POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Design e implementazione di una streaming platform con strumenti OpenSource

Relatore
Prof.ssa Michela Meo

Correlatore
Dott. Gianluca Perna

Candidato
Mario Inglese

Supervisore Aziendale
Blue Reply
Dott. Matteo Nisi

Anno Accademico 2020-2021

Ai miei sogni perché non
so vivere senza sognare

Sommario

La crescente diffusione dell'IoT sta producendo una notevole quantità di dati per i quali, nella maggior parte dei casi, è necessaria un'analisi in tempo reale. Ne sono un esempio l'ambiente della sicurezza stradale con telematics box pronte ad avvisarci in caso di incidenti per poter avviare protocolli di emergenza, l'ambito bancario dove qualunque transazione necessita di essere tracciata e validata nel minor tempo possibile, l'ambito industriale dove i dati dei vari sensori consentono di comprendere in tempo reale l'andamento della produzione ed evitare possibili guasti. Gli esempi possono essere tanti e, nella maggior parte dei casi, oltre ad elaborare un'enorme quantità di dati è necessario farlo nel minor tempo possibile. Inoltre, queste tecniche di elaborazione da sole non bastano per realizzare un sistema per l'analisi di dati in tempo reale ma bisogna anche possedere un sistema di storage adeguato. Questo sistema deve consentire di memorizzare, aggiornare e interrogare i dati con il più basso ritardo possibile. La vera sfida oggi, è riuscire ad essere pronti a gestire un aumento inatteso del traffico. Da qui, la necessità di realizzare una streaming platform altamente scalabile e performante per soddisfare queste necessità. Questa piattaforma deve consentire di acquisire dati da sorgenti esterne, elaborarli in tempo reale e fornire ottimi risultati nel più breve tempo possibile. Il progetto prevede l'utilizzo di framework di comunicazione (e.g. Kafka) e di strumenti di analytics (e.g. Flink) con lo scopo di realizzare un ambiente altamente affidabile e facilmente scalabile per soddisfare le richieste d'uso. Durante lo svolgimento del progetto si è entrati in una fase di design della piattaforma fino alla realizzazione di una Proof of Concept (PoC) che ha dimostrato l'effettivo utilizzo della piattaforma e l'adattamento della stessa ai requisiti dello use case. L'utilizzo della piattaforma è stato applicato alla cattura e classificazione del traffico dati real-time trasportato da flussi RTP che viene osservato in una rete in una delle seguenti classi: Audio, Low Quality (LQ) Video – 180p, Medium Quality (MQ) Video – 360p, High Quality (HQ) Video - 720p e Screen Sharing. Il sistema sviluppato parte analizzando il traffico dati real-time che, mediante tecniche basate su DPI (deep packet inspection), seleziona solo i flussi RTP. Per ogni flusso vengono estratte delle features che servono come input al classificatore, che con alta precisione è in grado di identificare il tipo di sorgente multimediale. Agire a livello di rete e classificare il traffico dati è il passo più importante verso la gestione efficace del traffico e se la classificazione viene effettuata in tempo reale, si possono intraprendere azioni adeguate per massimizzare la Quality of Experience (QoE) percepita dagli utenti.

Indice

Elenco delle figure

Elenco del Codice

Non sono i soldi che ti devono
cambiare la vita, ma i sogni.

[F. SILVESTRE]

Capitolo 1

Introduzione

1.1 Cosa sono i Big Data?

I Big Data [18] sono un'insieme di dati complessi e più vasti del solito. Questo insieme di dati è così complesso e vasto che un classico software di elaborazione di dati non è capace di gestirlo correttamente. Questi giganteschi volumi di dati possono essere impiegati per trattare problemi aziendali che prima non avremmo potuto gestire. Il concetto di big data è noto come le tre V.

1.1.1 Le tre V dei Big Data

- Volume: Il numero di dati presente è rilevante. Quando si parla di Big Data ci si trova a lavorare con una quantità di dati non strutturati elevata. Potrebbe trattarsi di dati con valore sconosciuto, come ad esempio apparecchiature idonee per sensori, click su una pagina web o traffico real-time all'interno di una rete. Per alcuni potrebbero essere centinaia di terabyte mentre per altri centinaia di petabyte ma in entrambi i casi la quantità è impegnativa.

- Velocità: Per velocità si intende la velocità con cui i dati vengono ricevuti. In alcuni casi però viene considerata anche la velocità con cui si agisce su di essi. Alcuni prodotti smart abilitati all'utilizzo di internet operano in tempo reale e necessitano di azioni effettuate in tempo reale. Un altro caso che richiede valutazioni in tempo reale è quello elencato precedente che riguarda il traffico real-time all'interno di una rete.

- Varietà: Per varietà si intende i diversi tipi di dati a disposizione. I classici tipi di dati erano strutturati mentre con i Big Data non sono strutturati. Quelli strutturati si adeguavano bene ad un database relazionale mentre quelli non strutturati, ad esempio audio e video per rimanere in tema di traffico real-time, hanno bisogno di una successiva elaborazione per supportare i metadati.

1.1.2 Verità e valore dei Big Data

Con il passare del tempo sono comparse altre due V che si sono aggiunte alle precedenti: veridicità e valore. Importante è quanto i dati sono veritieri e se ci si può fare affidamento. Allo stesso modo, è importante conoscere il loro valore perché fino a quando esso risulta sconosciuto il dato è inutile. Recuperare il valore nei Big Data non vuol dire solo esaminarli ma è un vero e proprio processo di scoperta che necessita di utenti aziendali, analisti e dirigenti in grado di identificare modelli, elaborare ipotesi e anticiparne il modo di agire.

1.1.3 Storia dei Big Data

Il concetto di Big Data sembra essere recente e nuovo però le prime quantità di dati si sono intraviste negli anni 60' con l'incremento di database relazionali e la nascita dei primi datacerter. Nel 2015 circa mediante Facebook e Youtube, le persone hanno riconosciuto la rilevante quantità di dati che stavano generando. Essenziali per la crescita e lo sviluppo dei Big Data sono stati framework open-source perché hanno reso la loro elaborazione più semplice e la loro archiviazione più economica. Con il passare degli anni, gli utenti stanno generando una smisurata quantità di dati però non sono soltanto loro a farlo. Con la nascita dell'IoT sono aumenti il numero di dispositivi connessi ad internet e l'avvento del machine learning ha generato ancora più dati. Un altro campo che ha generato una quantità gigantesca di dati è il cloud.

1.1.4 Vantaggi dei Big Data

- Permettono di avere risposte più precise e concrete perché ci sono maggiori informazioni.
- Metodo totalmente diverso nell'esaminare e risolvere problemi.

1.1.5 Casi d'uso dei Big Data

- Manutenzione predittiva: Le informazioni che permettono di predire dei guasti sono racchiusi all'interno dei dati strutturati e dei dati non strutturati. Esaminando queste informazioni si possono predire eventuali problemi prima che succedano e si può effettuare una manutenzione in modo efficiente.
- Frode: Gli scenari di sicurezza sono in costante crescita e non riguardano solo attacchi di hacker criminali. I Big Data consentono di individuare eventuali frodi all'interno dei dati.
- Sviluppo del prodotto: Alcune aziende, ad esempio Netflix, utilizzano i Big Data per anticipare eventuali richieste del cliente. Cercano di predire nuovi prodotti e nuovi servizi sfruttando quelli passati e quelli attuali.
- Esperienza del cliente: I Big Data permettono di ottenere informazioni da siti web, social media e altre fonti per arricchire l'esperienza di iterazione del cliente.

- Innovazione: I Big Data possono fornire un aiuto per aggiornare entità e processi. Questo aiuta a capire di cosa necessitano gli utenti per potergli offrire nuovi prodotti e nuovi servizi.

1.1.6 Sfide dei Big Data

I Big Data, come detto precedentemente, sono ampi. Sono nate tecnologie per archiviare enormi volumi di dati. Non è sufficiente solo archiviarli perché alcuni sono preziosi e bisogna prendersene cura. Dati rilevanti per il cliente devono essere puliti quindi richiedono molto lavoro. Prima di essere utilizzati devono essere preparati e curati. Infatti per questo la tecnologia dei Big Data sta cambiando.

1.1.7 Come funzionano i Big Data

I Big Data mettono a disposizione nuove soluzioni che forniscono nuove opportunità e nuovi modelli di business. Per cominciare occorrono tre azioni chiave:

- Integrare: I Big Data raggruppano dati derivanti da applicazioni e origini diverse. I classici meccanismi di integrazione dei dati non sono in grado di svolgere il loro compito. Vengono richieste nuove tecnologie e nuovi metodi per esaminare i dati.

- Gestire: I Big Data necessitano di spazio di archiviazione per i dati. I dati possono essere archiviati in qualunque forma e una soluzione potrebbe essere il cloud. Il cloud a poco a poco sta ottenendo popolarità perché permette di ampliare le risorse a seconda delle esigenze e supporta le caratteristiche di calcolo.

- Analizzare: Quando i dati vengono analizzati si comprende l'importanza dell'investimento fatto sui Big Data. Ispezionare i dati porta chiarezza e permette di fare nuove scoperte.

1.1.8 Ciclo di vita dei Big Data

Il ciclo di vita dei Big Data è possibile raffigurarlo mediante una catena di processi capace di prendere una vasta quantità di dati in input ed ottenere un sequenza di dati output. Nella figura 1.1 è presente una sequenza di fasi che si occupano di ricavare il valore dei dati riposto in un'architettura BigData.

- Data acquisition: In questa fase vengono raccolti e puliti i dati prima di consegnarli alla fase successiva. Questa fase è fondamentale per un'architettura Big Data e deve consentire di acquisire dati sia di strutture dinamiche sia di enormi quantità di dati conservando bassa la latenza. Per ottenere qualsiasi tipo di informazione, che va dalle strutturate alla non strutturate, è necessario l'utilizzo di opportuni protocolli.

- Data analysis: In questa fase vengono modificati i dati raw ottenuti dalla fase precedente e vengono ricavate le informazioni utili. Questa fase richiede processi di data mining, discovery, information extraction e machine learning.

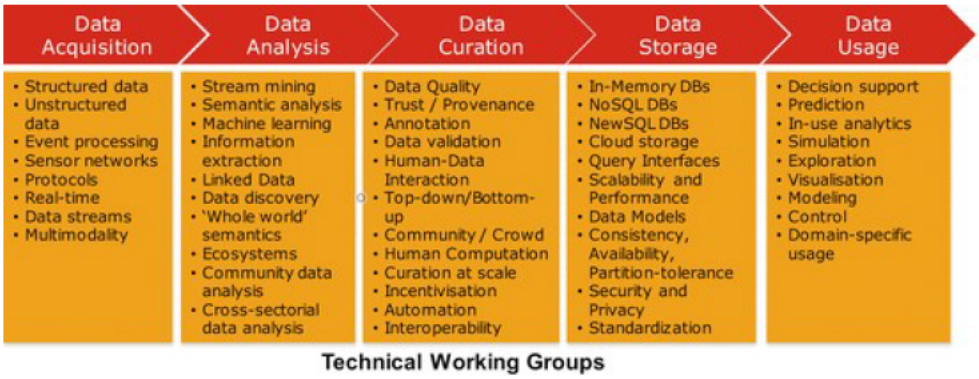

Figura 1.1. Big Data Value Chain [16]

- Data Curation: In questa fase si va ad ottimizzare la qualità dei dati per assicurare determinate caratteristiche.
- Data Storage: In questa fase vengono archiviati e gestiti dati in modo scalabile permettendo alle applicazioni un'accessibilità rapida ai dati. I database relazionali sono stati adoperati per molti anni ma con lo sviluppo della quantità e della complessità dei dati si è andati verso database non relazionali capaci di eliminare questi problemi e garantire una notevole flessibilità.
- Data Usage: In questa fase vengono forniti gli strumenti con cui andare ad integrare le attività di business e i dati analizzati.

1.2 Piattaforme di streaming

Con il passare del tempo le applicazioni Big Data stanno passando da un modello di esecuzione in batch a un modello di esecuzione in tempo reale. Il voler estrarre i dati e i relativi valori in tempo reale, ha portato questo cambiamento. Una volta acquisiti i dati, anche processarli in tempo reale è un aspetto importante per diverse applicazioni perché le consentirebbe di prendere le migliori decisioni possibili. Per elaborare flussi di dati in tempo reale c'è bisogno di una nuova architettura. In "Developing a Real-time Data Analytics Framework For Twitter Streaming Data"[2] è presente un esempio di architettura Big Data per l'analisi di dati in tempo reale. Nella figura 1.2 è presente una piattaforma scalabile e distribuita, che si occupa di esaminare in tempo reale i tweet che vengono prodotti dall'utente secondo per secondo. Si può notare la presenza di tre livelli che sono il data ingestion, il data processing e il data visualization con cui viene realizzata questa piattaforma. La parte di data ingestion viene realizzata da Apache Kafka che mediante degli opportuni producer interroga le API di Twitter per recuperare i tweets. Dopodiché i producers pubblicano sui topic le informazioni desiderate mentre i consumer recuperano queste informazioni e le forniscono al modulo successivo. La lettura e la scrittura sui topic viene amministrata da broker che fanno parte del cluster Kafka mentre lo Zookeerper

dirige i vari broker del cluster. Lo Zookeeper oltre a dirigere i diversi nodi del cluster, se dovesse verificarsi il malfunzionamento di un broker è grado di risolvere il problema per assicurare il regolare funzionamento. Il prossimo modulo, composto da Spark, si oc-

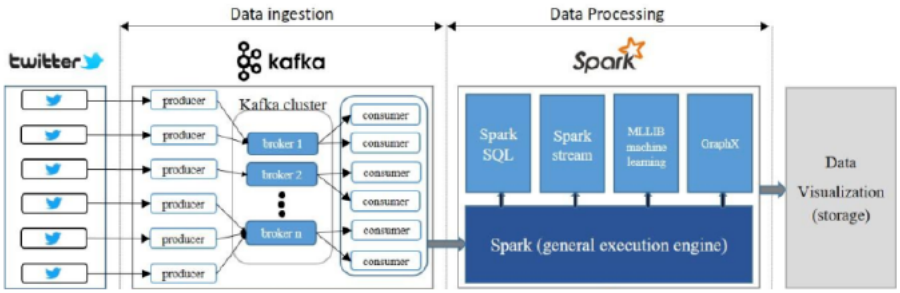

Figura 1.2. Developing a Real-Time Data Analytics Framework for Twitter Streaming Data [2]

cupa di esaminare i dati in tempo reale ricevuti dai consumer Kafka e li suddivide in mini-batches. Questi batches successivamente vengono elaborati da Spark e poi forniti al modulo successivo. L'ultimo modulo, il data visualization, permette di visualizzare i dati mediante un database non relazionale. Oltre a visualizzare il testo è possibile anche visualizzare immagini.
Invece, in "An Ingestion and Analytics Architecture for IoT applied to Smart City Use Cases" è presente un esempio di architettura progettata in ambito IoT[19].Un'applicazione di questo tipo necessita di rispondere in tempo reale ad eventi effettuando scelte basate su eventi passati. L'architettura di questa applicazione può essere suddivisa in due flussi: batch e stream. Possiamo notare i due flussi differenti nella figura 1.3. Il primo punto consiste nell'effettuare l'ingestion e quindi acquisire i dati. L'acquisizione dei dati viene effettuata da Node Red che permette di raccogliere informazioni provenienti da diverse fonti esterne e dispositivi in formato XML o JSON. E' possibile utilizzare funzionalità di pre-processing per rimuovere eventuali informazioni ridondanti presenti nei dati. Node Red adesso è in grado di propagare i risultati raggiunti su Kafka. Successivamente questi risultati potranno essere trasmessi dal message broker a OpenStack Swift utilizzando il tool open-source Sector. Siccome la quantità di dati è importante, vengono utilizzati i metadati all'interno di Swift per semplificare la ricerca delle varie informazioni mediante Elastic search. A questo punto, per accedere ai dati presenti sul database viene utilizzato Apache Spark che è un software di analisi batch. Questo software offre anche delle librerie per il machine learning. Per creare eventi più complessi e collegarli viene utilizzato il framework Complex Event Processing. Questo framework si occupa di effettuare il processamento in tempo reale. Inoltre fornisce diverse funzionalità che consentono di perfezionare l'acquisizione dei dati andando a semplificare la logica di business mediante delle regole da seguire fino a quando non viene raggiunto il risultato finale. Il flusso di dati stream viene eseguito contemporaneamente al risultato prodotto dall'analisi batch in questo componente dell'architettura.

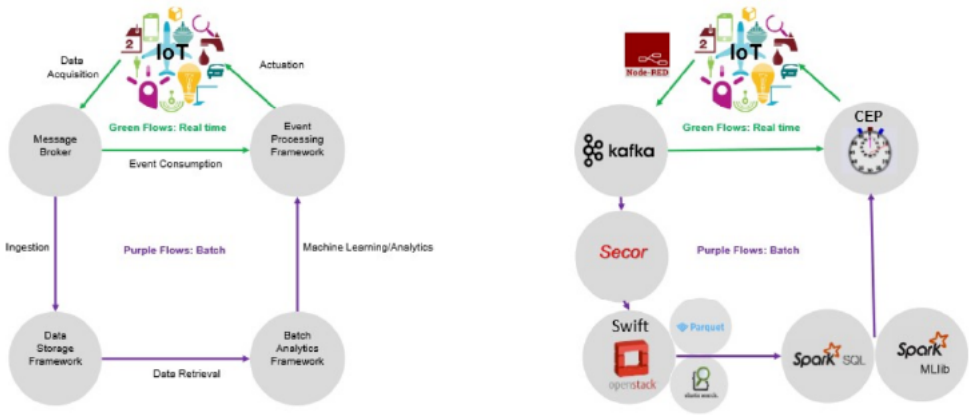

Figura 1.3. Analytics and Ingestion Architecture for IoT [19]

1.3 Use case della Streaming Platform

Negli ultimi anni, in particolare negli ultimi due, l'impiego di piattaforme di videoconferenza è diventato una prerogativa indispensabile. Infatti, ormai sono ampiamente utilizzate sia nel tempo libero che nel lavoro, consentendo alle persone di rimanere in contatto e alle aziende di risparmiare importanti costi di viaggio. La loro efficacia si è mostrata principalmente nel corso dei mesi di quarantena a causa della pandemia di COVID-19, dove i meetings online hanno consentito di portare avanti il tessuto industriale spostando il lavoro in remoto. Questo è stato possibile anche grazie ai miglioramenti dell'infrastruttura di rete, che, ci ha permesso di disporre di una larghezza di banda sempre più grande[17]. Tra le prime applicazioni di videoconferenza (lanciata negli anni 2000), troviamo Skype, che ha fatto da pioniere per le applicazioni RTC. In quegli anni, la maggior parte degli utenti disponeva di connessioni mediante modem via cavo, con scarsa larghezza di banda e alta latenza. Questo portava a continui malfunzionamenti dell'applicazione, rendendo di fatto il meeting online molto complicato e compromettendo l'esperienza utente. Invece, oggi il mercato offre un'infinità di piattaforme concorrenti per le videochiamate, che, con strategie algoritmiche più efficaci e larghezza di banda sempre maggiore ha spostato decisamente in alto la qualità del servizio offerto. Fornire servizi sempre migliori è ormai una prerogativa imprescindibile in un mercato altamente competitivo, e questo si traduce nella necessità di studiare nuove strategie per massimizzare la Quality of Experience (QoE) degli utenti, rendendo l'esperienza finale sempre più simile ad un'interazione reale. Si può pensare alla QoE come ad un'evoluzione della Quality of Service, dove oltre a stimare i parametri di rete si cerca di avere un feedback anche su come l'utente finale sta percependo il servizio. Non esiste un modo univoco di definire la QoE, essa chiaramente dipende dal tipo di servizio che si va ad utilizzare. In ambito di videoconferenza, classificare il traffico RTC[11] è sicuramente il primo passo verso la gestione efficiente del traffico, permettendo così ai dispositivi in rete di stimare la QoE percepita dagli utenti. Eseguire la classificazione in real-time, consente di avviare azioni adeguate con l'obbiettivo di risolvere eventuali

peggioramenti. Nella RTC, questo potrebbe equivalere ad identificare i flussi con priorità massima da quelli con priorità minima, allocando risorse in maniera più efficiente così da garantire continuità di servizio anche in situazioni critiche. In questo progetto viene realizzata una streaming platform per la cattura e la classificazione, in tempo reale, dei flussi multimediali RTC trasportati da flussi RTP. Il sistema che si va a sviluppare parte analizzando il traffico dati real-time che, mediante tecniche basate su DPI (deep packet inspection), seleziona solo i flussi RTP. Per ogni flusso saranno estratte delle features che serviranno come input al classificatore, che con alta precisione (F1-score 0.97) sarà in grado di identificare il tipo di sorgente multimediale.

Capitolo 2

Design della Streaming Platform

In questo capitolo verrà rappresentata la streaming platform da un punto di vista architetturale e verranno elencati gli strumenti open-source utilizzati per la sua implementazione. Per specificare l'architettura della piattaforma è stato utilizzato uno schema a livelli che garantisca un flusso di dati sicuro. I livelli con cui è stata progettata l'architettura di questa streaming platform sono:

- Data Ingestion Layer
- Data Collector Layer
- Data Processing Layer
- Data Storage Layer
- Data Query Layer
- Data Visualization Layer

2.0.1 Data Ingestion Layer

Il primo livello è il Data Ingestion Layer e viene impiegato per ottenere dati da sorgenti esterne. Una volta acquisiti questi dati si occupa di instradarli verso la destinazione corretta. Il data ingestion consiste, quindi, nell'acquisizione e nell'importazione dei dati e indica l'inizio della pipeline. I dati possono essere acquisiti in real-time oppure in batch. Nell'ingestion batch i dati vengono collezionati per intervallo di tempo mentre nell'ingestion real-time ogni volta che si presenta un evento questo viene mandato all'istante al livello successivo dell'architettura tentando di assicurare la latenza minore possibile. L'elaborazione in batch è quella più comune tra le due e qui i dati vengono raccolti, raggruppati e infine inviati alla destinazione periodicamente. Quando non è rilevante avere a disposizione dati in tempo reale, viene adoperata l'elaborazione in batch siccome è solitamente più conveniente rispetto a quella real-time. L'elaborazione in real-time, invece, non

necessita di nessun raggruppamento. I dati vengono generati, trattati e caricati appena vengono prodotti o riconosciuti dal Data Ingestion Layer. Questo tipo di acquisizione è più costoso perché necessita di sistemi di monitoraggio costanti per le fonti e di accettare nuove informazioni. Questo livello, inoltre, potrebbe essere adoperato per realizzare azioni di pre-processing sui dati che li organizzano per il livello successivo della piattaforma. Grazie al pre-processing si è in grado di andare a rendere migliore la qualità dei dati. Il Data Ingestion Layer di questa streaming platform sarà un Data Ingestion Layer real-time perché l'obbiettivo è catturare in tempo reale il traffico di rete per poi classificarlo nell'ordine di pochissimi secondi(1s). Per implementare questo livello non sono stati utilizzati strumenti open-source, come ad esempio Apache Flume o Apache NiFi, ma sono stati realizzati due script Python. Uno script si occupa di leggere i pacchetti di una cattura Wireshark da un file .pcap mentre l'altro si occupa di catturare pacchetti in tempo reale mediante l'utilizzo di Pyshark.

