Sistema di controllo e monitoraggio

Progetto sistemi distribuiti e cloud computing

Cristian Milia
Ingegneria Informatica
Università di Roma Torvergata
Italia
miliacristian4@gmail.com

Valentino Perrone
Ingegneria Informatica
Università di Roma Torvergata
Italia
perrone.valentino@gmail.com

Pusceddu Anthony
Ingegneria Informatica
Università di Roma Torvergata
Italia
pusceddu.anthony@gmail.com

Indice

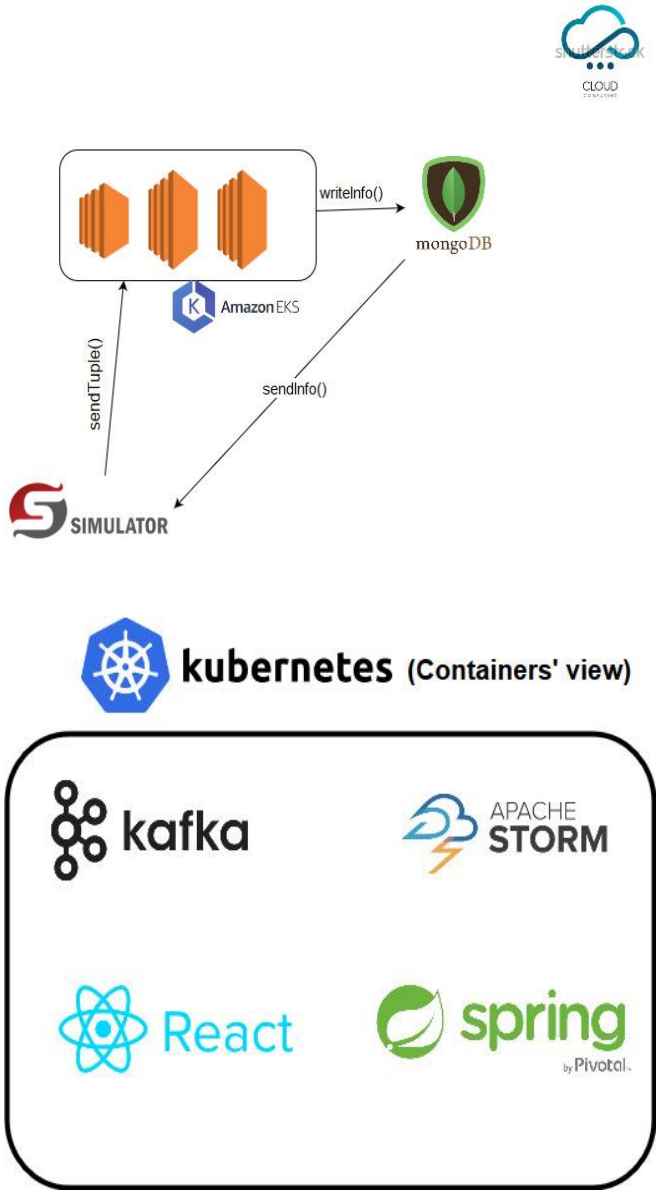
Introduzione

Il progetto realizzato è un sistema distribuito per il monitoraggio e controllo del traffico urbano, che fa uso di alcuni servizi Cloud forniti da AWS Amazon e Atlas. Da AWS viene utilizzato il servizio di gestione dei container EKS che utilizza principalmente due servizi Amazon: EC2 Medium e il Load Balancer mentre da Atlas viene utilizzato il database distribuito MongoDB.

Tale sistema distribuito permette di rilevare in tempo reale lo stato di congestione delle vie di una città (o di un suo quadrante) e di regolare la durata dei semafori posti alle intersezioni stradali.
Il sistema di monitoraggio offre informazioni in tempo reale sulla velocità e sul numero dei veicoli che attraversano l'intersezione e genera un allarme ogni qualvolta un semaforo ha un malfunzionamento, fornendo informazioni sulla posizione del semaforo e la tipologia della lampada guasta. Il sistema di monitoraggio permette di visualizzare il risultato di due query: la classifica delle 10 intersezioni con maggiore velocità media calcolate su diverse finestre temporali e le intersezioni aventi il valore della mediana del numero dei veicoli che hanno attraversato l'intersezione superiore al valore della mediana globale dei veicoli che hanno attraversato le intersezioni calcolate su diverse finestre temporali.
Il sistema di controllo gestisce la temporizzazione dei semafori in modo da ridurre il traffico stradale.
In particolare utilizza l'algoritmo di Webster per adattare la durata della luce verde del semaforo in base al numero (e alla velocità) dei veicoli in transito, al fine di garantire un livello adeguato di fluidità del traffico.
Viene infine fornita un'interfaccia grafica per visualizzare tutti i sensori, inserire, modificare le informazioni o eliminare un singolo sensore e per visualizzare le statistiche del sistema.
Per la realizzazione del sistema di monitoraggio e di controllo è stato utilizzato il framework ApacheStorm e ApacheKafka e il database MongoDB.

