunner`	`LocalApplicationRunner`
	remote	`RemoteApplicationRunner`	`RemoteApplicationRunner`
Application		`TaskApplicationDescriptor`	`StreamApplicationDescriptor`
Topology	*logical*	-	`OperatorSpecGraph` `OperatorImplGraph`
	physical	`StreamTask`	`StreamOperatorTask`
Source		`InputDescriptor`	`InputDescriptor`
Element		`Object`	`MessageStream<M>`
Processor/Operator		`StreamTask` interface **`process()`** method	`MessageStream` methods
Stream		-	`MessageStream`
Sink		`OutputDescriptor`	`OutputDescriptor`

Figura 3.6: Apache Samza - *Low-level API* vs *High-level API*

Ecco alcune considerazioni a riguardo:

- *Topology (logical)*: quando si utilizzano le *low-level* API di Samza non ha senso parlare di topologia; un'applicazione può definire un solo oggetto di tipo `StreamTask`, che deve quindi compiere tutte le trasformazioni. Se si utilizzano le *high-level* API, il framework mantiene una rappresentazione della "catena" di operatori di alto livello

utilizzati dal programmatore (`Operator[Spec|Impl]Graph`) che poi viene valutata da un singolo oggetto di tipo `StreamOperatorTask`.

- *Stream*: nelle API di basso livello, non c'è un'entità rappresentante questo concetto; in questo caso, lo stream consiste semplicemente nella sequenza di oggetti che flusice nell'oggetto `StreamTask`. Nelle API di alto livello questo concetto è invece rappresentato dalla classe `MessageStream`, che costituisce anche il punto d'accesso agli operatori di alto livello (`map`, `flatMap`, `filter`, etc.).

- *Processor/Operator*: nelle API di basso livello questo concetto corrisponde all'implementazione del metodo `process` della classe `StreamTask`. Nelle API di alto livello questo concetto trova corrispondenza in ciascuno degli operatori di trasformazione messi a disposizione dal framework (`map`, `flatMap`, `filter`, etc.).

Ciascun framework realizza a modo proprio, e secondo la sua architettura, i concetti dello stream processing che abbiamo identificato e gli espone tramite questa duplice API. Quest'ultimo esempio ci ha mostrato come Apache Samza riconduca il funzionamento delle API di alto livello a quelle di basso livello, e abbiamo anche potuto apprezzare con maggior dettaglio la diversa concezione di topologia che Samza ha rispetto agli altri framework.

Concludiamo facendo notare che, questa duplice modalità di programmazione dei framework di stream processing è un aspetto che può riguardare anche SPAF: questa prima versione di SPAF espone esclusivamente API di basso livello, basate sul concetto di *Processor* e riassumibili nella corrispondente gerarchia di classi; tuttavia, è ragionevole pensare a uno sviluppo futuro di SPAF, che si occupi di razionalizzare anche le *High-level* API dei vari framework, per fornire anche in questo caso un'interfaccia di programmazione unificata, ma basata su operatori di alto livello (approfondiremo questa possibilità nel capitolo 4).

3.2 Utilizzi avanzati di SPAF

In questa sezione illustreremo un esperimento di utilizzo avanzato di SPAF, che evidenzia la versatilità e le potenzialità del framework nello sviluppo di applicazioni di stream processing altamente distribuite.

3.2.1 Realizzazione di Super-Topologie

L'esperimento che illustreremo consiste nell'utilizzare più applicazioni SPAF, in maniera contemporanemanente e coordinata, con lo scopo di eseguire in maniera altamente distribuita (chiariemo presto che cosa intendiamo con questo aggettivo) la soluzione a un problema di stream processing.

L'idea principale, che si intende verificare sperimentalmente, si basa sulla decomposizione del problema di stream processing in sotto-problemi, e nella risoluzione di questi ultimi tramite l'utilizzo di più progetti SPAF indipendenti ma in grado di collaborare tra loro. In sostanza, ciascun progetto SPAF definirà una certa topologia logica in grado di risolvere un certo sotto-problema; ogni topologia riceverà dati tramite i connettori previsti da SPAF, gli elaborerà e inviera i dati elaborati nuovamente all'esterno. In questo scenario, il sistema dati a cui si connettono le applicazioni SPAF può essere sfruttato come mezzo di comunicazione tra le topologie dei singoli progetti; è evidente che i sistemi a code di messaggi rappresentano i candidati ideali per questo compito.

In altre parole, vogliamo realizzare una topologia di topologie. Le singole topologie, infatti, possono essere intese come nodi di elaborazione di tipo *black box*, appartenenti a un grafo diretto (normalmente aciclico), in grado di ricevere e inviare dati tramite gli archi che le collegano, quest'ultimi realizzati da diverse code di messaggi. Quanto descritto corrisponde appunto alla definizione di topologia usata in questo progetto, ma si pone a un livello superiore.

Ha etimologicamente senso riferirci alla struttura sopra esposta con il termine di **super-topologia**, dove il prefisso *super-* è da intendere nell'accezione generica di «che sta sopra» (*sùper- in Vocabolario - Treccani* 2022).

Nello schema mostrato in Figura 3.7 viene mostrato come realizzare concettualmente una super-topologia SPAF. Sarà necessario prevedere dei *Source* e *Sink* di "frontiera" (mostrati nei riquadri "sources" e "sinks"), da considerare rispettivamente come input e output dell'intera super-topologia. I nodi di elaborazione della super-topologia (rappresentati con dei tondi rossi) corrisponderanno invece ciascuno a una singola applicazione SPAF a sé stante. Ciascuna delle applicazioni SPAF dovrà definire una propria *Source* e un proprio *Sink*, e potrà essere costituita da una topologia di *Processor* (al momento, solo di tipo lineare). La comunicazione tra i nodi della super-topologia, ossia tra le varie

applicazioni SPAF, potrà avvenire tramite delle code di messaggi opportunamente predisposte tra ciascuna coppia di nodi che devono comunicare tra loro (questo concetto è rappresentato in figura dal dettaglio sugli archi rossi che collegano due topologie). Ovviamente, ciascuna topologia dovrà essere configurata in modo da ricevere e inviare dati alle giuste code di messaggi, siano esse "frontaliere" (nell'esempio in figura, le topologie T1, T2 e T5) o "interne" (T3 e T4).