2.0.2 Data Collector Layer

In questo livello viene prestata molta attenzione al trasporto dei dati dal livello di Ingestion a quello di Processing. Esso è composto dal message broker e viene adoperato per la comunicazione tra questi due livelli dell'architettura mediante lo scambio di messaggi che hanno una forma ben definita. La streaming platform dovrà effettuare una serie di operazioni in un determinato ordine per ottenere il risultato finale e affiché questo accada, il message broker costituisce il percorso che i dati dovranno osservare. Il message broker, per offrire una consegna assicurata dei messaggi e una loro archiviazione sicura, utilizza il message queue che memorizza e ordina i messaggi finché le applicazioni che ne fanno uso sono in grado di eseguirli. I messaggi all'interno della coda vengono archiviati nell'ordine con cui sono stati inviati e stanno nella coda fino alla garanzia del ricevimento. Tra le diverse applicazioni nasce una comunicazione asincrona che evita la perdita di dati importanti e permette ai sistemi di lavorare anche se presente latenza o connettività a tratti. Il message broker capisce i gestori delle code che sono adeguati a coordinare le varie code di messaggi e questo lo porta a cogliere i servizi che mostrano caratteristiche di routing di dati, controllo dello stato del client, costanza e comprensione dei messaggi. Inoltre, il message broker può osservare due diverse tipologie di messaggistica:

- Punto punto
- Produttore e consumatore

Nella prima tipologia è presente una relazione 1:1 tra il mittente e il destinatario del messaggio. Qualunque messaggio presente nella coda viene mandato ad un solo destinatario e viene consumato una sola volta. Questa tipologia di messaggistica viene utilizzata se un messaggio deve essere adoperato soltanto una volta. Mentre nella seconda tipologia viene adoperato il topic. Un produttore pubblicherà ogni messaggio prodotto su uno specifico topic e ciascun consumatore registrato a quel topic sarà in grado di avere quei determinati messaggi. Qui è presente una relazione 1:N tra il produttore del messaggio e i propri consumatori. Lo strumento open-source utilizzato per eseguire le operazioni di un message broker è Apache Kafka di cui si parlerà approfonditamente nel prossimo capitolo.

2.0.3 Data Processing Layer

In questo livello vengono elaborate le informazioni raccolte dal livello precedente. Al fine di pulire e trasformare i dati giunti da origini diverse, l'elaborazione dei dati deve riguardare ogni record all'interno dei dati in arrivo. Una volta che un record è pulito e trasformato, il lavoro è concluso. Questo livello presuppone di avere i dati da elaborare disposti in stream e che questi vengano elaborati utilizzando diverse operazioni. Per diminuire i tempi di elaborazione molte volte le operazioni vengono realizzate usando strutture a pipeline. Una caratteristica fondamentale di quello livello è l'elaborazione dei dati in parallelo escludendo problemi di sincronizzazione e legame dei dati. Quando viene ricevuto un evento, questo livello reagisce immediatamente: ad esempio, si possono aggiornare dei dati o far scattare un allarme. Lo strumento open-source che si occupa dello streaming processing in questa streaming platform è Apache Flink di cui si parlerà approfonditamente nel prossimo capitolo.

2.0.4 Data Storage Layer

In questo livello vengono mantenuti i dati prodotti e ricevuti dal livello precedente. I dati mantenuti sono dati elaborati e migliorati da informazioni valide dallo streaming processing. L'archiviazione diviene una sfida quando il volume di dati diventa enorme e ci sono varie soluzioni possibili per affrontare questo problema. Cercare una soluzione di archiviazione è molto importante quando la mole di dati diventa vasta. Questo livello è caratterizzato da un database non relazionale che si occupa di mantenere e archiviare i dati per poter essere poi visualizzati nel Data Visualization Layer. È stato utilizzato un database non relazionale perché, a differenza di un database relazionale, possiede i seguenti vantaggi:

- Può memorizzare diversi tipi di dati come JSON, grafi oppure coppie chiave-valore
- Nessun record deve avere le stesse caratteristiche degli altri quindi lo porta a non possedere una struttura fissa
- Garantisce la mancanza ridondanza e le incongruenze anche se i dati non sono normalizzati
- Garantisce la scalabilità orizzontale
- Garantisce enormi prestazioni nelle operazioni di lettura e scrittura su tutto il dataset
- Garantisce elevate performance mediante un trade-off di disponibilità e consistenza

Il database non relazionale open-source adottato in questo livello di questa streaming platform è Apache CouchDB di cui si parlerà approfonditamente nel prossimo capitolo.

2.0.5 Data Query Layer

In questo livello accade l'elaborazione analitica attiva dei dati. Il suo scopo fondamentale è raccogliere il valore dei dati per fornirli nel modo più adatto al livello successivo.

Per fare la query al database non relazionale e recuperare i dati verrà implementato un microservizio[15]. La scelta di implementare un microservizio è dovuta al voler realizzare la streaming platform mediante un approccio architetturale. Questo approccio, a differenza di quello tradizionale, divide la piattaforma in diversi servizi. Ogni operazione prende il nome di microservizio e viene compilata e implementata indipendentemente dalle altre. Perciò ogni microservizio può funzionare o meno senza danneggiare gli altri. Questa scelta facilita l'eventuale aggiunta di un nuovo servizio in modo semplice senza compromettere o danneggiare quelli già esistenti. All'interno del microservizio che verrà creato, si implementerà un'API che effettui delle chiamate al database non relazionale. L'API[13] (Application Programming Interface) è una serie di protocolli e definizioni con cui verrà realizzato questo livello della streaming platform. Permetterà al servizio realizzato di comunicare con altri senza sapere come sono stati sviluppati, facilitando così il miglioramento della streaming platform. Implementare un'API porta i seguenti vantaggi:

- Garantisce flessibilità
- Semplifica la progettazione
- Semplifica l'utilizzo
- Semplifica l'amministrazione
- Garantise la possibilità di innovazione
- Garantisce l'accesso alle risorse in totale sicurezza
- Garantisce integrità
- Permette di non esporre il database al front-end che è una cosa molto pericolosa

Con i livelli precedenti verrà implementata la parte di back-end, con il successivo quella di front-end mentre con questo livello si andrà ad implementare il back-end for front-end cioè un back-end che fornisce i dati al front-end perché il front-end è consigliabile che non acceda al database per motivi di sicurezza. All'interno dell'API verrà scritto del codice che interroga il database, il database restituirà dei dati in formato JSON memorizzati nel Data Storage Layer e questi dati vengono resi disponibili al livello successivo nel momento in cui li richiede. Quindi l'API verrà scritta ci consentirà di interrogare il database e consentirà al livello successivo di interfacciarsi con esso.

2.0.6 Data Visualization Layer

Questo livello è quasi certamente quello più rilevante perché consente all'utente di visualizzare il valore dei dati. Nella realizzazione di questo livello occorre utilizzare qualcosa che attragga l'attenzione dell'utente e che permetta una visualizzazione del valore dei dati correttamente compresa. Come è stato detto nello scorso livello, si potrebbe interrogare il database per ricavare il valore dei dati in questo livello ma non è corretto per una serie di motivi tra cui la sicurezza perché si andrebbe ad esporre il database. Per visualizzare il valore dei dati, in questo livello, verrà utilizzata l'API definita nel livello precedente. In

questo livello andremo a sviluppare una Web App che consenta di interfacciarsi con l'API e che permetta di visualizzare il valore dati. La visualizzazione del valore dei dati consiste nel decifrare questo valore in un contesto visivo, come ad esempio un grafico o una lista, per presentarli nel modo più comprensibile possibile. Per lo sviluppo di questa Web App verrà utilizzato il framework open-source Angular[1]. Utilizzare Angular porta una serie di vantaggi:

- Garantisce il controllo sulla scalabilità
- Accontenta una notevole quantità di requisiti di dati sviluppando modelli di dati su un modello push
- Garantisce la produttività
- Permette di realizzare un'infrastruttura scalabile
- Permette di estendere il linguaggio del modello con propri componenti e si possono utilizzare una vasta varietà di componenti esterni, senza doversi preoccupare troppo di far funzionare il codice
- Permette di realizzare app straordinarie

L'obiettivo principale di questo livello è quindi visualizzare il valore dei dati presenti sul database non relazionale, interfacciandosi con esso mediante l'utilizzo di un'API, su una Web App Angular sfruttando diversi componenti grafici.

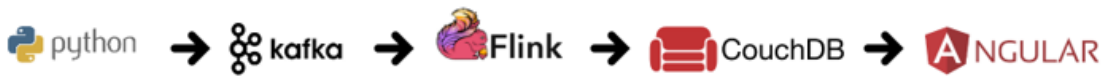

Figura 2.1. Main open-source tools used

Capitolo 3

Strumenti open-source utilizzati

Al mondo d'oggi, le applicazioni generano ininterrottamente dati in tempo reale e questi dati necessitano di essere indirizzati il più rapidamente possibile verso diversi tipi di ricevitori. Le applicazioni che generato i dati e quelle che li consumano sono tipicamente separate e, per garantire una loro integrazione, devono essere riqualificate. Per far questo è fondamentale un meccanismo che integri esattamente i dati del produttore e del consumatore per evitare ogni tipo di riscrittura dell'applicazione in ambedue le estremità. Una possibile soluzione a questi problemi è Apache Kafka[12] che consente di gestire vasti volumi di dati in tempo reale e per di più permette di instradare rapidamente questi dati a diversi consumatori. Apache Kafka può essere considerato come un sistema di code di messaggi ad elevata velocità e distribuito che distribuire i dati disponibili a più consumatori. Inoltre, consente di far integrare esattamente i dati dei produttori e i dati dei consumatori, garantendoli durevoli conservando i messaggi in arrivo su disco in una struttura dati a registro. Queste funzionalità vengono fornite in modo distribuito, elastico, scalabile e sicuro. Apache Kafka può essere distribuito su hardware, macchine virtuali e container sia in locale che in cloud.

3.1 Apache Kafka

Apache Kafka[14] è uno strumento open-source per il data streaming che consente di pubblicare, archiviare, trasformare e sottoscrivere flussi di record in tempo reale. E' stato ideato per gestire numerosi flussi di dati che arrivano da diverse sorgenti e che vengono consegnati a diversi consumatori. Con Apache Kafka è possibile trasferire enormi volumi di dati da un punto ad un altro nel medesimo istante e rimpiazzare il classico sistema di messaggistica. Il sistema era nato per gestire milioni di messaggi giornalmente per poi diventare uno strumento open-source per il data streaming capace di esaudire diverse esigenze.

3.1.1 A cosa serve Apache Kafka?

Apache Kafka[9] viene inserito nei flussi di data streaming che permettono la divisione dei dati tra sistemi e applicazioni. Viene adoperato quando si ha la necessità di avere un'elevata velocità e un'enorme scalabilità. Permette di limitare al minimo la latenza e in determinate applicazioni consente di diminuire il bisogno di integrazioni punto-punto per poter condividere dati. Gli utenti sono in grado di ricevere dati velocemente, cosa molto vantaggiosa quando si lavora in tempo reale. Apache Kafka risolve diversi dubbi introdotti dai Big Data ed è, per questo motivo, molto vantaggioso per aziende che necessitano di risolvere questo tipo di problema. In diversi casi d'uso in cui è presente l'elaborazione dei dati, la mole di dati si ingrandisce così velocemente da essere in grado di danneggiare applicazioni disponibili. Bisogna allora presagire una possibile scalabilità nell'elaborazione che permetta di reagire alla crescita esponenziale dei dati.

3.1.2 Come funziona Apache Kafka?

Apache Kafka è un sistema distribuito costituito da client e server che comunicano tra di loro mediante il protocollo TCP. E' in grado di essere distribuito su macchine virtuali, container in ambienti cloud e hardware.

- Client: Permettono di implementare microservizi e applicazioni distribuite che possono leggere, scrivere ed elaborare flussi di eventi in parallelo, tolleranti agli errori. Apache Kafka è dotato di svariati di questi client, cresciuti da una notevole quantità di client forniti dalla società di Apache Kafka.

- Server: Apache Kafka deve essere eseguito come un cluster di uno o diversi server, i quali si possono ampliare su diverse aree cloud o datacenter. Svariati di questi server danno origine al livello di archiviazione, che prende il nome di broker. Altri server, invece, utilizzano Kafka Connect per esportare ed importare costantemente dati come flussi di eventi per perfezionare Apache Kafka con sistemi disponibili quali database relazionali o altri cluster Apache Kafka. Per permettere di realizzare casi d'uso abbastanza critici, un cluster Apache Kafka deve essere profondamente scalabile e tollerante ai guasti.

3.1.3 Perché abbiamo bisogno di Apache Kafka?

Al mondo d'oggi, le aziende generano una sproporzionata quantità di dati. Per di più vengono generati anche dati quanto un utente visita pagine sul web, effettua dei click oppure naviga sui social network. Questi dati sono gestiti da classiche soluzioni che li assegnano a sistemi che li esaminano in offline. Inoltre, queste soluzioni non permettono di realizzare sistemi di elaborazione real-time. Tenendo conto della nascita di nuove applicazioni internet, i dati necessitano di un'analisi in tempo reale. Apache Kafka tenta di fondere l'elaborazione online e offline dei dati provvedendo a partizionare l'utilizzo in tempo reale su diverse macchine.

3.1.4 Concetti principali e terminologia di Apache Kafka

Un evento sta ad indicare che è successo qualcosa e i dati vengono letti o scritti su Apache Kafka come eventi. Un evento concettualmente possiede una chiave, un valore, un timestamp e degli headers facoltativi. I produttori sono le applicazioni client che pubblicano eventi su Apache Kafka mentre i consumatori sono quelli che leggono questi eventi. Produttori e consumatori sono totalmente separati su Apache Kafka che gli permette di possedere una notevole scalabilità. Inoltre, Apache Kafka mette a disposizione diverse garanzie come l'elaborazione degli eventi rigorosamente una volta. Gli eventi vengono organizzati in topic e un topic possiamo considerarlo come una cartella all'interno di un file system mentre gli eventi sono i file all'interno di quella cartella. I topic in Apache Kafka sono multi-produttore e multi-consumatore: un topic può possedere zero, uno o molti produttori che scrivono eventi e zero, uno o molti consumatori che leggono questi eventi. Un topic è costituito da diverse partizioni, un sottoinsieme dei dati che vengono serviti dal topic. Ogni singola partizione di un topic viene gestita da un broker Apache Kafka, un servizio installato sul nodo che possiede la partizione e permette a produttori e consumatori di accedere ai dati di un topic. Una partizione potrebbe essere duplicata al fine di assicurare la durabilità però potrebbe succedere che diversi broker vogliano amministrare la stessa partizione. Per evitare ciò, un broker viene eletto leader mentre gli altri follower. La posizione distribuita dei dati è fondamentale perché assicura alle applicazioni client di scrivere e leggere dati da/verso diversi broker nello stesso istante. Il posizionamento distribuito dei dati è molto importante per garantire la scalabilità perché consente alle applicazioni client di leggere e scrivere dati da/verso più broker contemporaneamente. Apache Kafka consente di leggere gli eventi all'interno di un topic tutte le volte che è indispensabile mentre sistemi tradizionali di messaggistica una volta letti gli eventi vengono eliminati. Si può precisare per quanto tempo Apache Kafka può custodire i nostri eventi mediante un'impostazione di configurazione, poi quelli vecchi vengono cancellati. L'archiviazione dei dati per un lungo periodo di tempo va bene in Apache Kafka perché le sue prestazioni sono costanti a differenza della dimensione dei dati. Il collocamento distribuito dei dati è fondamentale per la scalabilità perché permette alle applicazioni client di scrivere e leggere dati da più broker nello stesso momento. Quando viene scritto un nuovo evento su un topic, viene inserito in una delle partizioni del topic. Gli eventi che posseggono la stessa chiave vengono inseriti all'interno della stessa partizione e Apache Kafka consente a ogni consumatore di una precisa partizione di leggere sempre gli eventi di quella partizione rigorosamente nell'ordine con cui sono stati inseriti. Per far diventare i dati tolleranti ai guasti e profondamente disponibili, ciascun topic può essere replicato, anche tra data center, per far si che il numero di broker che posseggano una copia dei dati sia sempre maggiore. Questo permette, nel caso le cose vadano male, di effettuare una manutenzione sui broker. Un'impostazione comune di produzione possiede un fattore di replica pari a 3, cioè saranno continuamente presenti tre copie dei dati e questa replica viene effettuata a livello di partizioni dei topic. Nella figura 3.1 è presente un topic che possiede quattro partizioni P1-P4. Due diversi produttori stanno pubblicando, liberamente l'uno dall'altro, dei nuovi eventi sul topic scrivendo in rete eventi all'interno delle partizioni del topic. Gli eventi che posseggono la stessa chiave (indicati nella figura dal colore) vengono scritti all'interno della stessa partizione. Ambedue i produttori possono

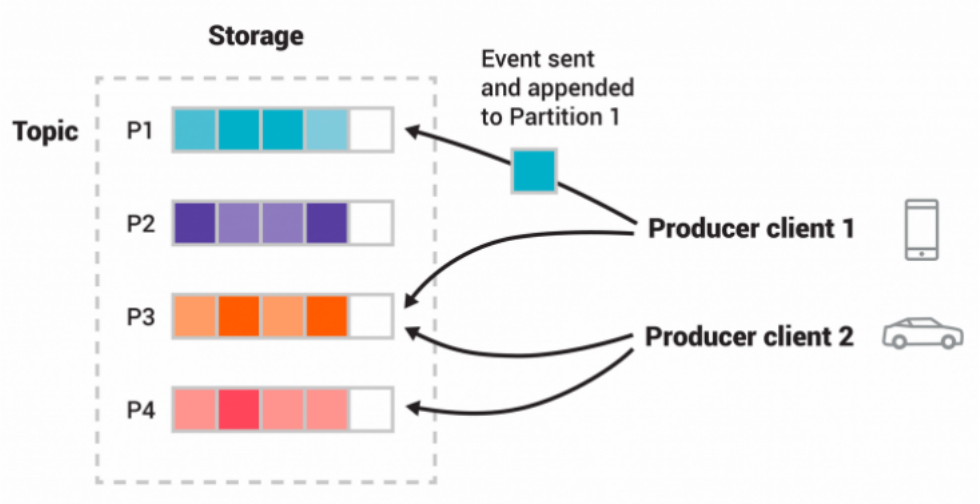

Figura 3.1. Example of a Kafka topic and its partitions [9]

scrivere all'interno della stessa partizione, se opportuno.

3.1.5 API di Apache Kafka

Apache Kafka possiede cinque API principali per Java e Scala:

- L'Admin API viene utilizzata per gestire topic, broker e altri oggetti Apache Kafka.
- La Producer API viene utilizzata per pubblicare un flusso di eventi su uno o più topic Apache Kafka.
- La consumer API viene utilizzata per leggere da uno o più topic e per manipolare il flusso di eventi generati.
- La Kafka Streams API viene utilizzata per realizzare microservizi e applicazioni di elaborazione dei flussi. E' dotata di funzioni per elaborare flussi di eventi. L'input viene preso da uno o più topic per produrre un output su uno o più topic, trasformando in modo adeguato i flussi di input in quelli di output.
- La Kafka Connect API viene utilizzata per creare e realizzare dei connettori che si occupano dell'importazione e dell'esportazione dei dati che producono o consumano flussi di eventi da e verso applicazioni esterne per far si che si possano integrare perfettamente con Apache Kafka. Generalmente non è indispensabile realizzare dei propri connettori perché Apache Kafka ne è dotato di diversi già esistenti.

3.1.6 Casi d'uso di Apache Kafka

Descrizione dei casi d'uso più diffusi di Apache Kafka:

- Messaggistica: Apache Kafka sostituisce efficacemente un classico broker di messaggi. I broker di messaggi possono mantenere nel buffer messaggi non elaborati, separare l'elaborazione dei dati dai produttori, e altro ancora. Apache Kafka, rispetto ad altri sistemi di messaggistica, possiede un throughput più vantaggioso, tolleranza agli errori e partizionamento integrato. Tipicamente l'utilizzo della messaggistica ha un throughput basso però può richiedere una bassa latenza.

- Metrica: Apache Kafka viene frequentemente adoperato nel monitoraggio operativo dei dati. Per questo motivo, vengono aggregate statistiche di applicazioni distribuite per generare dei feed di dati operativi.

- Origine dell'evento: L'origine dell'evento viene registrato come una serie di record ordinata nel tempo e riguarda la progettazione dell'applicazione. Apache Kafka fornisce un supporto ai dati di log elevato e questo gli consente di essere un back-end eccellente per un'applicazione realizzata con questo stile.

- Monitoraggio dell'attività del sito web: Apache Kafka è in grado di ripristinare la pipeline di monitoraggio del lavoro dell'utente come un insieme di feed pubblicazione-sottoscrizione in tempo reale. Questo vuol dire che le attività svolte dall'utente su un sito web vengono pubblicate su topic centrali con un topic per attività. Questi feed sono utilizzabili per sottoscrivere diversi casi d'uso come l'elaborazione in tempo reale, il monitoraggio in tempo reale e il riempimento di data warehouse offline. Il monitoraggio delle attività, tipicamente, è un volume enorme perché per ogni visita della pagina da parte dell'utente vengono generati molti messaggi.

- Commit Log: Apache Kafka può ricoprire il ruolo di commit log per un sistema distribuito. Facilita la replica dei dati tra i vari nodi e consente loro la risincronizzazione per ripristinare i propri dati in caso di errore.