Framework e servizi utilizzati

Per la realizzazione del sistema distribuito sono stati utilizzati vari Framework e servizi.
Tra i principali troviamo: il Servizio EKS di AWS Amazon, il Framework ApacheStorm, il Framework ApacheKafka, il DB MongoDB, il Framework Spring per il Back-end e la libreria React per il Front-end.
Di seguito uno schema ad alto livello dell'architettura utilizzata.



Docker

Docker è un progetto open-source che automatizza il deployment di applicazioni all'interno di container software.

Nel progetto ogni applicazione è posta all'interno di un container. I vantaggi nell'utilizzare Docker, e quindi di utilizzare la virtualizzazione al livello di sistema operativo per il deploy di applicazioni, sono differenti:

- Possibilità di gestire il deploy dell'applicazione senza doversi preoccupare della configurazione dell'ambiente di runtime.
- Portabilità: i container Docker possono essere eseguiti su diverse piattaforme cloud (come istanze EC2 di Amazon o istanze Google compute engine). Quindi un container eseguito in un'istanza EC2 può essere portato in un qualsiasi altro ambiente garantendo consistenza e funzionalità.
- Isolamento: Docker assicura che le applicazioni e le risorse siano isolate.

Kubernetes

Kubernetes è un progetto open source realizzato da Google che automatizza il processo di distribuzione e gestione di applicazioni multi-container su vasta scala. Kubernetes fornisce astrazioni di alto livello per la gestione di gruppi di container che consentono agli utenti di concentrarsi sul modo in cui desiderano eseguire le applicazioni, piuttosto che preoccuparsi dei dettagli specifici dell'implementazione. Kubernetes automatizza la distribuzione e lo scheduling di applicazioni containerizzate sopra un cluster in maniera efficiente.
Kubernetes permette di riavviare container che falliscono, riposizionarli o rischedularli su altri nodi quando un nodo di essi crasha. Permette inoltre scale up e scale down dei container in modo da fare load balancing sul cluster di nodi.

Nel progetto è stato utilizzato per gestire la replicazione, l'integrità ed il bilanciamento del carico dell'applicazione realizzata. Sopra kubernetes infatti sono eseguiti container contenenti le seguenti applicazioni che saranno approfondite nel dettaglio più avanti:

- Storm
- Kafka
- React
- Spring

Per la gestione del cluster su cui kubernetes opera è stato utilizzato Amazon EKS.

Servizio EKS di Amazon Web Services

Amazon Elastic Container Service for Kubernetes (EKS) è un servizio gestito per Kubernetes che semplifica l'uso di Kubernetes su AWS. EKS gestisce automaticamente la disponibilità e la scalabilità dei nodi del piano di controllo Kubernetes responsabili dell'avvio e dell'arresto di container. Con Amazon EKS è possibile sfruttare tutti i vantaggi della piattaforma AWS, prestazioni, scalabilità, affidabilità e disponibilità, nonché l'integrazione con i servizi di rete e sicurezza AWS. Amazon EKS esegue il piano di controllo Kubernetes in tre zone di disponibilità per assicurare alta disponibilità e rileva e sostituisce automaticamente i master con errori. Nel progetto è stato usato EKSCTL, un'interfaccia a riga di comando per la creazione di cluster EKS. Permette in pochi minuti di avere un cluster di macchine EC2 di un tipo predefinito ed in availability zones configurabili.