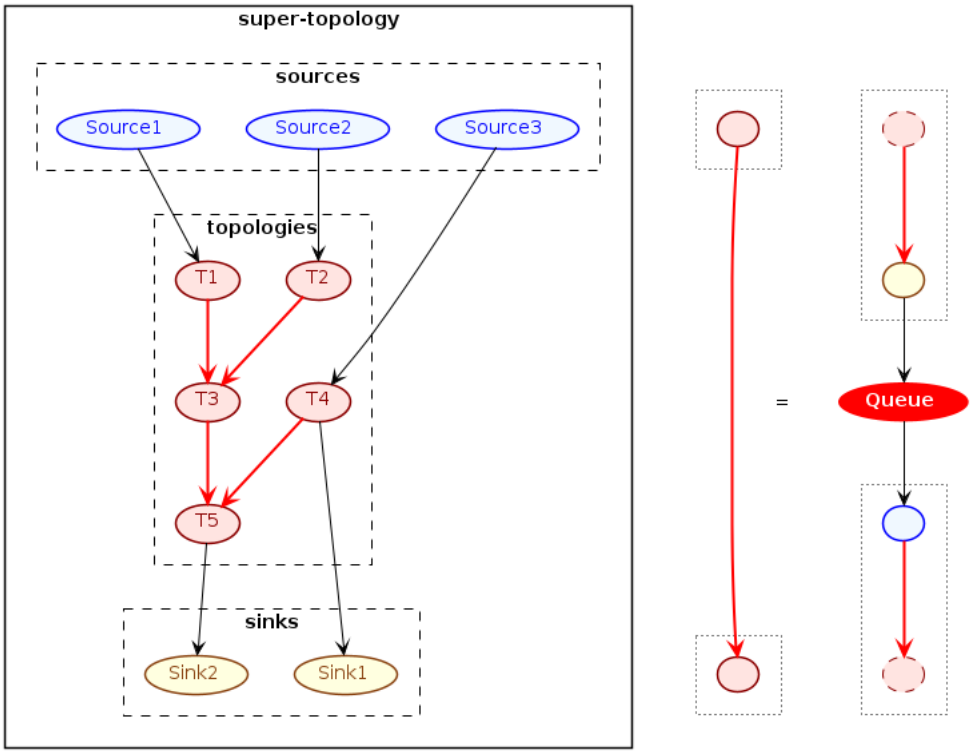

Figura 3.7: Schema di realizzazione di una super-topologia

Questo schema realizzativo merita alcune riflessioni:

- Dal punto di vista logico, questa configurazione consente di definire (super-)topologie di tipo DAG, superando i limiti imposti da questa prima versione di SPAF; all'interno del singolo nodo si è comunque limitati a topologie "lineari", ma questo vincolo viene superato grazie alle funzionalità dei sistemi a code di messaggi, in grado di aumentare il *Fan-in* e il *Fan-out* delle singole topologie.

- Ricordiamo il fatto che Apache Samza utilizza il concetto di sub-topologia, che consiste nell'eseguire l'intera topologia logica di un'applicazione SPAF come unico processo. Se realizzassimo una super-topologia di applicazioni SPAF con provider Samza otterremmo una distribuzione del calcolo confrontabile a quella ottenibile

tramite una singola applicazione sugli altri framework; ci ricondurremmo al concetto di topologia ("super-" elide "sub-").

La possibilità di utilizzare più applicazioni SPAF in maniera coordinata apre inoltre degli scenari interessanti:

- Ciascuna applicazione SPAF, ovvero ciascun nodo della super-topologia, può essere eseguita da un framework di stream processing potenzialmente diverso; ciascuna applicazione può infatti specificare il provider SPAF desiderato indipendentemente dalle altre.

- Dal punto precedente, ne consegue che ciascuna applicazione SPAF, dunque ciascun nodo della super-topologia, può potenzialmente essere eseguita su un cluster di nodi a sé; per questa ragione parliamo di esecuzione "altamente" distribuita.

La prima versione del framework SPAF non consente di gestire le super-topologie a livello di API; ogni applicazione è indipendente dalle altre e il suo ciclo di vita va gestito in maniera manuale. La realizzazione dello schema illustrato richiede piuttosto degli sforzi di tipo sistemistico, volti alla predisposizione dei cluster dei vari framework, alla predisposizione del sistema di messaggistica e alla configurazione delle code di messaggi necessarie, e infine all'avvio coordinato delle singole applicazioni. Qualora l'utilità di questo schema realizzativo risultasse indubbia, sarebbe possibile prevedere la realizzazione di strumenti facilitatori in grado di assolvere in maniera automatica i compiti appena descritti.

Concludiamo testimoniando che il funzionamento delle super-topologie ha trovato conferma da semplici prove empiriche realizzate nell'ambito di questo progetto.

Capitolo 4

Sviluppi Futuri

In questo ultimo capitolo verranno illustrate le principali occasioni di estensione di questa prima versione del framework SPAF e del progetto RAM^{3}S. Gli argomenti verranno raggruppati in base all'area di competenza dei due progetti. In generale, gli sviluppi futuri qui esposti consentirebbero di elevare l'attuale potenza espressiva ed esecutiva del nostro framework astratto, per arrivare a livelli confrontabili con i framework di stream processing veri e propri analizzati in questo progetto e consentire un loro più completo utilizzo.

4.1 Sviluppi Futuri per SPAF

Ora illustreremo gli sviluppi futuri riguardanti principalmente il framework realizzato in questo progetto. Per ciascun sviluppo futuro verrà fornita una motivazione e verrà abbozzata una possibile idea realizzativa.