- Elaborazione del flusso: Apache Kafka viene sfruttato da diversi utenti per elaborare dati in una pipeline di elaborazione formata da varie fasi, dove i dati grezzi di input sono utilizzati dai topic di Apache Kafka e successivamente aggregati, arricchiti oppure trasformati per un'aggiuntiva elaborazione. In Apache Kafka è presente una libreria leggera ma potente, per l'elaborazione dei flussi di dati, chiamata Kafka Streams.

- Aggregazione dei log: Uno dei maggiori utilizzi di Apache Kafka è l'aggregazione dei log. Questa aggregazione ottiene file di log fisici dai server e li dispone in una posizione centrale per l'elaborazione. Apache Kafka cattura dai file diversi dettagli e offre i dati in modo pulito. Questo consente di elaborare i dati con una bassa latenza, consumo distribuito e supporto semplice. Apache Kafka garantisce buone prestazioni grazie alla latenza end-to-end molto bassa.

3.1.7 Avvio di Apache Kafka

1. Ottenere Apache Kafka

   Scaricare ed estrarre l'ultima versione di Apache Kafka con i seguenti comandi:

(i) `tar -xzf kafka_2.13-2.8.0.tgz`

(ii) `cd kafka_2.13-2.8.0`

2. Avviare l'ambiente Apache Kafka

   (i) Aprire il terminale e lanciare il seguente comando per avviare lo ZooKeeper:
   `bin/zookeeper-server-start.sh config/zookeeper.properties`

   (ii) Aprire un altro terminale e lanciare il seguente comando per avviare il Kafka broker service:
   `bin/kafka-server-start.sh config/server.properties`

   Quando tutti i servizi saranno iniziati correttamente, si avrà un ambiente Apache Kafka in esecuzione.

3. Creare un topic per memorizzare gli eventi

   (i) Per creare un topic aprire il terminale e lanciare il seguente comando:
   `bin/kafka-topics.sh -create -topic topic-kafka -bootstrap-server localhost:9092`

4. Scrivere diversi eventi sul topic

   (i) Per scrivere eventi sul topic appena creato ed eseguire il client produttore aprire il terminale e lanciare il seguente comando:
   `bin/kafka-console-producer.sh -topic topic-kafka -bootstrap-server localhost:9092`
   Per fermare il client produttore premere `Ctrl-C` in qualsiasi instante.

5. Leggere gli eventi dal topic

   (i) Per leggere eventi presenti sul topic ed eseguire il client consumatore aprire il terminale e lanciare il seguente comando:
   `bin/kafka-console-consumer.sh -topic topic-kafka -from-beginning -bootstrap-server localhost:9092`

   Per fermare il client consumatore premere `Ctrl-C` in qualsiasi instante.

6. Importare/esportare dati come stream di eventi con Kafka Connect

   Potrebbe accadere di avere molti dati su database, classici sistemi di messaggistica o altri tipi di sistemi. Kafka Connect permette di importare ininterrottamente dati da sistemi esterni ad Apache Kafka e viceversa. Questo consente di integrare facilmente sistemi esistenti con Apache Kafka. Per fare questo sono disponibili dei connettori.

7. Elaborare gli eventi con Kafka Streams

   I dati salvati in Apache Kafka come eventi possono essere elaborati con la libreria Kafka Streams. Questa libreria permette di creare applicazioni e microservizi in

tempo reale, dove i dati di input e i dati di output vengono memorizzati nei topic Apache Kafka. Kafka streams unisce la facilità di scrivere e distribuire applicazioni lato client con i vantaggi della tecnologia lato server di Apache Kafka per trasformare le applicazioni e farle diventare scalabili, elastiche, tolleranti ai guasti e molto altro.

8. Terminare l'ambiente Apache Kafka

   L'ambiente Apache Kafka è stato avviato e può essere fermato in qualsiasi momento con i seguenti comandi:

   (i) Fermare il client producer e consumer con `Ctrl-C`

   (ii) Fermare il Kafka broker con `Ctrl-C`

   (iii) Fermare lo ZooKeeper con `Ctrl-C`

   Se si vuole cancellare anche tutti i dati del proprio ambiente Apache Kafka locale, inclusi gli eventi che sono stati creati, lanciare il seguente comando da terminale:

   ```
   rm -rf /tmp/kafka-logs /tmp/zookeeper
   ```

3.2 Apache Flink

Apache Flink[6] è un framework open-source di elaborazione distribuito che effettua calcoli stateful su flussi di dati limitati e illimitati,nato per lavorare in ambienti cluster ed effettuare calcoli ad alta velocità. Siccome la maggior parte della applicazioni di streaming vengono costruite per essere eseguite senza periodi di inattività, uno stream deve garantire un ottimo ripristino degli errori e possedere strumenti che permettano di monitorare le applicazioni in esecuzione. Esso pone molta attenzione alle caratteristiche operative dell'elaborazione del flusso, possiede un meccanismo di ripristino degli errori e consente di controllare le applicazioni in esecuzione.

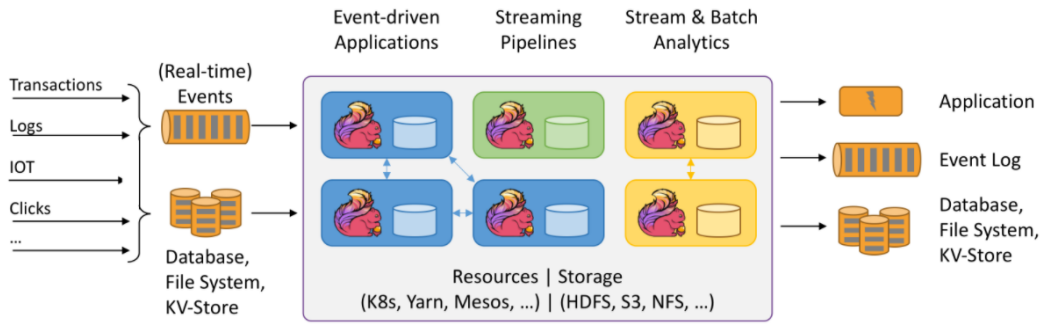

Figura 3.2. Stateful Computations over Data Streams [4]

3.2.1 Elaborazione di dati limitati e illimitati

Ogni tipo di dato viene generato come un flusso di eventi. Ne sono un esempio le misurazioni di un sensore, il registri dell'auto oppure le transazioni di una carta di credito. I dati vengono elaborati come flussi limitati o illimitati.

- I flussi limitati possiedono un inizio e una fine ben definiti. Essi vengono elaborati aggiungendo tutti i dati prima di realizzare qualunque calcolo. Importare i dati in modo ordinato non è fondamentale per elaborare i flussi limitati perché un insieme di dati limitato può essere ordinato in qualsiasi momento. L'elaborazione di flussi limitati è anche detta elaborazione batch.
- I flussi illimitati possiedono un inizio ma non una fine definita. Essi non finiscono, assegnano i dati così come vengono generati e vengono elaborati continuamente cioè gli eventi vengono gestiti subito dopo che sono stati ottenuti. Non si può aspettare l'arrivo di tutti i dati di input perché l'input è illimitato e non sarà mai ultimato. L'elaborazione di dati illimitati necessita frequentemente di eventi inseriti in un specifico ordine.

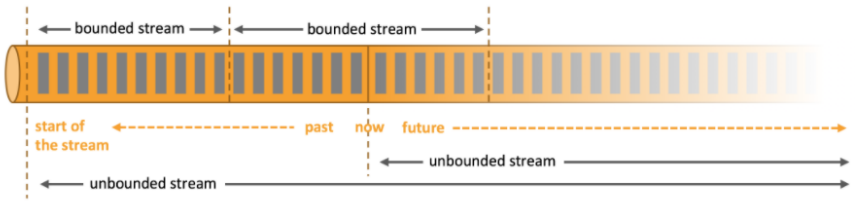

Figura 3.3. Bounded and unbounded streams [6]

Apache Flink si distingue nell'elaborazione di set di dati limitati e illimitati. La verifica accurata sul tempo e sullo stato gli permette di eseguire qualunque tipo di applicazione su flussi di dati illimitati. I flussi limitati, invece, vengono elaborati all'interno di una struttura dati con algoritmi progettati esclusivamente per un set di dati di dimensione definita, garantendo ottime prestazioni.

3.2.2 Deploy delle applicazioni

Apache Flink è un sistema distribuito che necessita di risorse di elaborazione per realizzare le applicazioni. Esso può essere configurato come cluster autonomo ma si integra anche perfettamente con i comuni gestori di risorse cluster come Kubernetes. È stato realizzato per lavorare perfettamente con i classici gestori delle risorse che forniscono precise modalità di implementazione con cui Apache Flink interagisce efficacemente con esse. Quando viene distribuita un'applicazione Flink, Apache Flink riconosce tutte le risorse chieste al gestore delle risorse e per comunicare con l'applicazione vengono usate le chiamate REST che gli permettono ad Apache Flink di integrarsi in molti ambienti. Se dovesse verificarsi un guasto, Apache Flink può richiedere nuove risorse.

3.2.3 Eseguire applicazioni su ogni scala

Apache Flink è nato per realizzare applicazioni di streaming in tempo reale su qualunque scala. Queste applicazioni devono essere parallelizzate in migliaia di attività che successivamente vengono eseguite e distribuite insieme in un cluster. Un'applicazione Flink può quindi usufruire di illimitate quantità virtuali di CPU, disco e memoria centrale. Apache Flink permette di mantenere comodamente lo stato dell'applicazione e possiede un algoritmo di checkpoint che gli garantisce di avere un piccolissimo impatto sulla latenza di elaborazione.

3.2.4 Sfruttare le prestazioni in memoria

Le applicazioni stateful Flink vengono ottimizzate per consentire l'accesso allo stato locale. Lo stato delle attività viene conservato in memoria o in strutture dati su disco che posseggono un accesso efficace. Le attività eseguono i vari calcoli sfruttando l'accesso allo stato locale che garantisce basse latenze di elaborazione. Apache Flink garantisce, in caso di errori, la coerenza dello stato esaminando frequentemente lo stato locale.

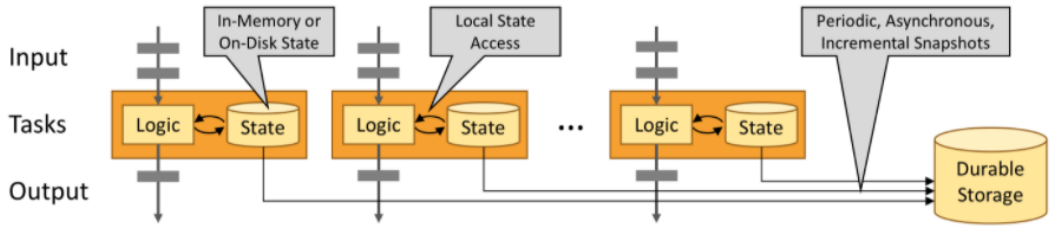

Figura 3.4. Leverage In-Memory Performance [6]

3.2.5 Elementi delle applicazioni streaming

Le applicazioni che vengono realizzate da un framework di elaborazione del flusso sono specificate dal modo in cui il framework esamina gli streams, lo stato e il tempo.

- Streams: Gli streams rappresentano un aspetto molto importante nell'elaborazione dei flussi perché possono avere aspetti diversi che caratterizzano il modo con cui vengono elaborati. Apache Flink è un framework di elaborazione duttile capace di trattare qualsiasi tipo di flusso.
  - Streams limitati e illimitati: Gli streams possono essere illimitati o limitati. Apache Flink[5] possiede caratteristiche complesse per elaborare streams illimitati, ma anche operatori che si occupano di elaborare streams limitati in modo efficiente.
  - Streams in tempo reale: Ogni dato viene generato come flusso e può essere elaborato in tempo reale non appena viene generato oppure successivamente. Le

applicazioni Flink possono elaborare flussi in tempo reale oppure in un secondo momento.

- Stato: Tutte le applicazioni di streaming complesse sono stateful cioè effettuano trasformazioni su eventi senza richiederne lo stato, mentre le applicazioni che applicano logica di business devono ricordarsi di eventi e risultati intermedi per potervi accedere successivamente. Lo stato di un'applicazione Flink è un aspetto fondamentale.

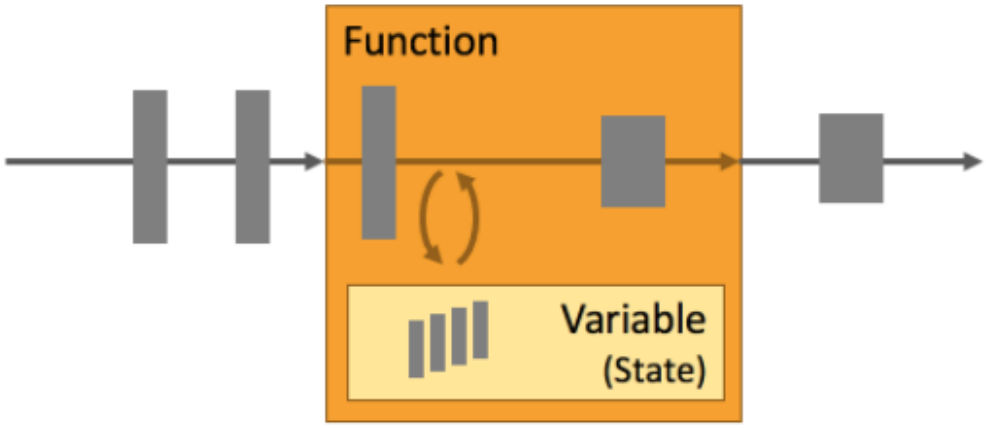

Figura 3.5. State of Flink applications[5]

Flink fornisce diverse funzionalità per per gestirlo:

  - Primitive multiple di stato : Apache Flink possiede delle primitive di stato per varie strutture dati. Lo sviluppatore può utilizzare la primitiva di stato migliore in base a ciò che deve fare.
  - Back-end di stato collegabile: Lo stato dell'applicazione viene ispezionato e gestito da un back-end di stato collegabile. Apache Flink possiede vari back-end di stato che memorizzano lo stato all'interno della memoria o su disco.
  - Coerenza dello stato esattamente una volta: Gli algoritmi di ripristino e di checkpoint di Flink in caso di errore garantiscono la congruenza dello stato dell'applicazione. Gli errori vengono risolti trasparentemente senza influenzare l'applicazione.
  - Stato molto grande: Apache Flink, grazie al proprio algoritmo di checkpoint, è capace di mantenere lo stato dell'applicazione nell'ordine di terabyte.
  - Applicazioni scalabili: Apache Flink garantisce la scalabilità delle applicazionei stateful assegnando lo stato a diversi job.

- Tempo: Il tempo è un altro aspetto fondamentale della applicazioni di streaming. I flussi di eventi sono caratterizzati dal fatto di essere prodotti in uno specifico momento e molte operazioni sui flussi sono basate sul tempo. Un'altra caratteristica rilevante dell'elaborazione dei flussi è come le applicazioni misurano il tempo. Per misurazione del tempo viene indicata la differenza tra il tempo dell'evento e quello di elaborazione. Flink possiede diverse funzionalità legate al tempo.
  - Modalità event-time: Le applicazioni che elaborano i flussi con questa semantica determinano i risultati utilizzando i timestamp degli eventi. Quindi, l'elaborazione degli eventi basati sul tempo permette di ottenere risultati precisi e coerenti a prescindere se sono in tempo reale oppure no.
  - Supporto per i watermark: Apache Flink utilizza i watermark per riflettere sul tempo delle applicazioni event-time. I watermark permettono di avere una buona latenza garentendo la correttezza dei risultati.
  - Gestione tardiva dei dati: Quando vengono elaborati flussi con la modalità event-time con i watermwark, può succedere che un'operazione venga completata prima che siano giunti tutti gli eventi associati ad essa. Questi eventi prendono il nome di eventi tardivi e Apache Flink offre diverse opzioni per gestirli.
  - Modalità processing-time: In aggiunta alla modalità event-time, Apache Flink possiede la semantica del tempo di elaborazione che permette di eseguire delle operazioni avviate da un orologio della macchina di elaborazione. Questa modalità viene utilizzata dalle applicazioni che possiedono requisiti di bassa latenza.

3.2.6 API a più livelli

Apache Flink possiede tre API a più livelli e ognuna può essere sfruttata in diversi casi d'uso.

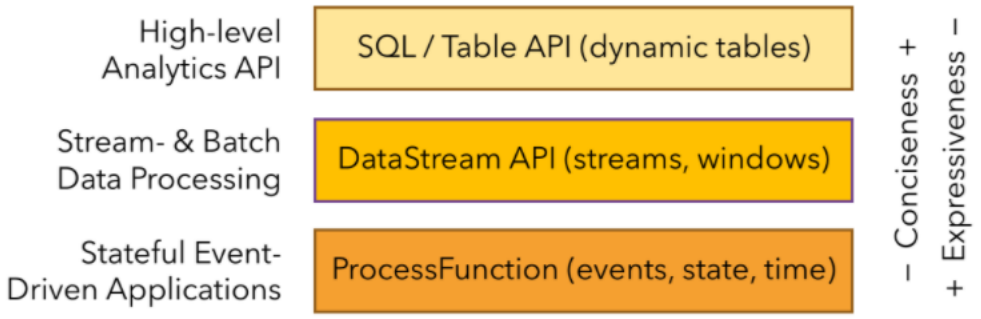

Figura 3.6. Flink APIs[5]

- ProcessFunctions: Le ProcessFunctions sono interfacce di funzioni fornite da Apache Flink per gestire i singoli eventi da uno o più flussi di input o eventi raggruppati all'interno di una finestra. Inoltre, consentono di monitorare tempo e stato. Una ProcessFunction può cambiare il proprio stato e realizzare una logica di business, per ogni evento, molto articolata.
- API DataStream: L'API DataStream possiede diverse primitive per effettuare operazioni di elaborazione su flussi. Questa API, presente per Java e Scala, possiede funzioni come reduce(), map() e aggregate().
- SQL and Table API: Apache Flink possiede due API relazionali, l'API table e l'API SQL, raggruppate per elaborare flussi. Le query vengono fatte sfruttando la semantica dei flussi limitati o quella dei flussi illimitati e il risultato prodotto in entrambi i casi è lo stesso. Entrambi le API possono essere integrate con le API DataStream. Le API relazionali di Apache Flink sono nate per semplificare le pipeline e le analisi dei dati.

3.2.7 Librerie

Apache Flink possiede delle librerie per l'elaborazione dei dati. Queste librerie generalmente si trovano all'interno delle API e non sono del tutto indipendenti.

- Elaborazione di eventi complessi (CEP): Questa libreria è inserita all'interno dell'API DataStream e consente di indicare modelli di eventi.
- API DataSet: Questa è l'API più importante di Apache Flink per l'elaborazione batch delle applicazioni. Gli algoritmi che sfruttano questa API sono nati prendendo spunto dagli operatori dei tradizionali database.
- Gelly: Questa è una libreria per l'elaborazione di grafici scalabili ed è predente all'interno dell'API DataSet.

3.2.8 Eseguire applicazioni non-stop 24 ore su 24, 7 giorni su 7

I guasti alle macchine e ai processi sono spesso presenti all'interno dei sistemi distribuiti. Un framework di elaborazione distribuito come Apache Flink[7] deve essere capace di riprendersi dai guasti ed eseguire le applicazioni di streaming 24 ore su 24, 7 giorni su 7. Bisogna garantire che lo stato dell'applicazione rimanga coerente per far si che l'applicazione possa riprendere l'elaborazione. Apache Flink possiede alcune caratteristiche che permettono alle applicazioni di essere coerenti e continuare a funzionare:

- Checkpoint coerenti: Apache Flink gestisce il ripristino mediante checkpoint coerenti dello stato dell'applicazione. Se dovesse verificarsi un errore, l'applicazione viene riavviata e lo stato viene preso dall'ultimo checkpoint.
- Checkpoint efficienti: Questo tipo di checkpoint viene sfruttato da Apache Flink per ripristinare lo stato dell'applicazione ma cercando di non farlo in modo troppo costoso.

- End-to-End Exactly-Once: Apache Flink possiede sink transazionali che permettono ai dati di essere scritti soltanto una volta, anche se dovessero esserci errori.
- Integrazione con i gestori di cluster: Apache Flink è assolutamente integrato con i gestori di cluster. Se un processo dovesse fallire, ne viene istantaneamente iniziato un nuovo che subentra nel suo lavoro.
- Configurazione ad alta disponibilità: Apache Flink possiede una direttiva ad alta disponibilità che gli permette di cancellare tutti i punti di errore.

3.2.9 Aggiornare, migrare, sospendre e riprendere le applicazioni

Le applicazioni di streaming possono spesso avere dei bug che devono essere corretti e devono essere fatti miglioramenti introducendo nuove funzionalità. Purtroppo aggiornare un'applicazione di streaming non è molto semplice. I Savepoint di Apache Flink sono una funzionalità molto potente ed anche rara che permettono di risolvere il problema dell'aggiornamento delle applicazioni stateful. Un Savepoint può essere utilizzato per avviare un'applicazione attabile con lo stato e poi inizializzarne uno nuovo. I Savepoints ammettono le seguenti funzionalità:

- Evoluzione dell'applicazione: I Savepoints possono essere sfruttati per migliorare le applicazioni.
- Migrazione del cluster: Sfruttando i Savepoints, le applicazioni possono essere trasferite in altri cluster.
- Aggiornamenti della versione di Apache Flink: Un'applicazione può essere trasferita per essere eseguita su una nuova versione di Apache Flink sfruttando un Savepoint.
- Ridimensionamento dell'applicazione: I Savepoints permettono di aumentare o diminuire il parallelismo di un'applicazione.
- Test A/B e scenari ipotetici: Le prestazioni o la qualità di più versioni diverse di un'applicazione possono essere confrontate avviandole dallo stesso Savepoint.
- Pausa e ripresa: Un'applicazione può essere interrotta fermando un Savepoint e successivamente essere ripresa dal Savepoint.
- Archiviazione: I Savepoints, per ripristinare lo stato di un'applicazione, possono essere archiviati.

3.2.10 Monitorare e controllare le applicazioni

Le applicazioni di streaming in esecuzione necessitano di essere supervisionate mediante strumenti di monitoraggio per prevenire eventuali problemi. Apache Flink fornisce un'API REST per verificare le applicazioni.