Apache Storm

ApacheStorm è un sistema di calcolo distribuito, a tolleranza di errore e open source. Esso è una delle tecnologie per lo stream processing maggiormente utilizzate, ad esempio da GROUPON, TWITTER, YAHOO, SPOTIFY, ALIBABA. Nel progetto è stato utilizzato tale framework per elaborare i flussi di dati in tempo reale provenienti dai sensori. Vengono quindi eseguiti calcoli per la risoluzione delle prime due query e per effettuare il sistema di controllo.

Vantaggi principali di ApacheStorm:

Uno dei vantaggi di Storm è la possibilità di scrivere i suoi componenti con quasi qualunque linguaggio di programmazione, nel nostro caso si è scelto Java.

Storm offre scalabilità dinamica è quindi possibile aggiungere o rimuovere i nodi di lavoro senza impatto sull'esecuzione delle topologie Storm, per utilizzare tale vantaggio è necessario disattivare e riattivare la topologia.

Una caratteristica utile di Storm è l'interfaccia utente chiamata Storm UI che consente di monitorare e gestire dal browser le topologie Storm in esecuzione.

L'architettura di Storm consente di avere dei componenti chiamati Spout i quali hanno il compito di inserire i dati ricevuti dai sensori in una topologia trasmettendo uno o piu flussi in tale topologia. I Bolt ricevono i dati dagli Spout o da altri Bolt e facoltativamente, possono trasmettere flussi nella topologia oppure scrivere in un archivio o servizio esterno, nel nostro caso la scrittura avviene nel MongoDbBolt.

Per quanto riguarda l'affidabilità, ApacheStorm garantisce l'elaborazione completa di ogni messaggio in arrivo, anche se l'analisi dei dati è distribuita tra centinaia di nodi. Il nodo Nimbus assegna attività agli altri nodi del cluster tramite Zookeeper. I nodi Zookeeper assicurano la coordinazione del cluster e consentono la comunicazione tra Nimbus e il processo Supervisor nei nodi di lavoro. Se un nodo di elaborazione si arresta, il nodo Nimbus riceve una notifica e provvede ad assegnare l'attività e i dati associati a un altro nodo.

Garantisce diversi livelli di elaborazione dei messaggi. Un'applicazione Storm di base può ad esempio garantire un'elaborazione at-least-once.
Le alternative proposte sono: ApacheFlink, Spark streaming e Twitter Heron. La scelta di ApacheStorm è dovuta al fatto che tale framework è maggiormente utilizzato e documentato e risulta didatticamente migliore perché offre delle api più a basso livello rispetto a Flink e Hearon. Spark streaming invece risulta particolarmente indicato per il batch processing, infatti può lavorare in due modalità: batch o streaming (mini-batch), la modalità streaming di Spark non è un vero e proprio stream processing poiché accumula dei mini-batch prima di processarli.

Apache Kafka

Apache Kafka è la piattaforma middleware di scambio di messaggi più popolare (publish subscribe e message queue), consente l'implementazione di tipi di elaborazione di messaggi in real-time, batch e stream processing. Offre salvataggio dello stream di dati con tolleranza ai guasti.

Essendo utilizzate dalla maggior parte delle applicazioni, Kafka deve fornire garanzie riguardo l'affidabilità, ecco perché fornisce diversi tipi semantica della comunicazione: at most one, at least one e exactly once.

Le varie semantiche vengono implementate lato consumatore mentre lato produttore il messaggio può essere riconsegnato piu di una volta al broker, in tal caso il broker mantiene l id del produttore e un numero di sequenza del messaggio in modo tale da rendere le operazioni idempotenti e quindi non effettuare di nuovo lo stesso calcolo più di una volta. Si può specificare che il produttore invii il messaggio aspettando l'ack per un tempo prefissato oppure che la chiamata avvenga in maniera del tutto asincrona. Nel sistema in considerazione la chiamata del produttore viene eseguita in maniera asincrona poiché non si necessita di avere un tempo di risposta preciso.
Per quanto concerne il consumatore, è possibile implementarlo rispettando una delle 3 semantiche messe a disposizione: At-most Once, At-least Once o exactly once.

La semantica at-most once è la semantica di default utilizzata dal sistema progettato. Una risposta se calcolata, lo è stata al più una volta.

Kafka è integrato con ApacheStorm al fine di elaborare i dati in tempo reale e in streaming.