4.1.1 Topologie di tipo DAG

Un limite evidente di questa prima versione del framework, indotto dalle ipotesi semplificative, è quello di poter specificare solo topologie di tipo "lineare". Tutti i framework di stream processing presi in cosiderazione permettono invece di definire la logica applicativa in termini di un DAG, offrendo così maggiore potere espressivo per la risoluzione del problema di stream processing e maggior controllo sulla decomposizione dell'appplicazione e sulla conseguente esecuzione in modalità distribuita.

Uno sviluppo futuro, piuttosto prioritario, di SPAF si dovrebbe occupare di realizzare il supporto alle topologie di tipo DAG. Questo lavoro dovrebbe coinvolgere in maniera piuttosto limitata le API pubbliche del framework, che dovrebbero meglio esprimere "verbalmente" il fatto che sarà possibile definire più *Source*, più *Sink* e più predecessori per ciascuno nodo della topologia, ma coinvolgerà in maniera decisamente più seria il codice di rappresentazione interna della topologia poiché non sarà più sufficiente memorizzare i nodi in una lista, ma servirà una struttura più evoluta.

Nella Sezione 2.3.5 è stata presentata un'occasione di refactoring del codice che gestisce la creazione e la rappresentazione della topologia in SPAF; le riflessioni contenute in tale sezione possono costituire un valido punto di partenza per la realizzazione di questo sviluppo futuro.

4.1.2 Computazione *stateful*

In questa prima versione di SPAF si è completamente tralasciato il tema della memorizzazione dello stato intermedio della computazione che avviene in un'applicazione di stream processing. Questa lacuna del framework, anche in questo caso indotta dalle ipotesi semplificative, si traduce in un'importante limitazione per il programmatore finale che, all'atto pratico, non può disporre di uno strumento ben integrato nel framework per memorizzare e recuperare i risultati parziali dell'elaborazione di uno stream.

Al momento, l'unica possibilità è infatti quella di usare direttamente nei *Processor* SPAF i supporti per la memorizzazione (ad esempio, file system) o librerie di terze parti (ad esempio, accesso a database) per salvare e recuperare lo stato della computazione. Così facendo, però, ci si può scontrare con problemi di performance e con difficoltà indotte dalla natura distribuita dell'applicazione.

In sostanza, il supporto alla computazione *stateful* è un tema complesso e richiede uno sforzo implementativo importante. Per questa ragione, uno sviluppo futuro di SPAF potrebbe consistere nella realizzazione di uno strato di supporto aggiuntivo alla computazione *stateful*, che si ponga come obiettivo la razionalizzazione dei concetti e delle pratiche esposte dai vari framework relativamente a questa funzionalità.

4.1.3 *High-level* API

Nella Sezione 3.1.3 abbiamo illustrato come ogni framework di stream processing esponga al programmatore finale due tipi di API: una di basso livello e una di alto livello.

Questa prima versione di SPAF mette a disposizione dello sviluppatore di applicazioni di stream processing solo una API di basso livello. Le API di basso livello sono certamente potenti perché permettono di specificare della logica di elaborazione arbitraria sugli stream; tuttavia, richiedono uno sforzo realizzativo maggiore al programmatore e non offrono nulla per la soluzione di problemi ricorrenti nella manipolazione degli stream.

Le API di alto livello dei framework presi in esame forniscono un vasto insieme di *stream operator* in grado di risolvere problemi ricorrenti, quali: la mappatura di valori in altri valori, il partizionamento dei dati in base alle loro caratteristiche, il filtraggio di taluni valori, la possibilità di definire operazioni di aggregazione sui dati, il *join* tra più stream, etc. I framework adottano un "vocabolario" pressoché comune per riferirsi ai vari operatori che realizzano le funzioni appena elencate.

È evidente come le API di alto livello aumentino notevolmente il potere espressivo di un framework di stream processing e lo rendano anche più accessibile ai programmatori alle prime armi.

Un conveniente sviluppo futuro per SPAF sarebbe dunque quello di realizzare una API di alto livello che raccolga le operazioni ricorrenti più importanti e disponibili nei vari framework di stream processing. Anche in questo caso è necessario effettuare un lavoro di traduzione tra le API di SPAF e quelle del framework sottostante. Si apre però una possibilità interessante: poiché SPAF può accedere sia alle *High-level* che alle *Low-level* API del framework sottostante, si potrebbe pensare di realizzare, nelle API di alto livello di SPAF, uno *stream operator* particolare e definire la sua logica implementativa nei vari provider di stream processing, utilizzando le API di basso livello dei framework per definire come debba avvenire il calcolo, eventualmente distribuito, di tale operatore.

4.1.4 *Type-safeness* estesa

Le API di SPAF fanno ampio uso dei *Java Generics* per consentire al programmatore finale di disporre della *type-safety*. La *type-safety* può essere presente a vari livelli nel processo produttivo di software, dalla scrittura del codice alla sua esecuzione, passando

per un'eventuale fase di compilazione (cosa che in Java avviene). Questa prima versione di SPAF si è interessata alla *type-safety* con lo scopo principale di agevolare la stesura del codice; l'obiettivo è quello di ottenere dell'aiuto da parte dell'IDE, tramite suggerimenti del codice e segnalazione di eventuali errori a tempo di compilazione. Si è raggiunto un primo risultato, seppure limitato al contesto di definizione del singolo *Processor* SPAF.

I risultati ottenuti dalla *type-safety* lato API non hanno però aclun effetto lato provider (cioè lato SPI). Nel codice dei vari provider non c'è alcuna consapevolezza rispetto ai tipi generici usati lato API, in pratica tutto viene trattato come `Object`; sta dunque al programmatore di applicazioni assicurarsi di specificare i giusti tipi di dato per l'input e l'output di ciascun *Processor* (e dei *Source* e *Sink*), poiché eventuali incongruenze possono essere scoperte solo a tempo di esecuzione.

Oltre a facilitare lo sviluppo del codice, abbiamo visto (nella Sezione 3.1.2) come la *type-safety* abbia delle implicazioni anche sull'esecuzione dell'applicazione di stream processing, per ciò che riguarda la serializzazione dei dati che viaggiano nella topologia. Questo aspetto è stato gestito parzialmente nella prima versione di SPAF, che affronta esclusivamente il problema della serializzazione per i *Source* e i *Sink*.