- Interfaccia utente web: Questa interfaccia consente ad Apache Flink di controllare, monitorare ed eseguire il debug delle applicazioni in esecuzione.
- Registrazione: Apache Flink utilizza l'interfaccia di registrazione slf4j e il framework di registrazione log4j.
- Metriche: Apache Flink possiede un sistema di metriche complesso utilizzato per raccogliere e segnalare metriche di sistema.
- API REST: Apache Flink possiede un'API REST sfruttata per sottoscrivere una nuova applicazione, prendere il Savepoint di un'applicazione in esecuzione o annullare un'applicazione.

3.2.11 Operatori di Apache Flink

Gli operatori di Apache Flink[8] cambiano uno o più DataStream in un altro DataStream. I programmi possono sfruttare più operatori per modificare il flusso di dati. Gli operatori più importanti che la DataStream API fornisce, sono i seguenti:

- Map: Prende un elemento in input e genera nuovo elemento. Ad esempio, dato un flusso di dati interi, il valore di ogni dato può essere triplicato.
- FlatMap: Prende un elemento in input e genera zero, uno o più elementi. Ad esempio, data una serie di frasi può dividerle in parole.
- Filter: Applica una funzione booleana ad ogni elemento e mantiene quelli per cui la funzione ritorna true. Ad esempio, si può selezionare tutti gli elementi diversi da uno.
- KeyBy: Prende un flusso e lo partiziona, a livello logico, in diverse partizioni. Ognuna di queste partizioni conterrà soltanto gli elementi che posseggono la stessa chiave. L'istanza di DataStream verrà trasformata in un'istanza KeyedStream.
- Reduce: Combina ogni elemento con il precedente ridotto ottenendo un unico valore finale. Ad esempio, se viene divisa una frase in parole permette di poter contare il numero di occorrenze di ogni parola all'interno della frase.
- Window: Consente di definire una finestra su un KeyedStream già partizionato. Raggruppa i dati all'interno di ogni chiave in base ad una determinata caratteristica. Questo operatore, siccome assume molta importanza nella realizzazione della streaming platform del progetto, successivamente verrà analizzato più in dettaglio.
- Aggregations: Aggrega in sequenza flussi di dati con la stessa chiave. Ad esempio, operazioni di aggregazione possono essere la somma, la ricercare del minimo o del massimo.
- Union: Consente di unire due o più flussi di dati creandone uno nuovo contenente gli elementi di tutti i flussi.

- Split: Consente di dividere un flusso in più flussi in base a una condizione.
- Iterate: Consente di eseguire nuovamente delle operazioni su un flusso di dati che viene aggiornato ad ogni operazione.
- Extract Timestamps: Consente di estrarre il timestamp da un record. Questo operatore è utile quando si lavora con finestre che utilizzano la semantica event-time.

Un operatore molto importante di Apache Flink è Windows. Questo operatore consente di dividere un flusso di dati in finestre di dimensioni finite, su sui applicare dei calcoli. Una finestra viene creata appena arriva il primo elemento che deve far parte di questa finestra e viene cancellata quando l'ora dell'evento o il tempo di elaborazione oltrepassa il timestamp di fine sommato ad un eventuale ritardo consentito indicato dall'utente. La prima cosa da fare quando viene creata una finestra è specificare se il flusso deve essere codificato oppure no. Se non viene chiamato KeyBy() il flusso non viene codificato mentre se viene chiamato il flusso infinito viene suddiviso il diversi flussi logici. Nei flussi con chiave può essere utilizzato qualsiasi attributo come chiave. Avere dei flussi con chiave consente di eseguire calcoli in parallelo da diverse attività perchè ogni flusso separato per chiave può essere eseguito incondizionatamente dal resto. Mentre nei flussi senza chiave, si avrà un unico flusso che non verrà diviso in flussi logici e un'unica attività che si occuperà si eseguire il flusso logico. Una volta fatto questo, bisogna definire un assegnatore di finestre. Un WindowAssigner si occupa di assegnare qualunque elemento in ingresso a una o più finestre. Le finestre basate sul tempo per indicare la dimensione della finestra utilizzano il timestamp di inizio e il timestamp di fine. Apache Flink possiede dei WindowAssigners predefiniti ma è possibile anche implementarli estendendo la classe WindowAssigner. I WindowAssigners predefiniti sono:

- Tumbling Windows: Un tumbling windows assigner assegna ogni singolo elemento ad una finestra con una dimensione specificata. Queste finestre posseggono una dimensione fissa e non si accavallano. Ad esempio, si può avere una tumbling window di 5 minuti dove ogni 5 minuti viene valutata la finestra corrente e ne viene avviata una nuova.
- Sliding Windows: Lo sliding windows assegna gli eventi a finestre con una lunghezza fissa. La dimensione viene configurata dal parametro window size e dal parametro window slide. Il parametro window slide indica la frequenza con cui la Sliding Window viene avviata. Queste finestre possono accavallarsi se la slide è minore della dimensione della finestra. Se dovesse accadere, gli elementi vengono assegnati a diverse finestre.
- Session Windows: Un session window assigner raccoglie gli eventi per sessioni di attività. Queste finestre non si accavallano e non hanno sia un'ora di inizio sia un'ora di fine fisse. Questo tipo di finestra viene chiusa quando non riceve elementi per certo tempo ossia quando c'è stato un intervallo di inattività.
- Global windows: Il global windows assigner assegna ogni elemento che ha la stessa chiave alla singola Global Window.

Dopo aver specificato il WindowAssigner, bisogna indicare l'operazione che si vuole effettuare su ogni finestra. Il compito di effettuare l'operazione spetta alla window function, che viene adoperata per elaborare gli elementi di ogni finestra appena una è pronta per essere elaborata. La window function può essere AggregateFunction, ReduceFunction, ProcessWindowFunction o FoldFunction. Siccome Apache Flink è capace di aggregare incrementalmente tutti gli elementi per ogni finestra quando arrivano, le prime due funzioni possono essere eseguite efficacemente. Una ProcessWindowFunction, invece, non è molto efficiente perché Apache Flink deve memorizzare tutti gli elementi di una finestra prima di chiamare la funzione. Questo tipo di funzione riceve un Iterable per tutti gli elementi all'interno della finestra e delle informazioni su di essa.

- AggregateFunction: Possiede tre tipi che sono uno di input (IN), uno di output (OUT) e un accumulatore (ACC). Il tipo di input indica il tipo degli elementi nel flusso di dati di input e l'AggregateFunction possiede un metodo che le consente di aggiungere all'accumulatore un elemento di input. L'interfaccia contiene anche un metodo per creare un accumulatore, per fondere due accumulatori e per togliere da un'accumulatore un'uscita. Appena giungono gli elementi di input in una finestra, Apache Flink li aggrega incrementalmente.
- ReduceFunction: Indica come vanno combinati due elementi di input per generarne un terzo dello stesso tipo.
- ProcessWindowFunction: Riceve un Iterable con ogni elemento della finestra e un oggetto Context che consente di accedere alle informazioni su ora e stato. Questo, purtroppo, va a discapito delle prestazioni perché gli elementi non non possono essere aggregati incrementalmente ma devono essere inseriti in un buffer fino a che la finestra non è pronta per essere elaborata.
- FoldFunction: Indica come un elemento di input deve essere combinato con un elemento di output dello stesso tipo. Questa funzione, per tutti gli elementi della finestra, viene chiamata incrementalmente. Siccome inizialmente non si ha un valore di output, il valore di input viene combinato con un valore predefinito.

Per stabilire quando una finestra può essere elaborata da una window function, si può anche assegnare un trigger al WindowAssigner.

3.3 Apache CouchDB

Apache CouchDB[10] è un database non relazionale che archivia i dati in documenti JSON. Siccome i dati sono archiviati in documenti JSON, la struttura dei dati o dei documenti può essere modificata dinamicamente. Si può accedere ai documenti mediante browser web tramite HTTP. Per interrogare, combinare e trasformare documenti viene utilizzato JavaScript. I dati possono essere distribuiti efficacemente ed è presente un accertamento automatico dei problemi. Apache CouchDB possiede diverse funzionalità, come la modifica dei documenti quando necessario oppure le notifiche quando avvengono modifiche in tempo reale. Inoltre, è anche molto disponibile, tollerante agli errori, coerente e garantisce la sicurezza dei dati.

3.3.1 Panoramica di Apache CouchDB

I database che archiviano i documenti si trovano all'interno di un server CouchDB. Ciascun documento possiede un nome univoco all'interno del database e Apache CouchDB offre un'API HTTP RESTful per leggere, modificare, aggiungere, aggiornare o eliminare i documenti all'interno del database. I documenti sono l'unità dati principale di Apache CouchDB e possono contenere qualunque numero di campi e allegati. I campi all'interno del documento posseggono un nome univoco e possono contenere diversi tipi di valori. Inoltre, non è presente un limite sulla dimensione del testo o del numero di elementi. Le modifiche ai documenti vengono effettuate da applicazioni client che caricano i documenti, li modificano e li salvano nuovamente nel database. Le modifiche fatte ai documenti o hanno successo o falliscono completamente cioè nel database non possono esserci documenti modificati o salvati parzialmente. Apache CouchDB possiede le proprietà Atomic Consistent Isolated Durable (ACID). I dati su disco non vengono sovrascritti e viene garantito che il database rimanga sempre in uno stato coerente. I documenti possono essere letti da un qualunque numero di client senza mai essere interrotti o bloccati. I dati sono già efficientemente impacchettati per l'archiviazione invece di essere divisi in tabelle e questo rappresenta un vantaggio per i documenti. Quando il file del database spreca una determinata quantità di spazio, i dati vengono clonati in nuovo file e quello vecchio viene eliminato quando tutti i dati sono stati copiati in quello nuovo. Nei database relazionali i dati vengono organizzati in tabelle mentre in Apache CouchDB vengono archiviati in documenti semi-strutturati che sono flessibili e posseggono una propria struttura. Apache CouchDB possiede un modello di visualizzazione chiamato viste che consente di affrontare il problema dell'aggiunta di dati a strutture semi-strutturate. Le viste sono una riproduzione dinamica del contenuto di un documento di un database e possono essere lette, interrogate e aggiornate.

3.3.2 Perché Apache CouchDB?

Apache CouchDB compare in una moderna generazione di sistemi per la gestione dei database ed è stato progettato per gestire traffico instabile. Unisce un modello evidente di archiviazione di documenti con un motore di query molto potente in un modo relativamente semplice. Il design di Apache CouchDB è molto simile a quello di un'architettura web con concetti di rappresentazione, metodi e risorse. Questo, però, viene integrato con metodi importanti per interrogare, combinare e filtrare dati. Inoltre, viene aggiunta una scalabilità enorme e una tolleranza agli errori. Database relazionali necessitano di dati modellati in anticipo mentre Apache CouchDB possiede un design povero di schemi che consente di aggregare i dati successivamente. È enormemente flessibile, possiede elementi per creare un sistema capace di soddisfare diversi problemi e la sua funzione principale è uniformare più database. Questa funzione gli consente di sincronizzare efficacemente database tra diverse macchine per l'archiviazione dei dati, distribuire i dati tra luoghi fisicamente lontani e distribuire i dati a un cluster di istanza CouchDB che hanno in comune una parte di richieste che giungono al cluster. Apache CouchDB, inoltre, consente di gestire gli errori.

3.3.3 Coerenza finale

Apache CouchDB si differenzia da altri sistemi distribuiti garantendo la consistenza finale piuttosto che preferire la consistenza assoluta alla disponibilità grezza. Garantisce scalabilità e per farlo è dotato di un meccanismo utile e intuitivo che gli consente di configurare le varie applicazioni. I database relazionali necessitano di controlli per ogni operazione al fine di garantire la consistenza mentre Apache CouchDB offre una facile realizzazione di applicazioni che preferiscono dei notevoli miglioramenti delle prestazioni alla coerenza istantanea. Apache CouchDB convalida i documenti all'interno del database con funzioni Javascript e facendo fare a lui questo lavoro si risparmia un gigantesco numero di cicli di CPU.

Capitolo 4

Implementazione della Streaming Platform

In questo capitolo verrà visto come è stata implementata la streaming platform capace di catturare e classificare il traffico RTC all'interno di una rete e la realizzazione di ogni suo singolo livello.

4.1 Realizzazione del Data Ingestion Layer

Per la realizzazione del Data Intestion Layer, livello che consente di acquisire dati da sorgenti esterne, non è stato utilizzato nessuno strumento della famiglia Apache ma sono stati realizzati due script Python. Il primo script consente di leggere pacchetti di rete da un file di cattura Wireshark con estensione .pcap e un secondo script capace invece di catturare in tempo reale il traffico all'interno di una rete utilizzando TShark.

4.1.1 Lettura di un file .pcap

Un file .pcap è un file con estensione associata soprattutto a Wireshark, programma adoperato per osservare le reti, contenente i dati dei pacchetti di rete. Nella streaming platform si andrà a leggere questo tipo di file di cattura, selezionare i flussi RTP mediante tecniche basate su DPI (deep packet inspection), estrapolare i campi principali RTP e UDP del pacchetto, e infine calcolare un file JSON per ogni pacchetto contenente i campi estrapolati per poi fornirlo al livello successivo. Lo script Python realizzato per leggere i pacchetti all'interno del file .pcap utilizza TShark[3] che è un analizzatore di protocollo di rete. TShark è semplicemente Wireshark ma utilizzato da riga di comando e trasforma il file di cattura .pcap in un formato che Python è capace di comprendere. Nello script realizzato è stato definito un comando TShark e successivamente si andrà ad effettuare la lettura del file .pcap utilizzando questo comando che consente di prendere i campi, necessari alla classificazione del traffico, dai vari pacchetti buttando via il resto. Il comando TShark utilizzato è il seguente:

```
command = f"""tshark -r {source_pcap} -Y {filtro} \
                     -T fields {" ".join(port_add)} \
                     -E separator=? \
                     -E header=y \
                     -e frame.time_epoch -e frame.number \
                     -e frame.len -e udp.srcport \
                     -e udp.dstport  -e udp.length \
                     -e rtp.p_type  -e rtp.ssrc \
                     -e rtp.timestamp  -e rtp.seq  \
                     -e rtp.marker -e rtp.csrc.item -e \
                     ip.src -e ipv6.src -e ip.dst -e ipv6.dst \
                     --enable-heuristic rtp_stun"""
```

- -r: Questa opzione, seguita dal file .pcap da leggere, quando vengono letti i pacchetti dal file permettere di visualizzare su standard output una riga di riepilogo per ogni pacchetto letto.
- -Y: Permette di aggiungere un filtro, nel caso di questo script sarà rtp.version==2. Si andranno a considerare soltanto i pacchetti RTP versione 2.
- -T: Permette di aggiungere un campo alla lista dei campi da visualizzare siccome fields è stato selezionato.
- -E: Determina un'opzione che esamina la stampa dei campi quando è selezionato -T fields. E' seguito da separator e header che sono entrambe due opzioni. Separator separa i vari campi con il carattere indicato e nel comando command è "?" mentre header che è uguale a "y" permette di stampare i campi catturati da -e su riga di comando.
- -e: Siccome -T fields è selezionato, aggiunge un campo alla lista di quelli da visualizzare. In questo script si andrà a visualizzare time_epoch che è il timestamp del pacchetto quando è stato catturato, number che è il frame number del pacchetto, la lunghezza del pacchetto, la porta sorgente udp, la porta destinazione udp, la lunghezza del pacchetto udp, il payload type, l'ssrc, il timestamp, il seguence number, il marker rtp, il csrc, l'ip sorgente e l'ip destinazione che possono essere ipv4 o ipv6. Si può notare che tra i vari campi RTP compaiono anche alcuni UDP perchè il protocollo RTP è basato sul protocollo UDP ovvero i pacchetti RTP viaggiano in pacchetti UDP.

Alla fine del comando è stato abilitato heuristics che si occupa di scaricare le decodifiche euristiche. Ci sarà un record per ognuna delle righe e i campi saranno delimitati da tabulazioni. Con questo comando TShark sono state definite delle delle opzioni e i campi che devono essere estratti da ogni pacchetto. Successivamente viene effettuata la seguente operazione che consente di leggere il file .pcap:

```
o,e = subprocess.Popen(command, encoding='utf-8',
                       stdout=subprocess.PIPE,
                       stderr=subprocess.PIPE,
                       shell=True).communicate()
```

La subprocess consente di creare nuovi processi, collegarsi alle pipe di output/error e di acquisire i codici di errore. La creazione e la gestione dei processi viene gestita dall'interfaccia Popen e si è scelto di utilizzarla perché consente di avere molta flessibilità. Il processo creato andrà a leggere il file .pcap indicato nel comando ed estrapolerà per ogni singolo pacchetto i campi selezionati. Il primo argomento è il comando da eseguire, il secondo indica il tipo di codifica che si vuole per l'output mentre i due successivi sono stdout e stderr. Infine, sulla Popen viene fatto un .communicate() che consente di interagire con il processo inviando dati a stdin, di leggere i dati da stdout/stderr e di aspettare la terminazione del processo. Il risultato di questa operazione, memorizzato all'interno di "o", sono i campi RTP versione 2 di ogni singolo pacchetto indicati dal comando e separati l'uno dall'altro dal separatore "?". Se si andasse a stampare il valore di "o" su stdout, si ha il seguente output: 1587404838.186145000?50972?233?5004?64693?199?111?0x431b827a?7977?7977?0?? 69.26.161.221??192.168.1.105? per ogni singolo pacchetto. Al livello successivo, come detto precedentemente, bisogna fornire per ogni pacchetto un file JSON contente i campi estrapolati. Si potrebbe manipolare il risultato della subprocess creando i file JSON per ogni pacchetto ma viene presa una strada più semplice che consiste nell'eseguire un'ulteriore operazione prima di creare i file JSON. Questa funzione prende il nome di read_csv che fa parte dello strumento di analisi Pandas. Si è scelto di percorrere questa strada per avere dei dati più semplici e facili da utilizzare. La funzione read_csv è stata utilizzata nel seguente modo:

```
df = pd.read_csv(StringIO(o), header=0, sep"?", low_memory = False)
```

Questa funzione crea una tabella dove ogni colonna rappresenta i campi estrapolati dai pacchetti e ogni riga contiene i valori di questi campi.

- StringIO(o): Rappresenta l'oggetto da cui vengono estratti i dati per creare la tabella.
- header: Indica i numeri di riga da utilizzare come nomi per le colonne e inizio dei dati. Il comportamento predefinito deduce il nome delle colonne: se non viene passato nessun nome il comportamento è uguale a quello di header=0 e i nomi delle colonne vengono ipotizzati dalla prima riga del file mentre se i nomi delle colonne sono passati chiaramente il modo di agire è come quello di header=nothing. Utilizzando header=0 viene denotata la prima dei dati invece della prima riga del file.
- sep: Indica il separatore tra i vari campi e nel comando command era stato utilizzato "?".
- low_memory: Indica se vengono utilizzati oppure no tipi misti. Avendolo settato a false viene garantita la mancanza di tipi misti.

La tabella creata è la seguente:

	frame.time_epoch	frame.number	...	ip.dst	ipv6.dst
0	1.587405e+09	333	...	69.26.161.221	NaN
1	1.587405e+09	338	...	69.26.161.221	NaN
2	1.587405e+09	340	...	69.26.161.221	NaN
3	1.587405e+09	342	...	69.26.161.221	NaN
4	1.587405e+09	343	...	69.26.161.221	NaN
	...	...	...	...	...
46728	1.587405e+09	50970	...	192.168.1.105	NaN
46729	1.587405e+09	50971	...	192.168.1.105	NaN
46730	1.587405e+09	50972	...	192.168.1.105	NaN
46731	1.587405e+09	50973	...	192.168.1.105	NaN
46732	1.587405e+09	50974	...	192.168.1.105	NaN

Si può notare che i valori della colonna ipv6.dst sono tutti uguali a "NaN". Un pacchetto può essere ipv4 o ipv6 quindi uno dei due campi viene settato a "NaN". Per migliorare la leggibilità e semplicità di questa tabella, utilizzando questi due comandi:

```
df["ip.src"] = df["ip.src"].fillna(df["ipv6.src"])
df["ip.dst"] = df["ip.dst"].fillna(df["ipv6.dst"])
```

si va ad accorpare la colonna ip.src, che indica l'ip sorgente ipv4, con la colonna ipv6.src e la colonna ip.dst, che indica l'ip destinazione ipv4, con la colonna ipv6.dst evitando di avere valori uguali a "NaN". A questo punto si può leggere riga per riga la tabella generata, creare il file JSON contenente le informazioni di ogni singola riga e fornirlo al livello successivo. Con:

```
parsed = json.loads(row.to_json())
```

si va a creare un file JSON contenente i valori della riga corrente e poi viene deserializzato in un oggetto Python. Se i dati da deserializzare non sono un oggetto JSON valido, viene generato un errore. Avendo a disposizione un file JSON non resta che inviarlo al livello successio e per farlo viene creato un produttore Kafka che si occupa di effettuare questa operazione. Il produttore Kafka viene creato nel seguente modo:

```
producer = KafkaProducer(bootstrap_servers = KAFKA_HOSTS,
            api_version = KAFKA_VERSION)
```

- bootstrap_servers: Deve possedere perlomeno un broker che risponda a una richiesta API per i metadati e nella funzione KafkaProducer indicata sarà uguale a KAFKA_HOSTS che è una costante e vale localhost:9092
- api_version: Indica la versione dell'API Kafka da utilizzare. Nella funzione KafkaProducer indicata può essere un valore tra 10 e 0 indicato dalla costante KAFKA_VERSION

Il produttore producer appena creato, per fornire il seguente file JSON al livello successivo, utilizzerà la funzione send implementata nel modo seguente:

```
producer.send('flink-topic', json.dumps(parsed).encode('ascii'))
```

Il primo parametro della send indica il topic Kafka su cui viene inviato il file JSON mentre il secondo è il file JSON da inviare però serializzato mediante "dumps". Viene riportato in Codice 6.1 il codice completo dello script Python che consente di leggere un file di una cattura Wireshark con estensione .pcap.