Per quanto concerne il funzionamento, la piattaforma viene distribuita come cluster su uno o più server. Il cluster ha la capacità di memorizzare vari topic, ovvero dei flussi di record. Ogni record contiene al suo interno tre dettagli, ossia una chiave, un valore e un timestamp. L'architettura è basata su quattro API principali: Producer API, che permette alle applicazioni di pubblicare flussi in uno o più topic; Consumer API, che consente l'elaborazione dei topic e del flusso prodotto dai record; Stream API, la quale riceve un determinato input dai topic e produce degli output, convertendo i flussi; Connector API, la quale gestisce il collegamento tra varie applicazioni.
Nell'applicazione si è scelto di utilizzare Apache Kafka per poter gestire il flusso di dati in arrivo ad Apache Storm in modo più efficiente possibile e per poter gestire l'eterogeneità dei dati.

MongoDb Atlas

Per quanto riguarda la persistenza dei dati è stato utilizzato MongoDB sul cluster di Atlas , un DBMS non relazionale orientato ai documenti. Classificato come un database di tipo NoSQL, MongoDB salva i documenti in stile JSON con schema dinamico (MongoDB chiama il formato BSON), rendendo l'integrazione di dati di alcuni tipi di applicazioni più facile flessibile e veloce.

La scelta di un database non relazionale è stata condizionata dai suoi vantaggi, MongoDB risulta meno complesso, altamente disponibile, scalabile orizzontalmente ed ottimizzato per gestire grandi quantità di dati a discapito di una consistenza meno forte.

La scelta di MongoDB rispetto ad altri non relazionali è dettata principalmente dal fatto che MongoDB ha una grande community e quindi è documentato molto bene ed offre una forte integrazione col framework di Storm, infatti Storm mette a disposizione delle classi per utilizzare Mongo,tra cui i MongoDB bolt.

MongoDB è nato per sfruttare la scalabilità orizzontale, il modello document-oriented permette naturalmente il frazionamento del database su diverse macchine e questo DB fornisce di default alcuni strumenti che ci permettono di gestire facilmente questo aspetto ed occuparci principalmente della programmazione. L'uso di MongoDB è concentrato in applicazioni Real-time.
L'aspetto di maggior interesse comunque resta le incredibili performance che questo DB può esprimere. MongoDB, infatti, usa quanta più RAM possibile come cache e cerca automaticamente di scegliere il miglior indice per le query, tutto è stato disegnato per le performance.

Per quanto riguarda l'uso che ne è stato fatto nel progetto, vengono salvati in MongoDB i risultati delle due query in due collection diverse, sdccMedian e sdccRank, le quali contengono entrambe tre documenti, uno per ogni finestra temporale.
All'interno delle finestre di sdccMedian troviamo l'id della finestra, la mediana globale degli incroci e una lista di: id incrocio e mediana locale.
Per quanto riguarda le finestre di sdccRank troviamo l'id della finestra e una lista di: id incrocio, valore nella classifica e velocità media.
È risultato utile utilizzare MongoDb per salvare i documenti in maniera annidata senza aver bisogno di effettuare operazioni di join.
Nella collection stateTrafficLight troviamo lo stato dei semafori, quindi abbiamo una serie di campi che identificano il numero del semaforo nell'incrocio, l'id dell'incrocio a cui appartiene il semaforo e una lista di 3 valori che danno informazioni sulle 3 luci del semaforo, se il valore è OK la luce non è guasta, se è diverso da OK allora è guasta.

Nella tabella sdccIntersection sono salvate le informazioni degli incroci e altre informazioni base impostate dall'amministratore della piattaforma.
È presente per ogni incrocio una lista di 4 sensori che mantiene: id incrocio, numero del sensore nell'incrocio, saturazione, latitudine e longitudine, una lista di 2 elementi chiamata fase che mantiene per ogni elemento un id, un tempo di verde e un tempo di rosso e l'id dell'incrocio.
La fase e la saturazione sono dei parametri che servono all'algoritmo di controllo di Webster che verrà spiegato nei prossimi paragrafi.

React

Per la realizzazione del front-end dell'applicazione è stata utilizzata la libreria di Facebook, React.