Uno sviluppo futuro su questi temi consentirebbe di ottenere un framework astratto più raffinato, capace di facilitare ulteriormente lo sviluppo di applicazioni e di predisporre la loro ottimizzazione su taluni framework di stream processing.

4.1.5 Connettori per JMS

Nella Sezione 2.3.3 abbiamo illustrato approfonditamente la realizzazione dei connettori SPAF per i *message broker* Apache Kafka e RabbitMQ. Per ciascuno dei due sistemi si sono realizzati un descrittore di connettore e tanti connettori concreti quanti sono i provider di stream processing implementati. Si sono pertanto realizzati in totale 10 sotto-progetti, 2 per i descrittori di connettori e 8 per i connettori concreti.

La difficoltà realizzativa di un connettore risiede principalmente lato provider, dunque nei connettori concreti, e dipende fortemente dalle peculiarità del framework di stream processing integrato, invece che dal sistema dati che si intende connettere (a titolo di cronaca, la realizzazione dei connettori concreti per Apache Spark è risultata piuttosto

impegnativa, poiché ci si è dovuti confrontare con aspetti relativi all'inizializzazione e alla serializzazione degli oggetti rappresentanti le connessioni ai *message broker*[1]).

La realizzazione di un descrittore di connettore è invece relativamente semplice, poiché si tratta essenzialmente di decidere quali configurazioni rendere disponibili per il sistema dati che si intende connettere e realizzare la rappresentazione a oggetti equivalente a tali configurazioni.

Con queste premesse, risulta evidente che la realizzazione di connettori per nuovi sistemi di tipo *message broker* possa risultare un compito oneroso.

Una strada possibile per ridurre drasticamente il numero di connettori da realizzare, sia di tipo descrittivo che concreto, per sistemi di tipo *message broker* è sfruttare il framework di astrazione JMS (*Jakarta (o Java) Messaging Service*).

In poche parole, *JMS* è un framework di astrazione per l'utilizzo di sistemi a code di messaggi, quali appunto Apache Kafka e RabbitMQ. Per fare un parallelismo, potremmo dire che «SPAF sta ai framework di stream processing come JMS ai sistemi a code di messaggi».

JMS offre un'interfaccia di programmazione unificata per le applicazioni che intendono utilizzare un *message broker*, questo semplifica il codice client per ricevere e inviare messaggi e rende l'applicazione indipendente dal sistema sottostante. Tutto ciò è però possibile a patto che esistano sistemi che mettano a disposizione il rispettivo provider JMS, infatti JMS usa il pattern Java SPI. Fortunatamente, la maggior parte dei sistemi a code di messaggi di tipo enterprise (tra cui, Apache Kafka[2] e RabbitMQ) ha realizzato il rispettivo provider ufficiale JMS[3].

Uno sviluppo futuro interessante per il nostro framework potrebbe dunque prevedere la realizzazione di un singolo connettore per JMS. Parliamo di singolo connettore, che però dovrà prevedere i seguenti sotto-progetti: un descrittore di connettore, che preveda le

[1] Si invita il lettore a consultare il codice del progetto `spaf-spark-streaming-connectors-kafka`

[2] Il provider JMS per Apache Kafka è stato realizzato dall'azienda Confluent, che realizza una versione commerciale dell'omonimo *message broker*: https://docs.confluent.io/platform/current/clients/kafka-jms-client/index.html

[3] Alcuni di questi provider sono elencati nella pagina Wikipedia relativa a JMS https://en.wikipedia.org/wiki/Jakarta_Messaging#Provider_implementations

configurazioni tipiche gestite da JMS stesso; un connettore concreto per ciascun provider di stream processing, che adatti il client JMS (pressocché identico in ogni provider) alle pecoliarità del framework integrato.

Tramite questo connettore aggiuntivo, SPAF risulterebbe potenzialmente collegabile a un qualsiasi sistema a code di messaggi. Qualora il *message broker* che si intende utilizzare non disponesse del relativo provider JMS, è sempre possibile realizzare tale provider in autonomia poiché JMS è una specifica aperta[4].

Per completezza, concludiamo riportando alcune riflessioni svolte durante lo sviluppo di questa prima versione del framework e che hanno svolto un ruolo di deterrenza nell'implementare immediatamente il connettore per JMS:

- Non tutti i framework di stream processing supportano direttamente JMS: l'unico che ha un supporto ufficiale è Storm (*Storm JMS Integration* 2022); per Spark[5] e Flink[6] esistono alcune librerie Open Source, ma non tutte offrono un supporto completo e in generale non sono molto popolari; per Samza non abbiamo trovato alcuna libreria.

- I *message broker* apparentemente più diffusi a oggi sono, in ordine: Kafka, RabbitMQ, ActiveMQ (*Stack Overflow Trends* 2022); per questi *message broker* esiste un supporto "nativo" (non JMS) praticamente ufficiale da parte di tutti framework di stream processing considerati, tali librerie realizzano un supporto avanzato rispetto a quello che si potrebbe ottenere con un'astrazione come JMS.

4.1.6 Configurabilità della topologia fisica

Alcuni framework di stream processing considerati consentono di controllare il grado di parallelismo nell'esecuzione della topologia logica. Questo parametro va a determinare la struttura della topologia fisica corrispondente, in particolare determina il numero di nodi fisici (i cosiddetti *Task*) da prevedere per lo svolgimento dei compiti specificati in unico

[4]Informazioni a riguardo sono disponibili nel sito ufficiale della specifica: https://javaee.github.io/jms-spec/

[5]Repository della libreria JMS spark receiver: https://github.com/tbfenet/spark-jms-receiver

[6]Repository della libreria Apache Flink Connector for JMS: https://github.com/miwurster/flink-connector-jms

nodo della topologia logica (cioè, il *Processor* SPAF). Il livello di granularità per il quale può essere specificato il grado di parallelsimo può spaziare da: livello di sistema (cioè, per qualsiasi applicazione); livello di singola applicazione (i cosiddetti *Job*); livello di singolo operatore.