4.1.2 Cattura del traffico in tempo reale

Il secondo script Python realizzato consente di catturare traffico in tempo reale all'interno di una rete, selezionare i flussi RTP mediante tecniche basate su DPI (deep packet inspection), estrapolare i campi principali RTP e UDP del pacchetto, e infine calcore un file JSON per ogni pacchetto contenente i campi estrapolati per poi fornirlo al livello successivo. Per la cattura del traffico in tempo reale è stato utilizzato Pyshark che è un wrapper Python per TShark e permette l'analisi dei pacchetti Python sfruttando i dissettori Wireshark. Per effettuare questo tipo di cattura con Pyshark viene utilizzato il seguente comando:

```
capture = pyshark.LiveCapture(interface='5')
```

Successivamente con un ciclo for si va ad analizzare in tempo reale tutti i pacchetti che vengono catturati. Il ciclo for è stato strutturato nel seguente modo:

```
for packet in capture.sniff_continuously():
    try:
        if packet.transport_layer == "UDP":
            version, padding, extension, CC, marker,
            payload_type, seq_number,rtp_timestamp,
            ssrc, no_parsing = RTP_layer(packet)
            if version == 2:
                json_kafka =
                    {'timestamps' : packet.frame_info.time,
                     'ip_src' : packet.ip.src,
                     'ip_dst' : packet.ip.dst,
                     'prt_src' : packet.udp.srcport,
                     'prt_dst' : packet.udp.dstport,
                     'len_udp' : packet.frame_info.len,
                     'rtp_timestamp' : rtp_timestamp,
                     'rtp_seq_num' : seq_number ,
                     'ssrc' : ssrc,
                     'p_type' : payload_type,
                     'rtp_marker' : marker}
                producer =
                    KafkaProducer(bootstrap_servers=KAFKA_HOSTS,
                    api_version=KAFKA_VERSION)
                producer.send('flink-output',
                              json.dumps(json_kafka).
                                encode('ascii'))
        except Exception as e:
            pass
```

All'interno di una rete può essere presente del traffico molto vario dove una parte utilizza come protocollo di trasporto TCP mentre l'altra UDP. Siccome il traffico RTP utilizza come protocollo di trasporto UDP, il primo controllo che viene effettuato sul traffico appena catturato è se utilizza il protocollo UDP. In caso affermativo, viene chiamata la funzione RTP_layer che estrae dal pacchetto i campi RTP necessari alla classificazione del traffico mentre i campi UDP necessari vengono estratti direttamente dal pacchetto. Successivamente, una volta estratti questi campi dal pacchetto, vengono inseriti all'interno di un file JSON e viene fornito al livello successivo. Appena creato il file JSON, viene controllato che la versione del protocollo RTP sia la seconda e in caso affermativo viene creato un KafkaProducer e viene inviato questo file JSON sul topic come è stato fatto nello script precedente. La funzione chiamata RTP_layer, che estrarre dal pacchetto i campi RTP e UDP, è stata implementata nel seguente modo:

```
def RTP_layer(Packet):
    try:
        payload = bin(int(str(Packet.udp.payload).
            replace(":",""),16))
        version = int(payload[2:4], 2)
        padding = int(payload[4])
        extension = int(payload[5])
        CC = int(payload[6:10],2)
        marker = int(payload[10])
        payload_type = int(payload[11:18],2)
        seq_number = int(payload[18:34],2)
        timestamp = int(payload[34:66], 2)
        ssrc = hex(int(payload[66:98], 2))
        no_parsing = payload[98:]
        return version, padding, extension, CC,
        marker, payload_type, seq_number, timestamp,
        ssrc, no_parsing
    except:
        return [False]*10
```

La prima operazione che viene fatta è estrarre il payload UDP del pacchetto e trasformarlo in bit perché il pacchetto RTP è incapsulato nel payload UDP. Dal payload UDP vengono estratti i bit corrispondenti ai seguenti campi RTP:

- version: Corrisponde alla versione del protocollo RTP ed è compreso tra il bit 2 e il bit 4. Questo campo è molto importante perché, come detto precedentemente, vengono considerati soltanto i pacchetti RTP versione 2.
- padding: Specifica se alla fine del pacchetto c'è un bit di padding ed è il bit 4 del payload UDP.
- extension: Specifica se tra l'header standard e il payload è presente l'Exstension header. Questo campo è il bit 5 del payload UDP.
- CC: E' il CSRC Count e corrisponde al numero di CSRC. Esso è compreso tra il bit 6 e il bit 10 del payload UDP.

- marker: Esso è soltanto un bit e se settato indica al livello applicativo che questo pacchetto può essere importante per lui. E' il bit 10 del payload UDP.

- payload_type: Questo campo è caratteristico per ogni profilo RTP ed è compreso tra il bit 11 e il bit 18 del payload UDP.

- seq_number: Corrisponde al sequence number che viene incrementato per ciascun pacchetto di uno. Esso è compreso tra il bit 18 e il bit 24 del payload UDP.

- timestamp: Il timestamp consente al destinatario di riprodurre all'intervallo adatto il media ricevuto. Esso è compreso tra il bit 34 e il bit 66 del payload UDP.

- ssrc: L'SSRC identifica univocamente l'origine dello stream nella sessione RTP. Esso è compreso tra il bit 66 e il bit 98 del payload UDP.

- no_parsing: Al suo interno vengono inseriti tutti i bit non utilizzati del payload UDP che vanno dal 98 fino all'ultimo.

Viene riportato in Codice 6.2 il codice completo dello script Python che consente di catturare il traffico in una rete in tempo reale.

4.2 Realizzazione del Data Collector Layer

Nella realizzazione di questo livello è stato utilizzato Apache Kafka per svolgere la fase di message broker. E' stato creato un topic chiamato "flink-topic" che memorizza i dati ricevuti dallo script Python in esecuzione e li rende poi disponibili al livello successivo della streaming platform. Per poter essere utilizzato, Apache Kafka è stato avviato eseguendo i seguenti passi:

- E' stato aperto un terminale e lanciato il seguente comando per avviare lo ZooKeeper:

  ```
  bin/zookeeper-server-start.sh config/zookeeper.properties
  ```

- E' stato aperto un altro terminale e lanciato il seguente comando per avviare il Kafka broker service:

  ```
  bin/kafka-server-start.sh config/server.properties
  ```

- E' stato creato il topic "flink-topic", per memorizzare i dati ricevuti dallo script Python del livello precedente, lanciando da terminale il seguente comando:

  ```
  bin/kafka-topics.sh -create -topic flink-topic -bootstrap-server
  localhost:9092
  ```

Appena avviato l'ambiente di Apache Kafka, lo script Python può memorizzare sul topic i dati appena saranno pronti per essere trasferiti al Data Processing Layer.

4.3 Realizzazione del Data Processing Layer

In questo paragrafo verrà realizzato il Data Processing Layer che si occupa di elaborare le informazioni raccolte dal livello precedente. Per la realizzazione di questo livello sono state utilizzate le API DataStream che hanno consentito di importare i dati, provenienti dal livello precedente, da Apache Kafka e di integrare nel codice Java degli operatori indispensabili per l'elaborazione di questi dati. Questo livello, quindi, si occupa di: ottenere i dati dal topic Apache Kafka creato nel livello precedente, tirar fuori da ogni record dei dati ottenuti il valore dei principali campi RTP e UDP, elaborarli per ottenere le features e fornire queste features come input al classificatore per identificare i diversi tipi di sorgenti multimediali all'interno della rete o del file .pcap in base al tipo di script Python in esecuzione. Il risultato prodotto dal classificatore, ovvero un file JSON contenente per ogni flusso di traffico la sua chiave composta e il suo tipo, viene reso disponibile al livello successivo. Nel codice in basso viene riportato il corpo centrale del codice sviluppato per realizzare questo livello con le diverse funzioni che consentono di eseguire le varie operazioni appena elencate. Nei successivi sottoparàgrafi, invece, vengono viste in dettaglio tutte le diverse operazioni svolte.

```
String inputTopic = "flink-topic";
String consumerGroup = "lower";
String address = "127.0.0.1:9092";
StreamExecutionEnvironment environment =
    StreamExecutionEnvironment.getExecutionEnvironment();
 FlinkKafkaConsumer<Packet> flinkKafkaConsumer =
    createStringConsumerForTopic(
    inputTopic, address, consumerGroup);
DataStream<Packet> stringInputStream = environment
    .addSource(flinkKafkaConsumer);
DataStream<Feature> stream =
    stringInputStream
    .keyBy(value->value.getIp_src())
    .keyBy(value-> value.getSsrc())
    .keyBy(value->value.getIp_dst())
    .keyBy(value->value.getPrt_dst_STRING())
    .keyBy(value->value.getPrt_src_STRING())
    .keyBy(value->value.getP_type_STRING())
    .window(TumblingProcessingTimeWindows.of(Time.seconds(1)))
    .process(new CreateTableFunction())
    .keyBy(value->value.getIp_src())
    .keyBy(value->value.getSsrc())
    .keyBy(value->value.getIp_dst())
    .keyBy(value->value.getPrt_src())
    .keyBy(value->value.getPrt_dst())
    .keyBy(value->value.getP_type())
    .map(new CalculateJsonSmall())
    .map(new POST_Classificatore())
    .map(new Contact_Node_RED());
```

4.3.1 Flink Kafka Consumer

FlinkKafkaConsumer consente di indicare il topic da dove leggere i dati, un DeserializationSchema che deserializza i byte presenti nei dati dell'oggetto Java indicato, il broker Kafka e il consumer group id. Viene riportato il codice Java della streaming platform che consente di acquisire i dati dal topic Kafka in direzione di Flink:

```
String inputTopic = "flink-topic";
String consumerGroup = "lower";
String address = "127.0.0.1:9092";
StreamExecutionEnvironment environment =
    StreamExecutionEnvironment.getExecutionEnvironment();
FlinkKafkaConsumer<Packet> flinkKafkaConsumer =
    createStringConsumerForTopic(
        inputTopic, address, consumerGroup);
DataStream<Packet> stringInputStream = environment
    .addSource(flinkKafkaConsumer);

public static FlinkKafkaConsumer<Packet>
    createStringConsumerForTopic(
    String topic, String kafkaAddress, String kafkaGroup ) {
      Properties props = new Properties();
      props.setProperty("bootstrap.servers",
      kafkaAddress);
      props.setProperty("group.id",kafkaGroup);
      FlinkKafkaConsumer<Packet> consumer =
      new FlinkKafkaConsumer<Packet>(
         topic, new PacketDeserializationSchema(),
         props);
      return consumer;
   }
```

FlinkKafkaConsumer consente di indicare il topic da dove leggere i dati, un DeserializationSchema che deserializza i byte presenti nei dati dell'oggetto Java indicato, il broker Kafka e il consumer group id. Viene riportato il codice Java della streaming platform che consente di acquisire i dati dal topic Kafka in direzione di Flink:

4.3.2 KeyBy

All'interno di una rete è possibile avere differenti tipi di traffico. Per indentificarli è stata utilizzata una chiave composta da i seguenti sei campi:

- ssrc
- ip sorgente
- ip destinazione
- porta sorgente

- porta destinazione
- payload type

Tutti i flussi di pacchetti ricevuti, grazie all'utilizzo di KeyBy, verranno partizionati in partizioni disgiunte e tutti i pacchetti con la stessa chiave verranno assegnati alla stessa partizione.

4.3.3 Tumbling processing-time windows

Un tumbling windows assigner consegna ogni dato a una finestra con una dimensione ben precisa. Questi tipi di finestre posseggono una dimensione fissa e non si sovrappongono. In questa streaming platform ci saranno delle tumbling windows da 1s che consentiranno

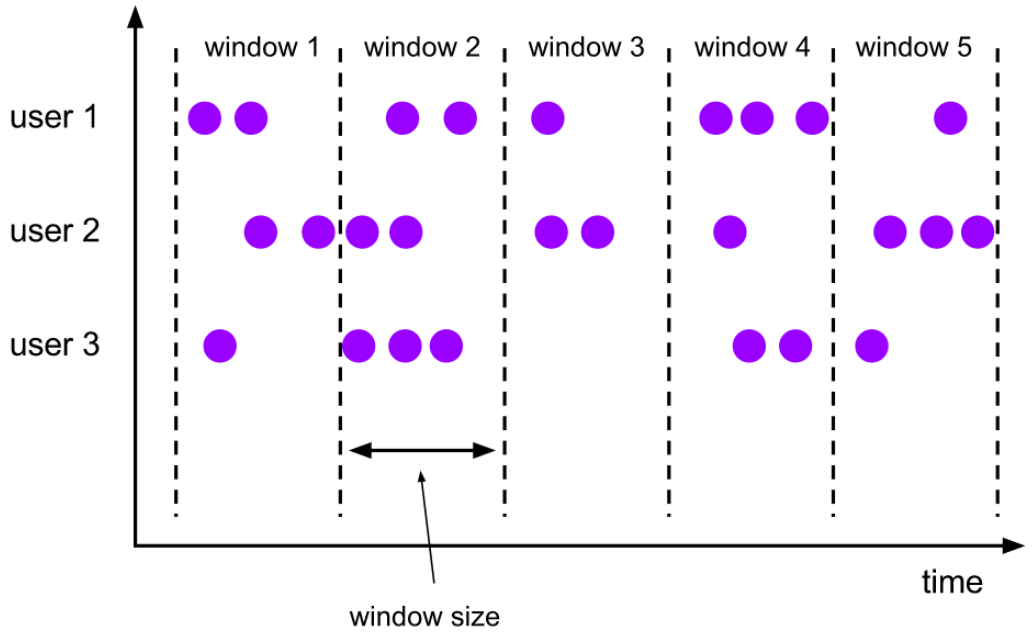

Figura 4.1. Tumbling Windows [4]

di elaborare i pacchetti secondo per secondo. Inoltre, grazie alla KeyBy utilizzata precedentemente, al termine di ogni secondo verranno elaborati flussi di pacchetti suddivi per chiave.

4.3.4 ProcessWindowFunction with CreateTableFunction

Questo sottoparàgrafo rappresenta il cuore dell'elaborazione streaming di Flink in questa streaming platform realizzata perché secondo per secondo per ogni pacchetto vengono calcolate le varie features ben scelte da fornire come input al classificatore per identificare

il tipo di sorgente multimediale. Viene implementata una classe chiamata CreateTableFunction contenente il metodo process che riceve un Iterable con gli elementi della finestra e un oggetto Context che consente di accedere alle informazioni su ora e stato. Come era stato già detto, il metodo process non è molto efficiente perché i vari pacchetti che vengono ricevuti non possono essere elaborati subito ma vengono memorizzati in un buffer fino a quando la finestra non è pronta per essere elaborata. Purtroppo non si può fare a meno di questo metodo perché non bisogna elaborare ogni pacchetto singolarmente ma c'è bisogno per ogni pacchetto del suo precedente per calcolare il cosíddetto "inter" per alcuni campi ovvero dato un campo del pacchetto corrente viene fatta la differenza tra il valore di quel campo e di quello del pacchetto precedente. Il primo pacchetto non avendo un predecessore avrà inter pari a 0 per ogni campo di cui si prova a calcolarlo. Il calcolo dell'inter viene fatto sul timestamp del pacchetto, sul timestamp RTP e sulla lunghezza del pacchetto UDP. Per calcolare le features quindi si va ad elaborare l'inter di questi campi dei pacchetti mentre come campo del pacchetto viene elaborata soltanto la lunghezza del pacchetto UDP. Il seguente codice ci consente di calcolare i vari inter:

```
int count = 0;
Packet old = null;

for (Packet in: input) {
    if(count == 0) {
        in.setInterarrival(0.0);
      in.setInterlength(0);
        in.setRtp_interarrival(0);
    }else {
      in.setInterarrival((
         in.getTimestamps() - old.getTimestamps()));
      in.setInterlength((
         in.getLen_udp() - old.getLen_udp()));
      in.setRtp_interarrival((
         in.getRtp_timestamp() - old.getRtp_timestamp()));
      }
      count++;
      old = in;
    }
```

Dopo aver calcolato i vari inter, per ogni flusso di pacchetti secondo per secondo, vengono calcolate le varie features. Le features che vengono calcolate sono il numero di pacchetti persi, la somma degli rtp marker e i kilobit per secondo. Oltre queste vengono anche calcolate le featurese comuni e le features "speciali" sull'inter_timestamp, sull'inter_timestamp RTP e sull'inter della lunghezza del pacchetto UDP. Le features comuni sono il massimo, il minimo, la media, la deviazione standard, la curtosi, l'indice di simmetria, il momento centrale di ordine 3, il momento centrale di ordine 4 e il percentile25. Le features speciali, invece, sono:

- max_min_diff: Rappresenta la differenza tra il massimo e il minimo.

- max_min_R: Rappresenta il rapporto tra il valore assoluto del massimo e la somma del valore assoluto del minimo e del valore assoluto massimo ovvero |max| / (|max| + |min|). Nel caso sia il massimo che il minimo valgano zero esso varrà zero.
- min_max_R: Rappresenta il rapporto tra il valore assoluto del minimo e la somma del valore assoluto del minimo e del valore assoluto del massimo ovvero |min| / (|max| + |min|). Nel caso sia il massimo che il minimo valgano zero esso varrà zero.
- max_value_count_percent: Rappresenta il rapporto tra il numero di volte che compare il valore maggiormente presente e il numero totale di valori.
- len_unique_percent: Rappresenta il rapporto tra il numero totale di valori distinti e il numero totale di valori.

Per memorizzare le varie features, per ogni flusso di pacchetti con una determinata chiave, è stata creata una classe chiamata "future" che contiene come attributi le diverse features mentre come metodi i loro setter e getter. Con le seguenti righe di codice vengono calcolate le diverse features:

- Numero di pacchetti persi:

```
if(in.getRtp_seq_num()!=(old.getRtp_seq_num()+1))
    rtp_seq_num_packet_loss +=
    (in.getRtp_seq_num()-old.getRtp_seq_num());
```

  Queste righe di codice che calcolano il numero di pacchetti persi vengono inserite nel ramo else del for dove vengono calcolati i vari inter.

- Somma degli RTP marker:

```
feature.setRtp_marker_sum_check((int) StreamSupport
    .stream(input.spliterator(), false)
    .map(Packet::getRtp_marker)
    .reduce(0,Integer::sum));
```

- Kilobit per secondo:

```
feature.setKbps((double) StreamSupport
    .stream(input.spliterator(), false)
    .map(Packet::getLen_udp)
    .reduce(0, Integer::sum)*8/1024);
```

Siccome le features comuni e le features statiche speciali vengono calcolate allo stesso modo per la lunghezza del pacchetto UPD, per l'inter_timestamp, per l'inter_timestamp RTP e per l'inter della lunghezza del pacchetto UDP vengono indicate soltanto le righe di codice riguardanti la lunghezza del pacchetto UDP:

- Massimo:

```
feature.setLen_udp_max((int) StreamSupport
    .stream(input.spliterator(), false)
    .map(Packet::getLen_udp).max(Integer::compare).get();
```

- Minimo:

```
feature.setLen_udp_min((int) StreamSupport
    .stream(input.spliterator(), false)
  .map(Packet::getLen_udp)
  .min(Integer::compare).get());
```

- Media:

```
feature.setLen_udp_mean((double) StreamSupport
    .stream(input.spliterator(), false)
  .map(Packet::getLen_udp)
  .reduce(0, Integer::sum) / (count));
```

- Max_min_diff:

```
feature.setLen_udp_max_min_diff(
   feature.getLen_udp_max() -
      feature.getLen_udp_min());
```

- Max_min_R e min_max_R:

```
if((feature.getLen_udp_max()==0) &&
    (feature.getLen_udp_min()==0)){
     feature.setLen_udp_max_min_R(0.0);
     feature.setLen_udp_min_max_R(0.0);
  }else {
     feature.setLen_udp_max_min_R(
         (double)((double)
         (Math.abs(feature.getLen_udp_max()))/
         (Math.abs(feature.getLen_udp_max())+
          Math.abs(feature.getLen_udp_min()))));

     feature.setLen_udp_min_max_R(
         (double)((double)
         (Math.abs(feature.getLen_udp_min()))/
         (Math.abs(feature.getLen_udp_max())+
          Math.abs(feature.getLen_udp_min()))));
  }
```

- Len_unique_percent:

```
feature.setLen_udp_len_unique_percent(((double) StreamSupport
    .stream(input.spliterator(), false)
    .map(Packet::getLen_udp).distinct()
    .count())/(count));
```

- Max_value_count_percent:

```
app = StreamSupport.stream(input.spliterator(), false)
     .map(Packet::getLen_udp)
     .collect(Collectors.groupingBy(e -> e,
               Collectors.counting())).values();
if(app.size() > 0)
     feature.setLen_udp_max_value_count_percent(
         (double)Collections.max(app)/(count));
```

- Deviazione standard:

```
feature.setLen_udp_std(Math.sqrt((StreamSupport
    .stream(input.spliterator(), false)
    .map(Packet::getLen_udp)
    .map(e -> (double)(
        (double)(e-feature.getLen_udp_mean()))*
        (double)(e-feature.getLen_udp_mean()))
    .mapToDouble(Double::doubleValue)
    .sum()/(count))));
```

- Momento centrale di ordine 3:

```
feature.setLen_udp_moment3(StreamSupport
    .stream(input.spliterator(), false)
    .map(Packet::getLen_udp)
    .map(e -> (double)(
        (double)(e-feature.getLen_udp_mean()))*
        (double)(e-feature.getLen_udp_mean())*
        (double)(e-feature.getLen_udp_mean()))
    .mapToDouble(Double::doubleValue).sum()/(count));
```

- Momento centrale di ordine 4:

```
feature.setLen_udp_moment4(StreamSupport
    .stream(input.spliterator(), false)
    .map(Packet::getLen_udp)
    .map(e -> (double)(
        (double)(e-feature.getLen_udp_mean()))*
        (double)(e-feature.getLen_udp_mean())*
        (double)(e-feature.getLen_udp_mean())*
        (double)(e-feature.getLen_udp_mean()))
    .mapToDouble(Double::doubleValue)
    .sum()/(count));
```

- Indice di simmetria:

```
if(feature.getLen_udp_moment3() != 0.0)
    feature.setLen_udp_skew((double)
        feature.getLen_udp_moment3()/
        Math.sqrt(Math.pow(Math.pow(feature.getLen_udp_std(), 2), 3)));
```

- Curtosi:

```
if(feature.getLen_udp_moment4() != 0.0)
    feature.setLen_udp_kurtosis(((double)
        feature.getLen_udp_moment4()/
        Math.pow(Math.pow(feature.getLen_udp_std(), 2), 2))-3);
```

- Percentile25:

```
app2 = StreamSupport.stream(input.spliterator(), false)
    .map(Packet::getLen_udp).sorted().collect(Collectors.toList());

double d = ((double)(count-1)*25/100);
    if(d % 1 == 0) {
      feature.setLen_udp_p25(app2.get((int) d));
    }else {
        int min = (int) d;
        int max = min + 1;
        double result = (double) ((app2.get(max) -
            app2.get(min))*(d-(int)d)) + app2.get(min);
     feature.setLen_udp_p25(result);
  }
```

Si può notare che il codice è stato sviluppato utilizzando la Streaming API di Java 8, questo per garantire una maggiore leggibilità. Dopo aver calcolato per ogni flusso di pacchetti le diverse features, viene effettuata una KeyBy identica alla precedente per avere la certezza di selezionare quelleo principali separatamente per ogni flusso.

4.3.5 Map function with CalculateJsonSmall

Con la process function precedente sono state calcolate le features dalle quali adesso bisogna selezionare quelle principali da fornire come input al classificatore. Per effettuare questa operazione è stata implementata una classe chiamata CalculateJsonSmall che al proprio interno contiene una map function. Per ogni flusso di pacchetti è stato creato un oggetto feature contenente le diverse features calcolate e ora questo oggetto viene passato a questa map function che andrà ad estrarre quelle principali. Secondo un processo di ingegnerizzazione e selezione del team Smart Data del Politecnico di Torino, dalle 130 features di partenza ne sono state selezionate le quattro che consentono di fare F1 score maggiore del 90%. Le quattro features sono le seguenti:

- la lunghezza media dei pacchetti UDP

- la len_unique_percent della lunghezza dei pacchetti
- il percentile25 della lughezza del pacchetto udp
- la len_unique_percent dell'inter del timespamp RTP

Dopo aver estratto le seguenti features, bisogna fornirle in input al classificatore per ottenere diversi il tipo di sorgente multimediale. Per fare questo viene creato un oggetto JSON, vengono inserite al proprio interno le features estratte e viene fornito come input al classificatore. La classe CalculateJsonSmall, con al proprio interno una map function che consente di effettuare le operazioni elencante in precedenza, è stata realizzata nel seguente modo:

```
public static class CalculateJsonSmall implements
    MapFunction<Feature, JSONObject> {

    @Override
    public JSONObject map(Feature feature) {

        JSONObject obj = new JSONObject();

        obj.put("len_udp_mean",
            feature.getLen_udp_mean());
        obj.put("len_udp_len_unique_percent",
            feature.getLen_udp_len_unique_percent());
        obj.put("len_udp_p25",
            feature.getLen_udp_p25());
        obj.put("rtp_interarrival_len_unique_percent",
            feature.getRtp_interarrival_len_unique_percent());

        obj.put("ssrc", feature.getSsrc());
        obj.put("ip_src", feature.getIp_src());
        obj.put("ip_dst", feature.getIp_dst());
        obj.put("prt_src", feature.getPrt_src());
        obj.put("prt_dst", feature.getPrt_dst());
        obj.put("p_type", feature.getP_type());

         return obj;
    }
}
```

Un esempio di oggetto JSON, contenente le features scelte, da fornire come input al classificatore è il seguente:

```
{
     "ssrc":"0x54912529",
     "ip_src":"192.168.1.105",
     "ip_dst":"69.26.161.221",
     "prt_src":64693,
     "prt_dst":5004,
```

```
    "p_type":127,
    "len_udp_mean":226,
    "len_udp_p25":226,
    "len_udp_len_unique_percent":0.3333333333333333,
    "rtp_interarrival_len_unique_percent":0.5
}
```

Nell'oggetto JSON oltre ad aver inserito le features scelte, sono stati inseriti anche i campi del pacchetto che compongono la chiave per consentire di distinguere le features per ogni tipo di flusso.

4.3.6 Map function with POST_Classificatore

Avendo a disposizione per ogni flusso un oggetto JSON contente i campi scelti, non resta che fornirlo al classificatore per ottenere il tipo di sorgente multimediale. Il classificatore machine learning, a cui è stato già insegnato che a determinati valori corrisponde un determinato tipo sorgente multimediale, non è stato realizzato ma è stato fornito dal team Smart Data del Politecnico di Torino. Questo classificatore è stato realizzato utilizzando il decision tree e nella figura 4.2 viene riportato il suo plot. Il classificatore è stato messo su un Web Server e per fornirgli l'oggetto JSON con le features e i campi che compongono la chiave, viene fatta una richiesta HTTP POST. In Codice 6.3 viene mostrato come è stato realizzato il server a cui fare la POST. Per poter utilizzare il seguente server è stato fornito anche un file .pickle e nella variabile LINK_PICKLE è stato messo il percorso dove è stato salvato il file .pickle. Per eseguire il codice da terminale viene utilizzato il seguente comando: python Server_classifier.py. La richiesta POST va fatta a IP:PORTA/forecast e il body della POST deve essere un JSON contenente le quattro features scelte. Per effettuare una richiesta HTTP POST al server ed ottenere il tipo di sorgente multimediale è stata realizzata una classe POST_Classificatore contenente una map function che effettua questa richiesta per ogni flusso di pacchetti. Il codice con cui è stata realizzata la classe è il seguente:

```
public static class POST_Classificatore implements
    MapFunction<JSONObject, JSONObject> {

    @Override
    public JSONObject map(JSONObject obj) throws IOException {

       JSONObject json = new JSONObject();
       json.put("len_udp_mean", obj.get("len_udp_mean"));
       json.put("len_udp_len_unique_percent",
       obj.get("len_udp_len_unique_percent"));
       json.put("len_udp_p25", obj.get("len_udp_p25"));
       json.put("rtp_interarrival_len_unique_percent",
       obj.get("rtp_interarrival_len_unique_percent"));
```

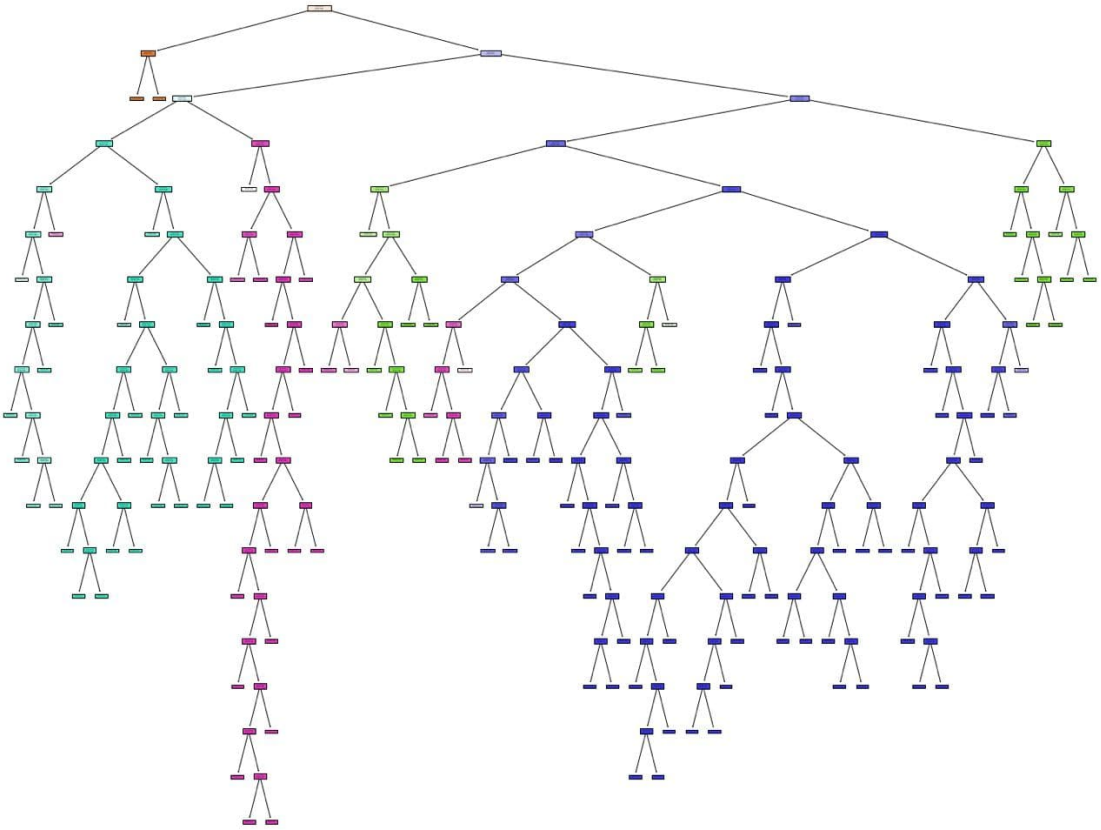

Figura 4.2. Plot of machine learning classifier

```
URL url = new URL("http://127.0.0.1:9093/forecast");

HttpURLConnection conn =
    (HttpURLConnection) url.openConnection();
conn.setDoOutput( true );
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-Type","application/json; utf-8");
conn.setRequestProperty("Accept","application/json");

try(OutputStream os = conn.getOutputStream()) {
  byte[] input=json.toString().getBytes("utf-8");
    os.write(input, 0, input.length);
}

StringBuilder response = new StringBuilder();
try(BufferedReader br = new BufferedReader(
  new InputStreamReader(conn.getInputStream(),"utf-8"))) {
```

```
        String responseLine = null;
        while ((responseLine = br.readLine())!=null) {
             response.append(responseLine.trim());
        }
       }

      JSONObject result = new JSONObject(response.toString());
      result.put("ssrc", obj.get("ssrc"));
      result.put("ip_src", obj.get("ip_src"));
      result.put("ip_dst", obj.get("ip_dst"));
      result.put("prt_src", obj.get("prt_src"));
      result.put("prt_dst", obj.get("prt_dst"));
      result.put("p_type", obj.get("p_type"));

      return result;
   }
}
```

Siccome la map function riceve un oggetto JSON contente oltre alle features scelte anche i campi che compongono la chiave viene creato un nuovo oggetto JSON, vengono inserite al proprio interno soltanto le features estratte dal JSON ricevuto e viene effettuata la richiesta HTTP POST con il JSON appena creato. Non si può effettuare la richiesta utilizzando l'oggetto JSON iniziale perchè, come detto precedentemente, la richiesta deve essere effettuata in un determinato modo. Il risultato ricevuto dal server classifier sarà un oggetto JSON contenente il tipo di sorgente multimediale a cui corrispondono le features dategli in input. Successivamente a questo oggetto vengono aggiunti anche i campi che compongono la chiave in modo da sapere quel tipo di sorgente multimediale a che flusso di pacchetti corrisponde e poter inserire successivamente nel database le informazioni più complete possibili. Un esempio di oggetto JSON contente la chiave del flusso di pacchetti e il tipo di sorgente multimediale corrispondente è il seguente:

```
{
    "ssrc":"0x54912529",
    "ip_src":"192.168.1.105",
    "ip_dst":"69.26.161.221",
    "prt_src":64693,
    "prt_dst":5004,
    "p_type":127,
    "streaming_type_id":3,
    "streaming_type":"screensharing"
}
```

4.3.7 Map function with Contact_Node_RED

Quando si conclude una fase di elaborazione tipicamente, mediante il FlinkKafkaProducer, viene pubblicato su Apache Kafka il risultato dell'elaborazione dove nel caso della streaming platform che si sta realizzando è un oggetto JSON per ogni flusso multimediale contenente la chiave composta del flusso e suo tipo di sorgente multimediale. Successivamente applicazioni esterne possono leggere dai topic Kafka il risultato che è stato pubblicato. In questa streaming platform non è scritto il codice che implementa il FlinkKafkaProducer ma per pubblicare il risultato dell'elaborazione viene effettuata una chiamata HTTP POST a Node Red che si occupa di memorizzare il file JSON ricevuto nel database non relazionale realizzato nel prossimo livello. Per effettuare questa operazione è stata realizzata una classe chiamata Contact_Node_RED che al proprio interno contiene una map function. Questa map function riceve come parametro un oggetto JSON, contenente la chiave primaria composta del flusso e il tipo di sorgente multimediale che corrisponde a quel flusso, e per ogni flusso effettua una chiamata HTTP POST verso Node Red per fornirgli l'oggetto JSON calcolato dalla fase di elaborazione. La classe Contact_Node_RED, contenente la map function che ci consente di raggiungere Node Red, è stata realizzata nel seguente modo:

```
public static class Contact_Node_RED implements
    MapFunction<JSONObject, JSONObject> {

    @Override
    public JSONObject map(JSONObject obj) throws IOException {

       obj.put("machine", "flinkDef");
       String output = "[";
       output = output + obj.toString() + "]";
       byte[] postDataBytes = output.getBytes("UTF-8");
       URL url = new URL("http://172.16.8.196:30880/flink/insert");
       HttpURLConnection conn = (HttpURLConnection)url.openConnection();
       conn.setRequestMethod("POST");
       conn.setRequestProperty("Content-Type","application/json");
       conn.setRequestProperty("Content-Length",
           String.valueOf(postDataBytes.length));
       conn.setDoOutput(true);
       conn.getOutputStream().write(postDataBytes);
       Reader in = new BufferedReader(
           new InputStreamReader(conn.getInputStream(),"UTF-8"));
       return obj;
    }
}
```

4.4 Realizzazione del Data Storage Layer

In questo livello vengono mantenuti dati ricevuti dal livello precedente ovvero gli oggetti JSON contenenti il tipo di sorgente multimediale per ogni flusso e la relativa chiave

composta. Per la realizzazione di questo livello oltre ad utilizzare Apache CouchDB che consente di mantenere i dati che vengono ricevuti, è stato anche utilizzato il framework Node Red. Node Red consente di ricevere i dati inviati dal livello precedente mediante una chiamata HTTP POST. In Node Red sono stati definiti dei nodi che consentono di ricevere i dati dal livello precedente, estrarne il payload che corrisponde all'oggetto JSON risultante dall'elaborazione del livello precedente e inserire questo JSON all'interno di Apache CouchDB che rappresenta il cuore di questo livello. Essendo Apache CouchDB un database non relazionale, non si presenta nessun problema nell'andare ad inserire gli oggetti JSON che gli vengono forniti da Node Red mediante una HTTP POST. Nella figura 4.3 vengono mostrate alcune righe del database Apache CouchDB non relazionale utilizzato mentre nella figura 4.4 viene mostrata, in dettaglio, una singola riga del database.

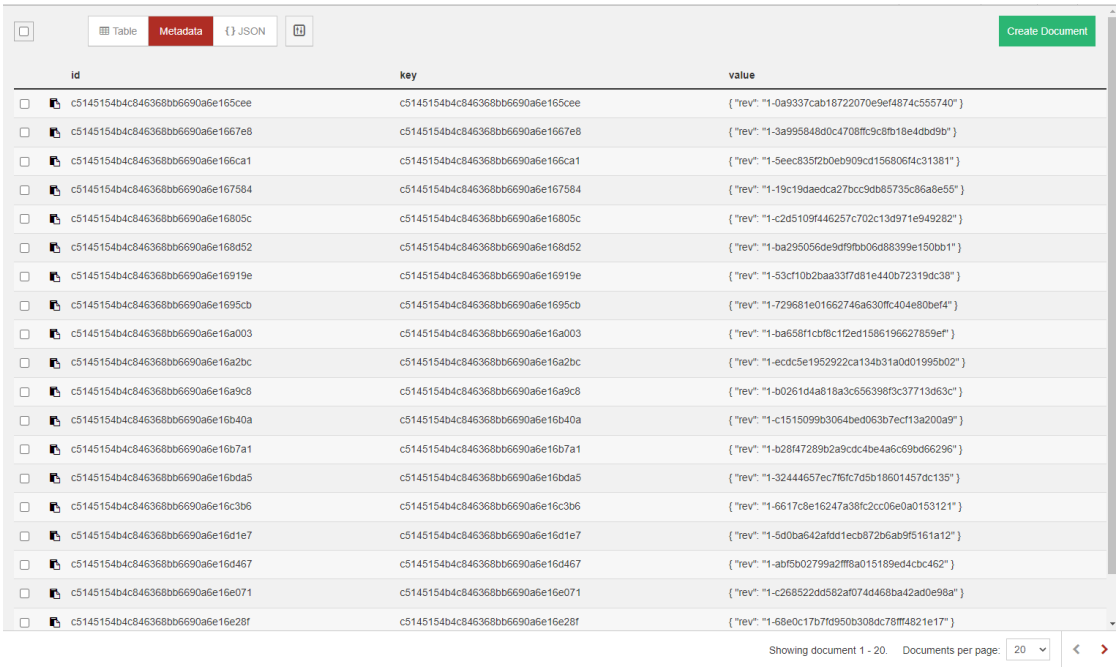

Figura 4.3. Rows in the database

Figura 4.4. File JSON in the database

4.5 Realizzazione del Data Query Layer

Per la realizzazione di questo livello è stato utilizzato un approccio a microservizi creandone uno e progettando al suo interno un'API che si occupa di interrogare il database e fornire i dati al front-end.

4.5.1 Progettazione dell'API

Questa streaming platform espone un'API che consente all'utente di interagire con il front-end ovvero la Web App realizzata nell'ultimo livello. Si tratta di un'API RESTful che utilizza HTTP ed è stata progettata in base a risorse che nel caso della streaming platform realizzata rappresentano i pacchetti di rete inseriti nel database non relazionale. Le risorse vengono identificate dall'URI che è il path con cui una risorsa viene acquisita e dove vengono effettuate delle operazioni su di essa. L'utente mediante il front-end interagisce con il servizio fornito dall'API scambiandosi rappresentazioni di pacchetti nel formato JSON. Non esiste come formato soltanto il JSON è stato scelto di utilizzarlo siccome i pacchetti sono inseriti con questo formato nel database. L'API della streaming platform è basata sull'HTTP e questo protocollo offre vari metodi per realizzare il principio CRUD. Le funzionalità fornite sono due ed entrambe sono state mappate con il medoto HTTP GET.

Risorsa	Metodo	Tipo	Descrizione
/sniffer/getPackets	GET	JSON	Ritorna tutti i pacchetti presenti nel database
/sniffer/getLastPackets	GET	JSON	Ritorna il pacchetto più vecchio per ogni ip sorgente presente nel database

La prima GET ritorna una lista con tutti i pacchetti presenti all'interno del database mentre la seconda ritorna una lista di pacchetti contenete per ogni ip sorgente il pacchetto più vecchio in base al timestamp presente nel file JSON di ogni pacchetto. Se il metodo GET ha successo viene restituito il codice di stato HTTP 200 che indica successo con la lista di JSON mentre 404 in caso di errore ad indicare che la risorsa non è stata trovata con un JSON contenente un messaggio di errore. Per semplificare la realizzazione dell'API, è stato utilizzato Swagger[20] che offre strumenti semplici da utilizzare che rendono questa fase più semplice. Swagger ha permesso di descrivere la struttura dell'API in modo semplice e andandola a leggere è stata creata una documentazione semplice e interattiva. Per la realizzazione grafica dell'API Swagger ha offerto un editor per scrivere codice del YAML mentre un pannello di visualizzazione ha consentito di vedere come appariva e come si comportava l'API appena realizzata. Nella figura 4.5 viene mostrato il pannello di visualizzazione dell'API realizzata. Dal pannello di visualizzazione si può notare la presenza di /sniffer/getPackets e /sniffer/getLastPackets che sono le due GET di cui si è parlato precedentemente. Esse, nello Swagger, prendono il nome di interfacce di ricerca e consentono di accedere al set di dati presenti nel database non relazionale. Nella figura 4.6, invece, viene mostrato il file JSON ritornato il caso di errore o di successo con il relativo codice.