I framework non sono concordi nell'esporre la configurazione di questo aspetto poiché, come abbiamo visto, ognugno di questi ha un modello di esecuzione differente:

- Apache Storm, per esempio, permette di specificare il livello di parallellismo desiderato a livello del singolo *Spout* o *Bolt*, in termini di numero di thread (chiamati *executor*) da allocare all'avvio dell'applicazione (il numero di thread può variare a tempo di esecuzione) (user2720864 2013). Come abbiamo visto nella Sezione 3.1.1, ciò è possibile utilizzando i metodi delle API che consentono di specificare il parametro `parallelismHint`.

- Apache Flink consente di specificare il grado di parallelismo per tutti i livelli di granularità sopra elencati (*Parallel Execution | Apache Flink* 2022). Per specificarlo a livello di sistema, cioè per tutti gli *execution environment* definibili in un cluster Flink, si può specificare la proprietà `parallelism.default` nel file `./conf/flink-conf.yaml`. Il livello di parallelismo del singolo *execution environment* può invece essere specificato invocando il metodo `setParallelism()` sull'oggetto corrispondente, questo agirà su tutti i *data source*, *data sink* e *operator* Flink. Infine, a livello di singolo *operator*, *data source* e *data sink* è possibile invocare il metodo `setParallelism()` della classe `Transformation`.

I provider realizzati per questa prima versione di SPAF non si avvalgono della possibilità di modificare il grado di parallelismo a livello del singolo operatore. D'altra parte, tale informazione non potrebbe che essere impostata in maniera uguale per tutti gli operatori e in maniera totalmente arbitraria, poiché le API di SPAF non consentono di specificare tale informazione.

Un possibile sviluppo futuro di SPAF, che consentirebbe di effettuare il *fine tuning* delle performance a tempo di esecuzione, potrebbe essere quello di esporre a livello delle API la possibilità di specificare il grado di parallelismo desiderato per ciascuna *Source*, *Sink* e *Processor* SPAF.

Un possibile mezzo realizzativo per esporre questa possibilità potrebbe essere quello delle Java *Annotation*. Si potrebbe prevedere una annotazione consultabile a tempo di esecuzione, tramite *reflection*, che porti con sé il valore desiderato del grado di parallelismo per il nodo della topologia logica sul quale è definita.

```
@Retention(RUNTIME)
@Target(TYPE)
public @interface ExecutionHints {
   int parallelism() default 1;
}
```

Una annotazione così definita potrebbe essere dunque specificata a livello della classe di ciascun *Processor*. Qui sotto un esempio di come verrebbe utilizzata per il *Processor* creato per il rilevamento dei volti.

```
@ExecutionHints(parallelism = 5)
public class FaceDetectionProcessor implements Processor<...> {

   @Override
   public void init() { ... }

   @Override
   public void process(...) { ... }

}
```

Oppure, un'alternativa più semplice potrebbe essere quella di prevedere dei metodi sovraccarichi a livello di API in grado di accettare anche il parametro indicante il grado di parallelismo.

```
public class Topology {
    ...

    public <KIn, VIn, KOut, VOut> Topology addProcessor(
        String name,
        Processor<KIn, VIn, KOut, VOut> processor,
        String parentName
    ) { ... }

    public <KIn, VIn, KOut, VOut> Topology addProcessor(
        String name,
        Processor<KIn, VIn, KOut, VOut> processor,
        String parentName,
        int parallelismHint
   ) { ... }

   ...
```

4.2 Sviluppi Futuri per RAM³S

In quest'ultima sezione, illustreremo invece gli sviluppi futuri che riguardano direttamente RAM³S nell'idea di utilizzarlo come generatore di progetti SPAF. Anche in questo caso, riporteremo i benefici che otterremmo da tali sviluppi e si forniranno spunti per le possibili realizzazioni.

4.2.1 Automazione Deployment su cluster Samza

Nella versione precedente di RAM³S il deployment su cluster Samza avveniva in maniera manuale: per eseguire l'applicazione in modalità distribuita era necessario avviare manualmente il jar corrispondente su ciascuna delle macchine del cluster. In realtà, Apache Samza si affida a YARN per effettuare questa operazione in maniera automatizzata e programmabile. Apache YARN fa parte del progetto Hadoop, è una piattaforma che consente la gestione delle risorse di calcolo in un cluster e il loro utilizzo per l'esecuzione distribuita delle applicazioni utente.

All'atto pratico, la procedura di deployment per Apache Samza è diversa da quella degli altri framework di stream processing presi in esame. Apache Samza non consente di effettuare il deployment di un'applicazione tramite la semplice distribuzione del pacchetto *jar* risultante dalla build; serve invece un archivio di tipo *tar*, avente una certa struttura interna, da sottoporre a YARN.

Qui sotto un esempio di come deve risultare la struttura dell'archivio:

```
samza-job-name-folder
├── bin
│   ├── run-app.sh
│   ├── run-class.sh
│   └── ...
├── config
│   └── application.properties
└── lib
    ├── samza-api-0.14.0.jar
    ├── samza-core_2.11-0.14.0.jar
    ├── samza-kafka_2.11-0.14.0.jar
    ├── samza-yarn_2.11-0.14.0.jar
    └── ...
```

Un archivio con tale struttura può essere generato adoperando un plugin per Maven nel POM del progetto; il plugin in questione si chiama `maven-assembly-plugin` e va oppportunamente configurato[7]. Tramite il plugin sarà possibile creare l'archivio tramite il solito comando per la build del progetto `mvn clean package`.

L'archivio così generato conterrà al suo interno lo script `bin/run-app.sh`, che può essere utilizzato per inviare l'applicazione a YARN e metterla così in esecuzione sul cluster[8].

Uno sviluppo futuro di RAM^{3}S potrebbe prevedere la generazione di un file POM pre-configurato per la creazione dell'archivio *tar* e potrebbe mettere a disposizione degli script proprietari per semplificare l'operazione di deployment sul cluster Samza (o meglio, sul cluster YARN).