4.5.2 Implementazione dell'API

Nella progettazione dell'API RESTful di cui si è appena parlato, sono state definite risorse e relazioni. Adesso, nella fase di implementazione, verrà visto come è stata implementata e resa disponibile questa API al livello successivo della streaming platform. Il front-end deve poter chiamare l'API per interfacciarsi con il database però, come detto precedentemente, sia l'API che la chiamata al database devono essere nascoste al front-end. Sono state realizzate due funzioni chiamate getPacketsElements() e getLastPacketsElements() che fungono da wrapper alle funzioni che effettueranno una richiesta HTTP GET per ottenere le risorse richieste. Le due funzioni wrapper vengono chiamate nel seguente modo:

```
let api = this.snifferStatusService.getPacketsElements();
let api_last = this.snifferStatusService.getLastPacketsElements();
```

Figura 4.5. Streaming platform API

All'interno di api ci saranno tutte le risorse ritornate dal database ovvero la lista totale di pacchetti mentre in api_last la lista di pacchetti più vecchi per ip sorgente. La getPacketsElements() è implementata nel seguente modo:

```
private API_PATH = 'http://localhost:3009/api/sniffer/';
constructor(private http: HttpClient) { }

getPacketsElements(): Observable<any[]>{
    return this.http.get(this.API_PATH + 'getPackets')
        .pipe(map((ritorno : any) => {
              return ritorno;
            }), catchError(this.handleError));
}
```

mentre la getLastPacketsElements():

```
getLastPacketsElements(): Observable<any[]>{
    return this.http.get(this.API_PATH + 'getLastPackets')
        .pipe(map((ritorno : any) => {
```

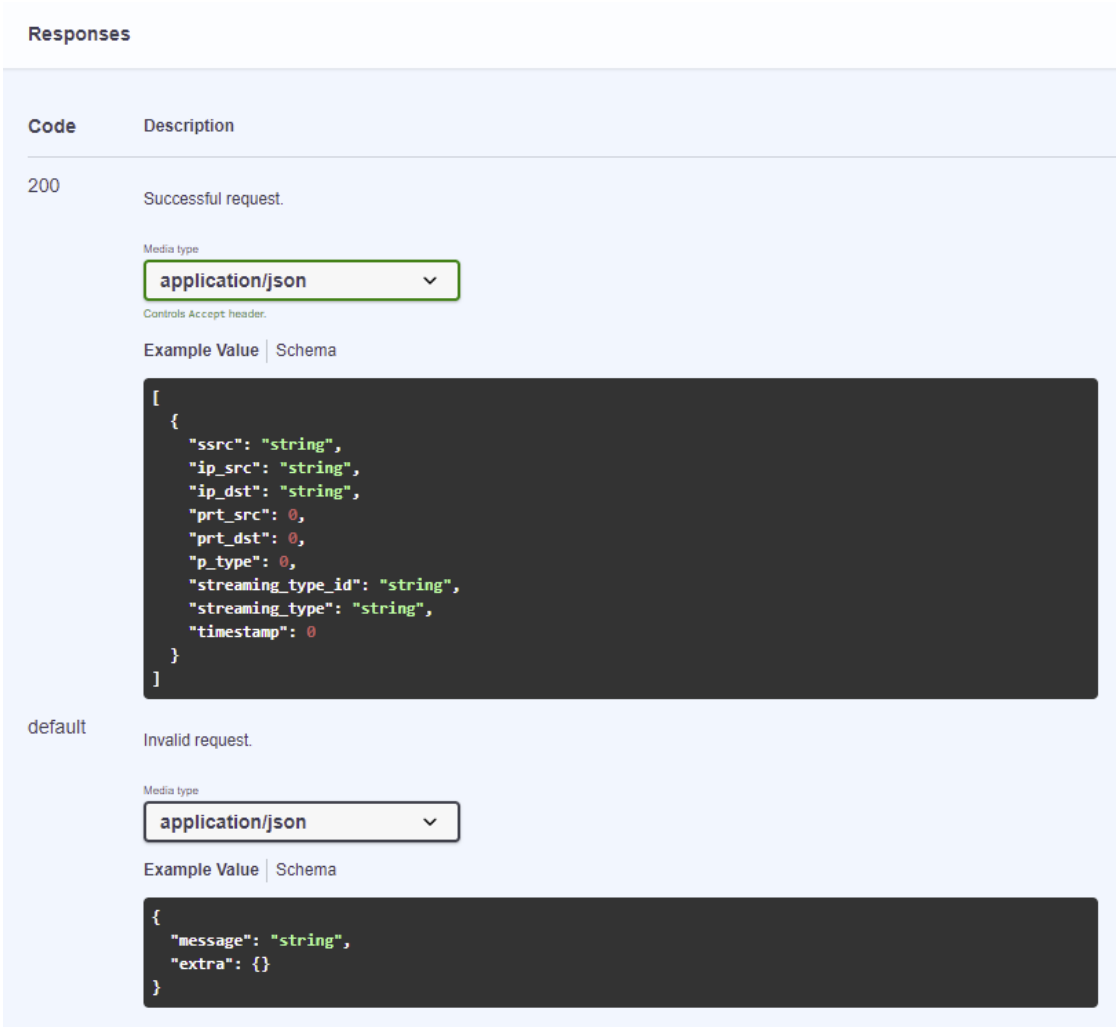

Figura 4.6. getPackets/getLastPackets responses

```
            return ritorno;
          }), catchError(this.handleError));
}
```

In API_PATH si può notare la presenza di 3009 che è la porta in locale su cui è in esecuzione il microservizio. Mediante queste chiamate HTTP GET il front-end andrà a richiedere le risorse presenti nel database e queste chiamate vengono intercettate dal controller che per ritornare le risorse al front-end affiderà il compito di interrogare il database al service. Il controller amministra le richieste HTTP in entrata e fornisce i dati ricevuti dalle richieste HTTP al service che si occupa di eseguire il lavoro ovvero interrogare correttamente il database. Quindi, il controller spedisce i dati della richiesta HTTP al service e coordina queste chiamate al service. Il service, invece, esegue il proprio

compito ovvero interrogare il database. Una volta portato a termine il proprio compito, il service ritorna il risultato al controller. Il service possiede la logica utile a garantire i requisiti e ritornare ciò che il consumatore dell'API ha richiesto. Il controller è stato implementato nel seguente modo:

```
export class SnifferController {
  public static async getPackets(): Promise<PacketResponse[]> {
    const snifferService = new SnifferService();
    return await snifferService.getPackets(req);
  }
  public static async getLastPackets():
        Promise<PacketResponse[]> {
    const snifferService = new SnifferService();
    return await snifferService.getLastPackets();
  }
}
```

PacketResponse è un' interfaccia che contiene i campi presenti in ogni riga del database ed è stata rappresentata nel seguente modo:

```
export interface PacketResponse {
    ssrc: string;
    ip_src: string;
    ip_dst: string;
    prt_src: number;
    prt_dst: number;
    p_type: number;
    streaming_type_id: string;
    streaming_type: string;
    timestamp: number;
}
```

Il codice con cui è stato implementato il service viene indicato in Codice 6.4 Nel caso di getLastPackets() si può notare che sono stati ritornati tutti i pacchetti memorizzati all'interno del database e poi è stata realizzata una funzione chiamata Filter che si occupa di filtrare tutti i pacchetti ritornati mantenendo quelli più vecchi in base al timestamp per ogni ip sorgente.

4.6 Realizzazione del Data Visualization Layer

Questo livello viene realizzato utilizzando lo strumento open-source Angular che, mediante l'utilizzo di opportuni componenti grafici, consente di visualizzare le informazioni ritornate dall'interrogazione al database. Come detto precedentemente, in questo livello non viene interrogato il database ma viene utilizzata l'API implementata nel livello precedente. Qui ci si limita a chiamare nel component le funzioni getPacketsElements() e getLastPacketsElements() che fungono da wrapper alle funzioni che effettueranno una richiesta HTTP GET per ottenere le risorse presenti nel database. Nella figura 4.7 viene visualizzata la prima metà della dashboard realizzata che, mediante l'utilizzo di una mat-table, consente di visualizzare tutta la lista di pacchetti ritornata dalla getPacketsElements(). Questa funzione deve essere chiamata appena viene creato il componente quindi

all'interno dell'ngOnInit. Per una migliore leggibilità e avere del codice ottimizzato, la funzione getPacketsElements() non viene chiamata direttamente all'interno dell'ngOnInit ma viene realizzata una funzione chiamata listPackets() che chiama la getPacketsElements() e, a sua volta, la listPackets() viene chiamata nell'ngOnInit. La chiamata alla

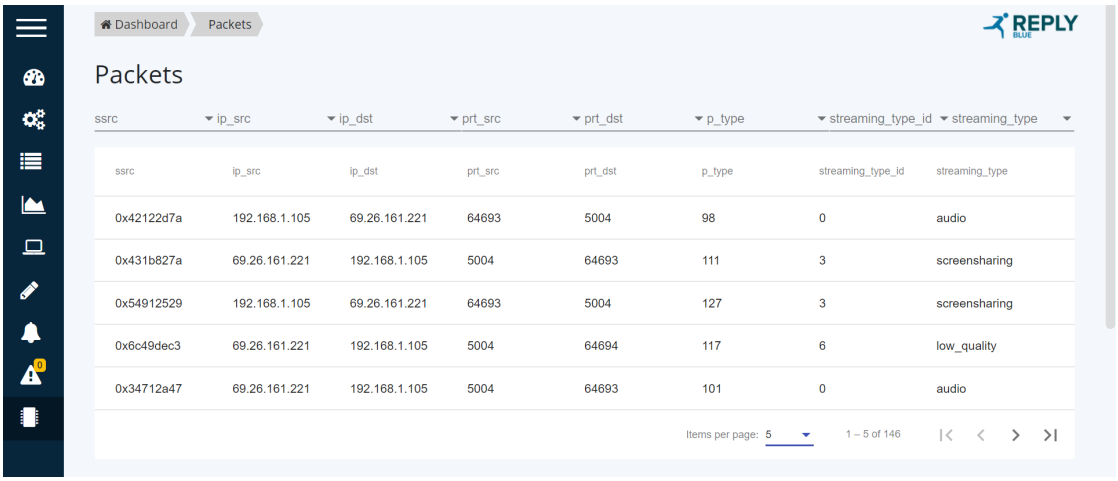

Figura 4.7. All packets in a mat-table

getPacketsElements(), nella listPackets(), viene effettuata nel seguente modo:

```
let api = this.snifferStatusService.getPacketsElements();
```

L'origine dei dati della mat-table è l'oggetto dataSource di tipo MatTableDataSource quindi le informazioni ritornate dalla getPacketsElements() vengono copiate all'interno di dataSource.data per consentire alla mat-table di poter visualizzare queste informazioni. La mat-table, indicata all'interno dell'html, viene definita colonna per colonna. Per non riportare tutto il codice html che consente di realizzare la mat-table completa, viene riportata soltanto la colonna dell'ssrc ma l'ng-container viene realizzato per ogni campo del pacchetto, nel codice html, per la realizzazione completa di questo livello.

```
<mat-table [dataSource]="dataSource" matSort>
    <ng-container matColumnDef="ssrc">
        <mat-header-cell *matHeaderCellDef mat-sort-header>
            ssrc
        </mat-header-cell>
        <mat-cell *matCellDef="let element">
            {{element.ssrc}}
        </mat-cell>
    </ng-container>
    ......
    <mat-header-row *matHeaderRowDef="displayedColumns">
    </mat-header-row>
    <mat-row *matRowDef="let row; columns:displayedColumns">
    </mat-row>
```

```
</mat-table>
```

Nella definizione della mat-table si può notare la presenza di dataSource che, come detto precedentemente, rappresenta l'origine dei dati. Le informazioni all'interno del database possono essere tante e per garantire all'utente che utilizza la Web App una maggiore comprensione delle informazioni visualizzate è stata aggiunta la possibilità di poter utilizzare dei filtri. Sopra la mat-table è stato aggiunto un menù a tendina che consente di filtrare tutti i pacchetti visualizzati in base al campo o ai campi che vengono selezionati. Nella figura 4.8 viene mostrato il menù a tendina dell'ssrc che consente di filtrare i pacchetti in base all'ssrc selezionato mentre nella figura 4.9 vengono visualizzate le informazioni dei pacchetti filtrate per ssrc, ip destinazione, porta destinazione e payload type. Il filtro

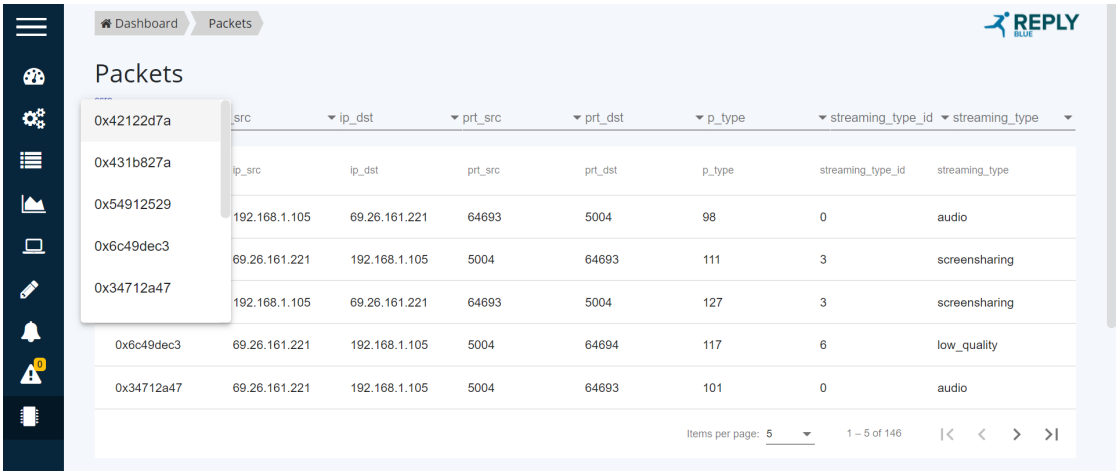

Figura 4.8. Filter packets for ssrc

per i pacchetti, in base alle informazioni selezionate nei vari menù a tendina, non viene applicato prima di chiamare la getPacketsElements(), quindi prima che venga effettuata la query al database, ma successivamente. Quando viene selezionata un'informazione da un menù a tendina, viene chiamata una funzione relativa alla colonna selezionata per il filtro e vengono filtrare le informazioni presenti nel dataSource.data che visualizza la mat-table. Una volta filtrate queste informazioni la mat-table, in tempo reale, visualizza le informazioni aggiornate. Per ogni campo del pacchetto, quindi per ogni colonna della mat-table, è stata realizzata una funzione di filtraggio ma per semplicità viene mostrata solo l'implementazione di quella dell'ssrc. Quando viene selezionato un ssrc dal menù a tendina, viene chiamata la seguente funzione che filtra le informazioni presenti nel dataSource.data in base all'ssrc selezionato:

```
onChangeSelectedSsrc(ssrc: string) {
    this.dataSource.data = this.listPacket.filter(function (el){
      return el.ssrc == ssrc
    });
}
```

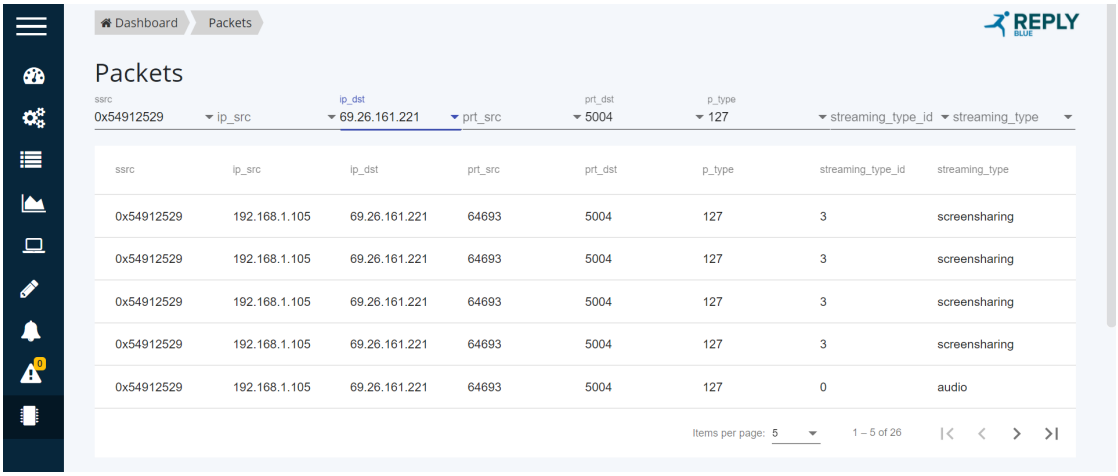

Figura 4.9. Filter packets for ssrc, ip destination, port destination and payload type

Nella prima metà della dashboard sono state visualizzate le informazioni dei vari pacchetti memorizzati nel database sfruttando una mat-table per la visualizzazione grafica e l'API sviluppata nel livello precedente, chiamata indirettamente dalla funzione wrapper getPacketsElements(), per ottenere le informazioni da visualizzare. Nella figura 4.10 viene visualizzata la seconda metà della dashboard. Sulla sinistra viene visualizzato l'ip sorgente e la data (ovvero il timestamp) dei pacchetti più vecchi presenti nel database per ip sorgente. La visualizzazione dei last packets è molto simile a quella di tutti i pacchetti

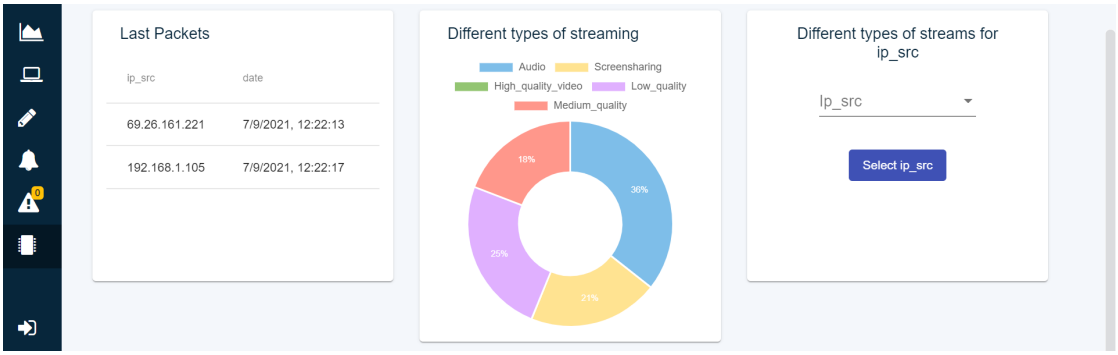

Figura 4.10. Last packets, percentages of all packets and percentages of filtered packets

avvenuta nella prima metà della dashboard. Viene utilizzata, anche questa volta, una mat-table con cui vengono visualizzati l'ip sorgente e la data. Per la visualizzazione di queste informazioni nella sorgente dei dati di questa mat-table vengono copiate le informazioni ritornate dalla getLastPacketsElements() che, come già detto precedentemente, funge da wrapper alla funzione che effettua l'HTTP GET per ottenere le risorse presenti nel database. La getLastPacketsElements verrà chiamata all'interno di una funzione

chiamata listLastPackets() che a sua volta viene chiamata all'interno dell'ngOnInit. La getLastPacketsElements() viene chiamata nel seguente modo:

```
let api_last = this.snifferStatusService.getLastPacketsElements();
```

Il valore di api_last verrà copiato all'interno dell'oggetto dataSourceLast di tipo MatTableDataSource che rappresenta l'origine dei dati di questa mat-table. Il codice html che consente di realizzare questa mat-table per visualizzare ip sorgente e data dei pacchetti più vecchi per ip sorgente è il seguente:

```
<mat-table [dataSource]="dataSourceLast">
    <ng-container matColumnDef="ip_src">
        <mat-header-cell *matHeaderCellDef>
            ip_src
        </mat-header-cell>
        <mat-cell *matCellDef="let element">
            {{element.ip_src}}
        </mat-cell>
    </ng-container>
    <ng-container matColumnDef="timestamp">
        <mat-header-cell *matHeaderCellDef>
            date
        </mat-header-cell>
        <mat-cell *matCellDef="let element">
            {{element.timestamp}}
        </mat-cell>
    </ng-container>
        <mat-header-row *matHeaderRowDef=
            "displayedColumnsLast">
        </mat-header-row>
        <mat-row *matRowDef="let row;
            columns: displayedColumnsLast">
        </mat-row>
</mat-table>
```

La parte centrale della seconda metà della dashboard contiene, invece, un grafico a torta che visualizza le percentuali dei diversi tipi di sorgenti multimediali rilevati all'interno della rete. L'origine dei dati di questo grafico sarà un vettore contenente in ogni posizione il conteggio del numero di pacchetti di un determinato tipo di sorgente. Questo grafico visualizza i valori presenti all'interno del vettore ma per ottenere anche il valore percentuale viene definita una option. Il primo elemento del vettore è stato fatto corrispondere al numero di pacchetti che trasportano come tipo di sorgente l'audio, il secondo lo screensharing, il terzo l'high quality video, il quarto il low quality e il quinto il medium quality. I conteggi dei differenti tipi di sorgenti verranno copiati all'interno di questo vettore dopo aver chiamato, all'interno della listPackets(), la getPacketsElements(). Il risultato di questa chiamata verrà filtrato facendo rimanere soltanto una lista contenente gli id corrispondenti ai diversi tipi di sorgenti. Successivamente mediante un ciclo viene letto ogni singolo valore di questa lista e il base al tipo di id viene incrementato il valore presente in una determinata posizione del vettore che sarà l'origine dei dati del grafico. Per filtrare il risultato della getPacketsElements() in modo da ottenere una lista degli id dei diversi tipi di sorgenti viene, utilizzato un .map realizzato nel seguente modo:

```
this.stypeList = result.map(ip => ip['streaming_type']);
```

Adesso, mediante un ciclo, in base al valore che contiene ogni singolo elemento della lista viene incrementato un elemento del vettore packetData che conterrà in ogni determinata posizione il numero di pacchetti che trasportava un determinato tipo di sorgente. Questo ciclo è stato implementato nel seguente modo:

```
for (let i of this.stypeidList) {
    if (i == 0) {
        this.packetData[0]++;
    } else if (i == 3) {
        this.packetData[1]++;
    } else if (i == 5) {
        this.packetData[2]++;
    } else if (i == 6) {
         this.packetData[3]++;
    } else if (i == 7) {
        this.packetData[4]++;
    }
}
```

Per aggiungere al grafico il valore delle percentuali, viene definita una option inizializzando il valore di render con percentage:

```
optionGrafico: ChartOptions = {
    responsive: true,
    plugins: {
      labels: {
        render: 'percentage',
        fontColor: 'white',
        fontSize: 10,
      },
    }
};
```

Il codice html che consente di visualizzare il grafico è il seguente:

```
<canvas baseChart id="graphic" #chartMaintenance="base-chart"
                height="100%" width="100%"
        [data]="packetData"
        [labels]="packetLabels"
        [colors]="chartColors"
        [chartType]="packetType"
        [options]="optionGrafico">
</canvas>
```

- data: indica la sorgente dei dati che in questo caso sarà il vettore packetData contenente il conteggio dei pacchetti per ogni tipo di sorgente multimediale. Il vettore packetData è stato definito nel components nel seguente modo:

  ```
  public packetData: number[] = [0, 0, 0, 0, 0];
  ```

Gli elementi di questo vettore vengono successivamente incrementati dal ciclo mostrato precedentemente.

- labels: indica il nome delle etichette presenti nel grafico a torta che in questo caso sarà il vettore packetLabels contenente il nome dei diversi di pi di sorgenti multimediali. Il vettore packetLabels è stato definito nel components nel seguente modo:

```
public packetLabels: string[] = [
    'Audio',
    'Screensharing',
    'High_quality_video',
    'Low_quality',
    'Medium_quality'
];
```

- colors: indica il codice del colore da assegnare ai vari tipi di sorgenti multimediali da mostrare nel grafico a torta che in questo caso sarà il vettore chartColors. Il vettore chartColors è stato definito nel components nel seguente modo:

```
public chartColors: Array<any> = [
{
    backgroundColor: ['#7ebee9',
                      '#ffe391',
                      '#93C572',
                      '#E0B0FF',
                      '#FF978B'],
}];
```

- chartType: indica il tipo di grafico da utilizzare che in questo caso viene indicato dalla stringa packetType. Questa stringa sta ad indicare di voler utilizzare il grafico a torta ed è stata definita nel components nel seguente modo:

```
public packetType: string = 'doughnut';
```

- options: sta ad indicare le opzioni da applicare al grafico utilizzato. L'opzione realizzata è quella mostrata precedentemente, ovvero:

```
optionGrafico: ChartOptions = {
    responsive: true,
    plugins: {
      labels: {
        render: 'percentage',
        fontColor: 'white',
        fontSize: 10,
      },
    }
};
```

Essa è stata chiamata optionGrafico ed è di tipo ChartOptions. Questa opzione consente di visualizzare sul grafico il valore percentuale corrispondente per ogni tipo di sorgente multimediale mediante un font di colore bianco e di dimensione 10.