4.2.2 Generazione di progetti e DSL

L'ultimo sviluppo futuro che vogliamo proporre riguarda l'attività di creazione di nuovi progetti di stream processing. Al momento, per creare un nuovo progetto, è necessario eseguire manualmente una serie di attività che sono potenzialmente identiche per progetti distinti: creazione della cartella di progetto; scrittura del file POM di progetto; scrittura del *boilerplate code* di una tipica applicazione SPAF; scelta del provider di stream processing e del provider di connettori; creazione dei file di configurazione dell'applicazione; procedura di deployment su cluster.

La community open-source ha sviluppato diversi strumenti per automatizzare il genere di attività necessarie a predisporre un nuovo progetto software.

Lo stesso Maven mette a disposizione il concetto di *Archetype*: in breve, un *Archetype* è uno strumento per definire dei modelli (*template*) di progetti Maven (la definizione di "archetipo" è infatti quella di «Primo esemplare, modello» (*archètipo in Vocabolario - Treccani* 2022)). Un *Archetype* Maven consente agli autori di librerie e framework di creare modelli di progetti Maven per gli utenti, e fornisce agli utenti i mezzi per generare

[7]Per un'opportuna configurazione del plugin `maven-assembly-plugin`, vedasi l'esempio nel repository ufficiale di Apache Samza: https://github.com/apache/samza-hello-samza/blob/master/pom.xml

[8]Per dettagli sull'utilizzo di tale script si faccia riferimento alla documentazione ufficiale di Apache Samza a riguardo: https://samza.apache.org/learn/documentation/latest/deployment/yarn.html

versioni parametrizzate di quei modelli di progetto (*Maven - Introduction to Archetypes* 2022).

Un altro strumento molto interessante, più potente degli *Archetype* Maven, e che si può prestare ad assolvere i compiti sopra descritti è Yeoman[9]. Yeoman nasce come generatore di progetti di tipo *web application*, ma la sua struttura consente in realtà di realizzare il cosiddetto *scaffolding* (un termine italiano equivalente potrebbe essere l'"imbastitura") di progetti la cui natura è arbitrariamente gestibile dall'autore di librerie o framework. In poche parole, Yeoman ruota attorno al concetto di *generator*: un plugin in grado di essere eseguito tramite il comando `yo` (la CLI (*Command Line Interface*) di Yeoman), per imbastire progetti completi, o parti significative di questi.

Questo genere di strumenti esprime però la sua massima potenza nel caso in cui vengano affiancati da un linguaggio in grado di descrivere in maniera dichiarativa la struttura specifica del progetto che si intende generale.

In pratica, stiamo ipotizzando la realizzazione di un DSL (*Domain Specific Language*) per la descrizione ad alto livello di progetti SPAF.

Per illustrare questa idea, è più facile seguire un esempio pratico.

Supponiamo che fosse possibile descrivere in modo dichiarativo la struttura della topologia logica che vogliamo realizzare nella nostra applicazione SPAF.

Prendiamo come esempio la topologia illustrata nella Figura 4.1.

La topologia illustrata in figura potrebbe rappresentare un'evoluzione dell'attuale applicazione di esempio per il riconoscimento dei volti. In questa nuova topologia sarebbe prevista la possibilità di alimentare l'applicazione di stream processing con nuovi volti da riconoscere a tempo di esecuzione (tramite il *Source* denominato *Suspects Photos*); i volti così forniti in input verrebbero ricercati dal processore *Face Recognition* nelle immagini ricevute in input dall'altra sorgente *Input Photos*; l'output del processore di riconoscimento dei volti (una struttura a oggetti) potrebbe dunque essere utilizzata dagli ultimi due processori, *Face Marking* e il nuovo *Alert Generator*, che si occuperebbero di disegnare nell'immagine originale dei riquadri attorno ai volti rilevati (colorandoli in maniera condizionale rispetto all'avvenuto riconoscimento o meno) e di generare una nuova struttura a oggetti che rappresenti un allarme di avvenuta rilevazione sospetto; gli output di questi

[9]Sito ufficiale di Yeoman: https://yeoman.io/

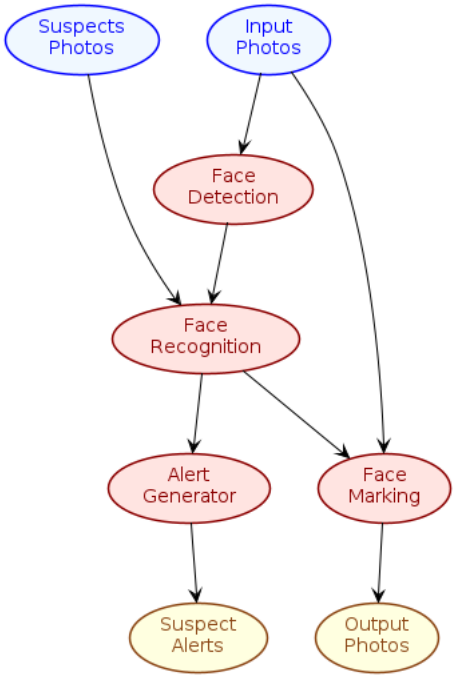

Figura 4.1: Topologia di un'applicazione di individuazione di sospetti

due processor vengono dunque spediti presso due destinazioni destinte, rispettivamente *Output Photos* e *Suspect Alerts*.

Al netto della logica implementativa di ciascun processore appena descritto, vorremmo poter definire la struttura di questa topologia in modo dichiarativo. In realtà, ciò è già avvenuto all'atto della scrittura di questa sezione poiché il diagramma mostrato in figura è stato realizzando utilizzando il linguaggio PlantUML, dell'omonimo strumento open-source[10] per la realizzazione di diagrammi UML (si è "abusato" del diagramma dei casi d'uso per rappresentare la nostra topologia; dopotutto, l'importante è ottenere un DAG).

La topologia appena mostrata potrebbe dunque essere descritta con un linguaggio ispirato a PlantUML. Il listato sottostante riporta un'idea prototipale di DSL:

```
source InputPhotos
source SuspectsPhotos

processor FaceDetection
processor FaceRecognition
processor FaceMarking
processor AlertGenerator

sink OutputPhotos
sink SuspectAlerts
```

[10]Sito ufficiale di PlantUML: https://plantuml.com/

```
11
12 InputPhotos --> FaceDetection
13 FaceDetection --> FaceRecognition
14 SuspectsPhotos --> FaceRecognition
15 FaceRecognition --> FaceMarking
16 InputPhotos --> FaceMarking
17 FaceMarking --> OutputPhotos
18 FaceRecognition --> AlertGenerator
19 AlertGenerator --> SuspectAlerts
```

Un interprete per un linguaggio di questo tipo potrebbe prendere in input un file contenente la descrizione della topologia e invocare il generatore di progetti in modo che produca un *boilerplate code* più avanzato di quello ipotizzato a inizio sezione; oltre al *boilerplate code*, il generatore potrebbe predisporre le classi che dovranno accogliere il codice implementativo dei *Processor* definiti nella descrizione.

```
...

Topology topology = new Topology()
   .addSource("InputPhotos", inputPhotosSource)
   .addSource("SuspectsPhotos", suspectsPhotosSource)
   .addProcessor("FaceDetection", new FaceDetectionProcessor(), "InputPhotos")
   .addProcessor("FaceRecognition", new FaceRecognitionProcessor(config),
      "SuspectsPhotos", "FaceDetection")
   .addProcessor("FaceMarking", new PersonFaceMarkingProcessor(),
      "InputPhotos", "FaceRecognition")
   .addProcessor("AlertGenerator", new AlertGeneratorProcessor(),
      "FaceRecognition")
   .addSink("SuspectAlerts", suspectAlertsSink, "AlertGenerator")
   .addSink("OutputPhotos", outputPhotosSink, "FaceMarking");

...
```

Il linguaggio appena illustrato potrebbe essere esteso in modo da consentire la definizione dei tipi di dato gestiti da ciascun nodo della topologia, dunque di specificare in quali stream (rappresentati dai simboli -->, nel DSL) viaggiano tali tipi di dato.

Bibliografia

archètipo in Vocabolario - Treccani (2022). Istituto della Enciclopedia Italiana fondata da Giovanni Treccani S.p.A. URL: https://www.treccani.it/vocabolario/archetipo/ (visitato il 01/03/2022).

Berni, Mila (2021). «Inclusione di Apache Samza e Kafka nel framework RAM3S». Alma Mater Studiorum - Università di Bologna.

Builder (2022). Refactoring Guru. URL: https://refactoring.guru/design-patterns/builder (visitato il 14/02/2022).

Concepts (2022). Apache Storm. URL: https://storm.apache.org/releases/2.3.0/Concepts.html (visitato il 07/02/2022).

Deitel, Harvey M. e Paul J. Deitel (2006). «Java - Tecniche avanzate di programmazione». In: 3ª ed. APOGEO s.r.l. Cap. 8.6, p. 369.

Deployment - Resource Providers - Standalone (2022). Apache Flink. URL: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/deployment/resource-providers/standalone/overview/ (visitato il 07/03/2022).

Deployment - Run on YARN (2022). Apache Samza. URL: https://samza.apache.org/learn/documentation/latest/deployment/yarn.html (visitato il 07/03/2022).

Deutsch, L. Peter (1989). «Design reuse and frameworks in the Smalltalk-80 system». In: *Software Reusability Volume II: Applications and Experience*. A cura di Alan J. Perlis Ted J. Biggerstaff. Addison-Wesley, pp. 57–71.

Doré, Gustave (2013). «La Divina Commedia di Dante Alighieri». In: a cura di Arnoldo Mondadori Editore S.p.A. 1ª ed. Oscar Mondadori; pag. 90.

Flink Architecture | Apache Flink (2022). Apache Flink. URL: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/flink-architecture/#tasks-and-operator-chains (visitato il 08/03/2022).

Fluent interface - Wikipedia (2022). Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Fluent_interface (visitato il 08/03/2022).

Gamma, Erich et al. (1994a). «Design Patterns: Elements of Reusable Object-Oriented Software». In: 1ª ed. Addison-Wesley Professional. Cap. 1.6, pp. 26–27.

— (1994b). «Design Patterns: Elements of Reusable Object-Oriented Software». In: 1ª ed. Addison-Wesley Professional. Cap. 1.6, p. 11.

— (1994c). «Design Patterns: Elements of Reusable Object-Oriented Software». In: 1ª ed. Addison-Wesley Professional. Cap. 4, pp. 185–193.

— (1994d). «Design Patterns: Elements of Reusable Object-Oriented Software». In: 1ª ed. Addison-Wesley Professional. Cap. 3, p. 87.

— (1994e). «Design Patterns: Elements of Reusable Object-Oriented Software». In: 1ª ed. Addison-Wesley Professional. Cap. 5, pp. 331–339.

Glue code - Wikipedia (2022). Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Glue_code (visitato il 09/03/2022).

JNDI Overview (2022). Oracle. URL: https://docs.oracle.com/javase/jndi/tutorial/getStarted/overview/index.html (visitato il 10/03/2022).

kryo/README.md at master - EsotericSoftware/kryo (2022). URL: https://github.com/EsotericSoftware/kryo/blob/master/README.md (visitato il 28/02/2022).

Laskowski, Jacek (2021a). *RDD Lineage — Logical Execution Plan.* URL: https://books.japila.pl/apache-spark-internals/rdd/lineage/ (visitato il 26/02/2022).

— (2021b). *Stage - The Internals of Apache Spark.* URL: https://books.japila.pl/apache-spark-internals/scheduler/Stage/ (visitato il 10/03/2022).

lineage - Traduzione del vocabolo e dei suoi composti, e discussioni del forum. (2022). WordReference.com. URL: https://www.wordreference.com/enit/lineage (visitato il 01/03/2022).