Tutti i filtraggi che vengono apportati utilizzando i menù a tendina sulla mat-table della prima metà della dashboard, vengono riportati in tempo reale anche nella visualizzazione del grafico. Infine, nella parte in basso a destra della seconda metà della dashboard, viene consentito di applicare un filtro per ip sorgente al risultato della getPacketsElements() e di visualizzare le percentuali dei diversi tipi di sorgenti multimediali dei pacchetti restanti mediante un altro grafico a torta. Mediante l'utilizzo di un menù a tendina di checkbox quindi vengono selezionati uno o più ip sorgenti, vengono filtrati tutti i pacchetti che posseggono quegli ip sorgenti dalla lista di pacchetti ritornata della getPacketsElements() ovvero da tutti i pacchetti presenti nel database e le percentuali dei diversi tipi di sorgeti dei pacchetti restanti mediante un grafico. Nella figura 4.11, a destra, viene visualizzato il menù a tendina in cui viene selezionata la checkbox corrispondente all'ip sorgente 192.162.1.105. Nella figura 4.12, a destra, invece viene visualizzato il grafico con

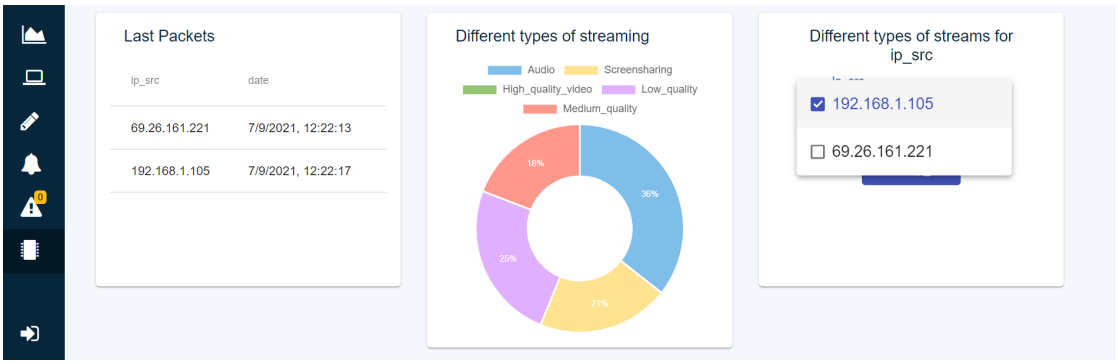

Figura 4.11. Select checkbox of ip source

il filtro per ip sorgente applicato. Se fossero stati selezionati tutti gli ip sorgenti del menù a tendina, entrambi i grafici a torta risulterebbero uguali. Anche l'origine dei dati di questo nuovo grafico sarà un vettore contenente in ogni posizione il conteggio del numero di pacchetti di un determinato tipo di sorgente però non bisogna considerare i pacchetti che hanno gli ip sorgenti selezionati nel menù a tendina. La logica applicata è la stessa del grafico precedente ma questa volta nel ciclo viene controllato che il pacchetto corrente non abbia l'ip sorgente pari a uno di quelli selezionati. Quando l'utente avrà effettuato la selezione dal menù a tendina, cliccherà sul bottone "select ip_src" e verrà chiamata la funzione "clickButton" che si occuperà, mediante un ciclo, di calcolare i valori da visualizzare nel grafico e li copierà nel vettore sorgente dei dati. Questa funzione clickButton è stata definita nel seguente modo:

```
clickButton() {
    this.packetDataFilter[0] = 0
    this.packetDataFilter[1] = 0
    this.packetDataFilter[2] = 0
    this.packetDataFilter[3] = 0
    this.packetDataFilter[4] = 0
    this.button = 0;
```

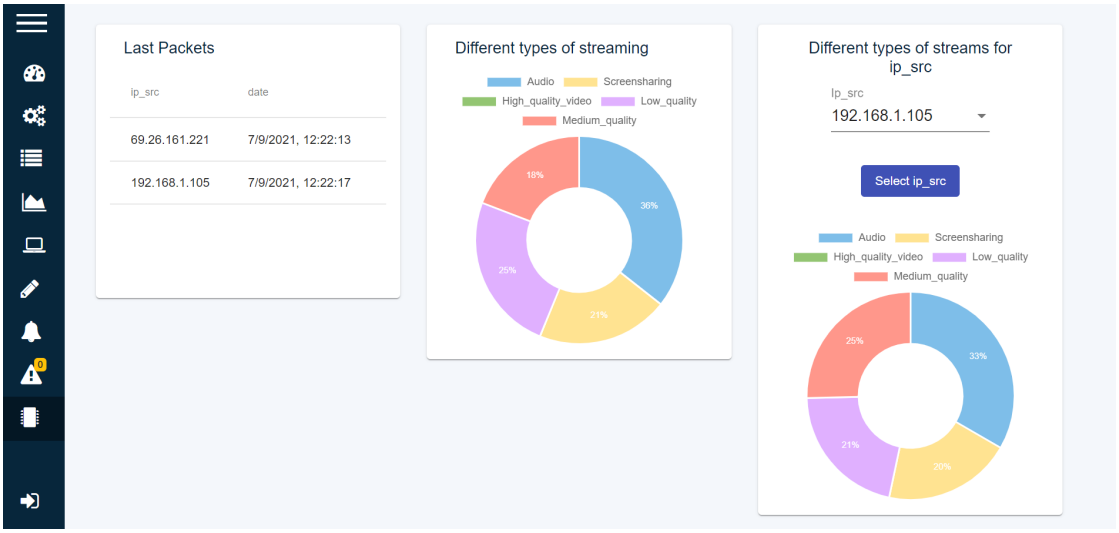

Figura 4.12. baseChart filtered for ip source selected

```
    for (var val = 0; val < this.ipSelected.length; val++)
    {
      for (var j = 0; j < this.listPacket.length; j++) {

        if(this.listPacket[j]['ip_src'] == this.ipSelected[val])
        {
          if (this.listPacket[j]['streaming_type_id'] == 0) {
            this.packetDataFilter[0]++;
          } else if (this.listPacket[j]['streaming_type_id'] == 3) {
            this.packetDataFilter[1]++;
          } else if (this.listPacket[j]['streaming_type_id'] == 5) {
            this.packetDataFilter[2]++;
          } else if (this.listPacket[j]['streaming_type_id'] == 6) {
            this.packetDataFilter[3]++;
          } else if (this.listPacket[j]['streaming_type_id'] == 7) {
            this.packetDataFilter[4]++;
          }
        }
      }
    }
    this.button = 1;
    this.chartMaintenance.chart.update();
}
```

Il vettore packetDataFilter è l'origine dei dati, il vettore listPacket contiene tutti i pacchetti ritornati mediante la getPacketsElements() mentre il vettore ipSelected contiene gli ip sorgenti selezionati dal menù a tendina. Il vettore ipSelected è stato inizializzato con gli ip sorgenti selezionati sfruttando ngModel. Inoltre è presente anche una variabile button che è un flag e in base al valore che possiede, nell'html indica se visualizzare oppure

no il grafico. Questo grafico è possibile visualizzarlo quando il vettore sorgente packetDataFilter è stato inizializzato con il conteggio dei pacchetti dei diversi tipi di sorgenti multimediali. Il codice html che consente di visualizzare il menù a tendina o il grafico, in base al valore che assume il flag button, è il seguente:

```
<mat-card-content class="my-card-content"
                    style="text-align: center;
                           font-size: 18px!important">
    <mat-form-field>
        <mat-select [(ngModel)]="ipSelected" placeholder="Ip_src"
            multiple>
            <mat-option *ngFor="let ip of filteredIpsrc" [value]="ip">
                {{ip}}
            </mat-option>
        </mat-select>
    </mat-form-field>

    <button mat-flat-button color="primary" style="margin: 15px"
                           (click)="clickButton()">
                Select ip_src
    </button>
</mat-card-content>

<mat-card-content class="my-card-content" *ngIf="button == 1">
    <ng-container>
        <canvas baseChart id="graphic2" #chartMaintenance2="base-chart"
                  height="100%" width="100%"
                [data]="packetDataFilter"
                [labels]="packetLabels"
                [colors]="chartColors"
                [chartType]="packetType"
                [options]="optionGrafico">
        </canvas>
    </ng-container>
</mat-card-content>
```

Capitolo 5

Conclusioni

Il fenomeno dei Big Data è in continuo sviluppo ed è destinato a condizionare fortemente le decisioni finanziare ed economiche del mercato. Esaminare i dati streaming permette alle aziende di utilizzare vantaggiose strategie di bussiness per ottimizzare i processi di produzione e incrementare i profitti ottenuti. In questo progetto, nella fase di design sono stati indicati gli strumenti open-source utilizzati per realizzare una streaming platform capace di acquisire, elaborare, archiviare e visualizzare dati in tempo reale. Dopo aver indicato le caratteristiche dei principali strumenti utilizzati, si è passati all'implementazione della streaming platform che consente di analizzare traffico dati real-time che, mediante tecniche basate su DPI, seleziona solo i flussi RTP. Per ogni flusso vengono estratte delle features che serviranno successivamente come input al classificatore, che con alta precisione è in grado di identificare il tipo di sorgente multimediale in una delle seguenti classi: Audio, Low Quality (LQ) Video – 180p, Medium Quality (MQ) Video – 360p, High Quality (HQ) Video – 720p e Screen Sharing. I risultati ottenuti, durante l'esecuzione di questa piattaforma, hanno dimostrato come sia stato possibile identificare il tipo di sorgente multimediale dei flussi multimediali RTC trasportati da flussi RTP, provenienti da un file .pcap o da una cattura real-time in una rete, con un F1 score maggiore del 90%. Il lavoro svolto dalla streaming platform è solamente il primo step di una piattaforma progettata per collezionare informazioni dettagliate del traffico RTC. Nonostante si sia scelto di analizzare traffico RTC osservato in una rete, il design della streaming platform è pensato per essere facilmente adattabile anche ad altri tipi di scenari in cui siano sempre presenti dati inviati in real-time.

Capitolo 6

Codice

Codice 6.1. Lettura di un file .pcap

```
import json
import time
import subprocess
import pandas as pd
from io import StringIO

import numpy as np
from kafka import KafkaProducer

KAFKA_HOSTS = 'localhost:9092'
KAFKA_VERSION = (0, 10)

def publish():
    source_pcap = "3_p.pcapng"
    filtro = "rtp.version==2"
    used_port = [5004]
    port_add = []
    for port in used_port:
        port_add.append("-d udp.port==" + str(port) + ",rtp")

    command = f"""tshark -r {source_pcap} -Y {filtro} \
                    -T fields {" ".join(port\_add)}
                    -E separator=? \
                    -E header=y -e frame.time_epoch \
                    -e frame.number -e frame.len \
                    -e udp.srcport -e udp.dstport \
                    -e udp.length  -e rtp.p\_type -e \
                    rtp.ssrc -e rtp.timestamp -e rtp.seq \
                    -e rtp.marker -e rtp.csrc.item \
                    -e ip.src -e ipv6.src \
                    -e ip.dst -e ipv6.dst  \
                    --enable-heuristic rtp_stun"""

    o,e = subprocess.Popen(command, encoding='utf-8',
                           stdout=subprocess.PIPE,
```

```
                        stderr=subprocess.PIPE,
                        shell=True).communicate()
    df = pd.read_csv(StringIO(o), header=0, sep="?",
                low_memory=False)
    return df

def run():
    df = publish()

    df["ip.src"] = df["ip.src"].fillna(df["ipv6.src"])
    df["ip.dst"] = df["ip.dst"].fillna(df["ipv6.dst"])
    df.drop(["ipv6.src", "ipv6.dst"], axis=1, inplace=True)
    df['rtp.p_type'] = df['rtp.p_type'].
            apply(lambda x: str(x).split(",")[0])
    df["rtp.csrc.item"] = df["rtp.csrc.item"].
                                fillna("empty")
    df = df.astype({'frame.time_epoch': 'float64',
                    'frame.number': 'int32',
                    'frame.len': 'int32',
                    'udp.length': 'int32',
                    'rtp.p_type': 'int',
                    'rtp.timestamp': np.int64,
                    'udp.srcport': 'int32',
                    'dp.dstport': 'int32',
                    'rtp.marker': 'int32',
                    'rtp.seq': 'int32',
                    })
    df = df.rename(columns={
        'frame.time_epoch': 'timestamps',
        'frame.number': 'frame_num',
        'frame.len': 'len_frame',
        'ip.src': 'ip_src',
        'ip.dst': 'ip_dst',
        'udp.srcport': 'prt_src',
        'udp.dstport': 'prt_dst',
        'udp.length': 'len_udp',
        'rtp.p\_type': 'p_type',
        'rtp.ssrc': 'ssrc',
        'rtp.timestamp': 'rtp_timestamp',
        'rtp.seq': 'rtp_seq_num',
        'rtp.marker': 'rtp_marker',
        'rtp.csrc.item': 'rtp_csrc'
        })

    for index, row in df.iterrows():
        parsed = json.loads(row.to_json())
        producer = KafkaProducer(bootstrap_servers=KAFKA_HOSTS,
                        api_version=KAFKA_VERSION)
        producer.send('flink-topic',json.dumps(parsed).endoce('ascii'))

    return df
```

```
if __name__ == '__main__':
    df = run()
```

Codice 6.2. Cattura del traffico real-time in una rete

```
import subprocess
import pyshark
from kafka import KafkaProducer
import nest_asyncio
import tornado
import json

KAFKA_HOSTS = 'localhost:9092'
KAFKA_VERSION = (0, 10)

def RTP_layer(Packet):
    try:
        payload = bin(int(str(Packet.udp.payload).
            replace(":",""),16))
        version = int(payload[2:4], 2)
        padding = int(payload[4])
        extension = int(payload[5])
        CC = int(payload[6:10],2)
        marker = int(payload[10])
        payload_type = int(payload[11:18],2)
        seq_number = int(payload[18:34],2)
        timestamp = int(payload[34:66], 2)
        ssrc = hex(int(payload[66:98], 2))
        no_parsing = payload[98:]
        return version, padding, extension, CC, marker,
        payload_type, seq_number, timestamp, ssrc, no_parsing
    except:
        return [False]*10


def function():
    capture = pyshark.LiveCapture(interface='5')
    for packet in capture.sniff_continuously():
        try:
            if packet.transport_layer == "UDP":
                version, padding, extension, CC,
                marker, payload_type, seq_number,
                rtp_timestamp, ssrc,
                no_parsing = RTP_layer(packet)
                if version == 2:
                    json_kafka = {
                    'timestamps' : packet.frame_info.time,
                    'ip_src' : packet.ip.src,
                    'ip_dst' : packet.ip.dst,
                    'prt_src' : packet.udp.srcport,
                    'prt_dst' : packet.udp.dstport,
                    'len_udp' : packet.frame_info.len,
                    'rtp_timestamp' : rtp_timestamp,
                    'rtp_seq_num' : seq_number ,
```

```
                    'ssrc' : ssrc,
                    'p_type' : payload_type,
                    'rtp_marker' : marker}
                encode('ascii'))
                producer = KafkaProducer(bootstrap_servers=KAFKA_HOSTS,
                                         api_version=KAFKA_VERSION)
                producer.send('flink-output',
                    json.dumps(json_kafka).encode('ascii'))
        except Exception as e:
            pass

if __name__ == '__main__':
    function()
```

Codice 6.3. Web Server

```
import cherrypy
import json
import pickle
import pandas as pd
LINK_PICKLE = "results_jitsi_DT_1000_hpt_True.pickle"

class Classifier(object):
    mappa = {
                "0": "audio",
                "3" : "screensharing",
                "5": "high_quality_video",
                "6" : "low_quality",
                "7" : "medium_quality"
            }

    exposed = True
    def __init__(self):
        file = open(LINK_PICKLE, "rb")
        self.clf = pickle.load(file)

    def GET (self,*uri,**params):
        return "RTP classifier"

    def POST(self,*uri,**params):
        if (uri[0]=='forecast'):
            try:
                result = {}
                info_json=cherrypy.request.body.read()
                info_dict=json.loads(info_json)
                data = pd.DataFrame([info_dict.values()],
                    columns=info_dict.keys())
                print(data.columns, data.shape)
                prediction = self.clf["clf"].predict(data)
                print(prediction, type(prediction),
                    prediction[0])
                result["streaming_type"] =
                    self.mappa[str(prediction[0])]
                result["streaming_type_id"] =
```

```
                    str(prediction[0])
                print(result)
                json_object= json.dumps(result, indent= 4)
                print(json_object)
                return json_object
            except Exception as e:
                print("Errore", e)
        return cherrypy.HTTPError(status=400,
            message="Invalid request")

if __name__== '__main__' :
        conf = {
            '/': {
            'request.dispatch' :
                cherrypy.dispatch.MethodDispatcher()
            }
        }
        cherrypy.tree.mount(Classifier(),'/',conf)
        cherrypy.config.update({'server.socket_port' : 9093})
        cherrypy.engine.start()
        cherrypy.engine.block()
```

Codice 6.4. Service per interrogare il database

```
export class SnifferService {
    private static readonly TIMEOUT = 10000;

/**
    * @description Thi method returns a list of packets with or
      without ip_src/ip_dest query parameters
    * @returns{Nano.MangoResponse<PacketResponse>.docs}
    * @memberofSnifferService
*/

    public async getPackets():
            Promise<PacketResponse[]> {

        const defaultSelector = {
            "_id": { '$gte': null }
        };

        let mangoQuery: Nano.MangoQuery = {
            "selector": {
                "_id": {
                    '$gte': null
                }
            },
            "limit": ALL
        };

        try {
            let couchDb = SnifferService.getCouch();
            var alerts = await couchDb.find(mangoQuery);
        } catch (error) {
```

```
            console.error("Cannot find packets", error);
            throw new ErrorResponse(error.description,
                error.statusCode, error.error, error.reason);
        }

        return alerts.docs;
    }

/**
    * @description Thi method returns a list of packets, a packet
      for each ip_src with the old timestamp
    * @returns{Nano.MangoResponse<PacketResponse>.docs}
    * @memberofSnifferService
*/

    public async getLastPackets():
            Promise<PacketResponse[]> {

        const defaultSelector = {
            "_id": { '$gte': null }
        };

        let mangoQuery: Nano.MangoQuery = {
            "selector": {
                "_id": {
                    '$gte': null
                }
            },
            "limit": ALL
        };


        try {
            let couchDb = SnifferService.getCouch();
            var packets = await couchDb.find(mangoQuery);
        } catch (error) {
            console.error("Cannot find packets", error);
            throw new ErrorResponse(error.description,
                error.statusCode, error.error, error.reason);
        }

        let lastPacketsArray = SnifferService.Filter(packets);

        return lastPacketsArray;
    }

    private static Filter(packets: Nano.
            MangoResponse<PacketResponse>){
        let lastPacketsArray = [];

        for (let obj of packets.docs) {
            let elemento = lastPacketsArray
                find(item => item.ip_src === obj.ip_src);
```

```
            if (!elemento) {
                lastPacketsArray.push(obj);
            } else {
                if (obj.timestamp > elemento.timestamp) {
                    lastPacketsArray.splice(lastPacketsArray.
                        findIndex(oggetto => oggetto.ip_src ===
                            elemento!.ip_src),1);
                    lastPacketsArray.push(obj);
                }
            }
        }
        return lastPacketsArray;
    }


    private static getCouch():
            Nano.DocumentScope<PacketResponse> {
        let config: Nano.Configuration = {
            url: DB_CONFIG.url,
            requestDefaults: {
                timeout: this.TIMEOUT,
                auth: {
                    username: DB_CONFIG.auth.user,
                    password: DB_CONFIG.auth.pass
                }
            }
        };
        try {
            let nano = Nano(config);
            var db: Nano.DocumentScope<PacketResponse> =
                nano.db.use(DB_CONFIG.database);
        } catch (error) {
            console.error("Cannot_init_database" + DB_CONFIG.database);
            throw error;
        }
        return db;
    }
}
```

Bibliografia

[1] Angular. The modern web developer's platform. 2010-2021. URL https://angular.io/.

[2] Nasseh Tabrizi Babak Yadranjiaghdam, Seyedfaraz Yasrobi. Developing a real-time data analytics framework for twitter streaming data. 2017 IEEE 6th International Congress on Big Data.

[3] G. Combs. Tshark - the wireshark network analyzer 3.4.8. URL https://www.wireshark.org/docs/man-pages/tshark.html.

[4] Apache Software Foundation. Apache flink - stateful computations over data streams. 2014-2021. URL https://flink.apache.org/.

[5] Apache Software Foundation. What is apache flink? - applications. 2014-2021. URL https://flink.apache.org/flink-applications.html.

[6] Apache Software Foundation. What is apache flink? — architecture. 2014-2021. URL https://flink.apache.org/flink-architecture.html.

[7] Apache Software Foundation. What is apache flink? - operations. 2014-2021. URL https://flink.apache.org/flink-operations.html.

[8] Apache Software Foundation. Apache flink - operators. 2014-2021. URL https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/dev/datastream/operators/overview/.

[9] Apache Software Foundation. Apache kafka documentation. 2017. URL https://kafka.apache.org/documentation/.

[10] Apache Software Foundation. Apache couchdb 3.1.1 documentation. 2021. URL https://docs.couchdb.org/en/stable/.

[11] M. Trevisan P. Garza M. Meo M.M. Munafò G. Carofiglio G. Perna, D. Markudova. Online classification of rtc traffic. IEEE 18th Annual Consumer Communications Networking Conference Las Vegas 2021.

[12] Nishant Garg. *Learning Apache Kafka Second Edition*. 2015.

[13] Red Hat. Cos'è un'api? 2021. URL https://www.redhat.com/it/topics/api/what-are-application-programming-interfaces.

[14] Red Hat. Come funziona apache kafka? 2021. URL https://www.redhat.com/it/topics/integration/what-is-apache-kafka.

[15] Red Hat. Cosa sono i microservizi? 2021. URL https://www.redhat.com/it/topics/microservices/what-are-microservices.

[16] W. Wahlster J. Cavanillas, E. Curry. *New Horizons for a Data-Driven Economy*. T. Becker, 2016.

[17] I. Drago M.M. Munafò M. Mellia M. Trevisan, D. Giordano. Five years at the edge: Watching internet from the isp network. IEEE/ACM Trans. on Networking Vol:28 2020.

[18] Oracle. Cosa sono i big data? URL https://www.oracle.com/it/big-data/what-is-big-data/.

[19] Guy Gerson-Golan Guy Hadash Francois Carrez Paula Ta-Shma, Adnan Akbar and Klaus Moessner. *An Ingestion and Analytics Architecture for IoT Applied to Smart City Use Cases*. IEEE Internet of Things Journal Vol:5 2018.

[20] SmartBear Software. What is swagger? URL https://swagger.io/docs/specification/2-0/what-is-swagger/.