Marz, Nathan (2011). *A Storm is coming: more details and plans for release.* Twitter. URL: https://blog.twitter.com/engineering/en_us/a/2011/a-storm-is-coming-more-details-and-plans-for-release (visitato il 10/03/2022).

Maven - Introduction to Archetypes (2022). Apache Maven. URL: https://maven.apache.org/guides/introduction/introduction-to-archetypes.html (visitato il 11/02/2022).

Overview | Apache Flink (2022). Apache Flink. URL: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/datastream/overview/#anatomy-of-a-flink-program (visitato il 08/03/2022).

Overview | Apache Flink (2022). Apache Flink. URL: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/datastream/fault-tolerance/serialization/types_serialization/#supported-data-types (visitato il 08/03/2022).

Parallel Execution | Apache Flink (2022). Apache Flink. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/execution/parallel/ (visitato il 08/03/2022).

Running Topologies on a Production Cluster (2022). Apache Storm. URL: https://storm.apache.org/releases/current/Running-topologies-on-a-production-cluster.html (visitato il 07/03/2022).

Samza - Spark Streaming (2022). Apache Samza. URL: https://samza.apache.org/learn/documentation/0.9/comparisons/spark-streaming.html (visitato il 08/03/2022).

Sax, Matthias J. (2015). *Storm Compatibility in Apache Flink: How to run existing Storm topologies on Flink*. URL: https://flink.apache.org/news/2015/12/11/storm-compatibility.html (visitato il 27/02/2022).

Sconosciuto (2019). «README_JARS.txt». File di testo rinvenuto nel direttorio /home/datalab/local del file system condiviso del cluster denominato "datalab", messo a disposizione dall'Università di Bologna per le attività di ricerca sull'elaborazione dati distribuita.

Serialization (2022). Apache Storm. URL: https://storm.apache.org/releases/2.3.0/Serialization.html (visitato il 28/02/2022).

Spark Streaming - Spark 3.2.1 Documentation (2022). Apache Spark. URL: https://spark.apache.org/docs/latest/submitting-applications.html (visitato il 07/03/2022).

Spark Streaming - Spark 3.2.1 Documentation (2022). Apache Spark. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html%5C#other-points-to-remember (visitato il 07/03/2022).

Spark Streaming - Spark 3.2.1 Documentation (2022). Apache Spark. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations-on-dstreams (visitato il 14/02/2022).

Spark Streaming - Spark 3.2.1 Documentation (2022). Apache Spark. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html#input-dstreams-and-receivers (visitato il 14/02/2022).

Spark Streaming - Spark 3.2.1 Documentation (2022). Apache Spark. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html#transform-operation (visitato il 22/02/2022).

Stack Overflow Trends (2022). Stack Overfow. URL: https://insights.stackoverflow.com/trends?tags=apache-kafka%2Crabbitmq%2Cactivemq%2Camazon-sns%2Cibm-mq (visitato il 03/03/2022).

Storm JMS Integration (2022). Apache Storm. URL: https://storm.apache.org/releases/2.3.0/storm-jms.html (visitato il 07/03/2022).

Streams Architecture - Confluent Documentation (2022). Confluent. URL: https://docs.confluent.io/platform/current/streams/architecture.html#processor-topology (visitato il 14/02/2022).

Streams Concepts - Confluent Documentation (2022). Confluent. URL: https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-processor (visitato il 14/02/2022).

sùper- in Vocabolario - Treccani (2022). Istituto della Enciclopedia Italiana fondata da Giovanni Treccani S.p.A. URL: https://www.treccani.it/vocabolario/super/ (visitato il 01/03/2022).

Szczur, Paweł (2022). *apache flink - Items not groupped correctly - CoGroupByKey - Stack Overflow*. Stack Overfow. URL: https://stackoverflow.com/questions/37473682/items-not-groupped-correctly-cogroupbykey (visitato il 15/01/2022).

Throughput - Wikipedia (2022). Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Throughput (visitato il 17/02/2022).

Topological sorting - Wikipedia (2022). Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Topological_sorting (visitato il 21/02/2022).

Topological sorting - Wikipedia (2022). Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Topological_sorting#Depth-first_search (visitato il 21/02/2022).

TopologyBuilder (Storm 2.3.0 API) (2022). Apache Storm. URL: https://storm.apache.org/releases/2.3.0/javadocs/org/apache/storm/topology/TopologyBuilder.html (visitato il 07/03/2022).

Tuple (Flink : 1.15-SNAPSHOT API) (2022). Apache Flink. URL: https://nightlies.apache.org/flink/flink-docs-master/api/java/org/apache/flink/api/java/tuple/Tuple.html (visitato il 14/02/2022).

Tuple (Storm 2.3.0 API) (2022). Apache Storm. URL: https://storm.apache.org/releases/current/javadocs/org/apache/storm/tuple/Tuple.html (visitato il 14/02/2022).

Understanding latency - Web Performance | MDN (2022). MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Understanding_latency (visitato il 05/02/2022).

user2720864 (5 dic. 2013). *parallel processing - how to tune the parallelism hint in storm - Stack Overflow*. Stack Overfow. URL: https://stackoverflow.com/questions/20371073/how-to-tune-the-parallelism-hint-in-storm (visitato il 02/03/2022).

Postfazione

Il lavoro di tesi presentato in questo elaborato non rappresenta soltanto la conclusione di un percorso di studi affrontato con serietà e passione, anche se svolto in maniera inconsueta (l'autore ha affrontato la Laurea Magistrale in qualità di studente-lavoratore), ma porta con sé una connotazione simbolica molto forte, legata alle difficoltà che l'autore non era riuscito a superare in un precedente tentativo dello stesso percorso, poi abbadonato.

Lo stesso autore, con qualche anno di più, ma soprattutto con una maggiore maturità, porta ora a compimento un percorso intrapreso nuovamente dal principio e lo suggella con un lavoro di tesi che risulta significativo anche nella sua natura, la rinascita.

Nicolò Scarpa