React usa il linguaggio JSX, che è fondamentalmente un modo dichiarativo di inserire HTML in JavaScript.
La caratteristica fondamentale di React è il concentrarsi sulla parte visuale dell'applicazione suddividendola in blocchi (componenti) che contengono il proprio stato e la propria logica di gestione, mentre la comunicazione con il server o la strutturazione dei moduli sono demandati a librerie esterne.
Il vantaggio di tale libreria in confronto ad altre è quello del riutilizzo del codice, è possibile rendere i componenti il più possibile autonomi e riutilizzabili in un medesimo contesto, l'approccio di React è "Separation of Concerns" al contrario di altri framework che si concentrano sulla separazione della logica di gestione degli eventi dalla presentazione, cosa che non avviene in React.

Nell'applicazione React viene utilizzato per la visualizzazione dei risultati delle query, le statistiche di ogni topologia e gli incroci con i rispettivi sensori. E' possibile aggiungere un nuovo incrocio oppure eliminarlo o modificarlo. E' possibile riparare un semaforo rotto. E' presente una side-bar sul lato sinistro che rappresenta il menù ed è possibile trovare le varie viste.

E' stata inoltre utilizzata la libreria Redux per la gestione dello stato, in particolare con Redux è possibile aggiornare lo stato del front-end in maniera temporizzata così da essere consistenti con le modifiche apportate dalle query e dal sistema di controllo.

Spring

Spring, grazie all'utilizzo dei template e dell'Inversion of Control, facilita l'utilizzo della tecnologia di MongoDB per l'accesso ai dati, esonerando lo sviluppatore dalle attività di contorno.
Il concetto di base della Dependency injection consiste nell'avere un oggetto separato che si occupi della risoluzione delle dipendenze tra gli oggetti e della loro inizializzazione.

Simulatore

Come fase preliminare il db di Mongo viene correttamente riempito con dei dati riguardanti tutti gli incroci e tutti i semafori di ogni incrocio. Tali dati vengono immessi attraverso uno script simulativo iniziale.
Nello script iniziale le informazioni degli incroci e dei semafori vengono prodotte in modo tale che ogni entità occupi una posizione precisa all'interno di una griglia randomica, necessario per simulare un quadrante di una città.
Tra le ulteriori informazioni troviamo la durata della luce verde, la durata della luce rossa, lo stato delle lampade e la saturazione per ogni semaforo.
Una volta fatto ciò viene avviato il simulatore vero e proprio.
Il simulatore nel progetto agisce da procuder di Kafka, infatti ogni intervallo temporale costruisce tutte le tuple da inviare raccogliendo i dati presenti nelle collections del database di Mongo. Una volta costruite le tuple le invia al broker di Kafka su un unico topic.
Durante la costruzione della tupla vi è una probabilità di rottura delle lampade del semaforo che si sta processando.
Una volta che le tuple vengono processate dai Kafka Spout essi le inviano al bolt successivo per far iniziare il vero processamento delle varie topologie.
Tutte le topologie hanno come ultimo bolt un MongoDB bolt che ha lo scopo di aggiornare le informazioni sulle varie collezioni in modo tale che in un successivo intervallo temporale il producer di Kafka generi le tuple coerentemente allo stato delle informazioni presenti nel DB.

Apache Storm: Topologie

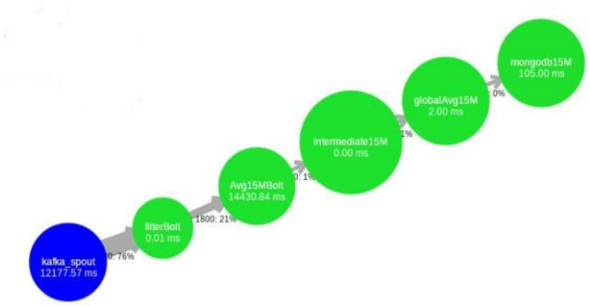
Come detto in precedenza per il processamento dei dati è stato utilizzato il framework ApacheStorm.

Tale framework è necessario per soddisfare i requisiti del progetto.

Tali requisiti sono:

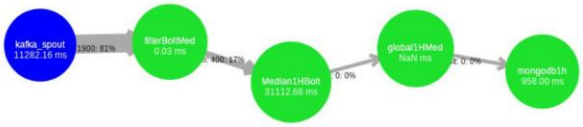
- Fornire una risposta alle 2 query sullo stato del traffico.
- Implementare un sistema di controllo del traffico (implementato mediante l'algoritmo di Webster).
- Generare un allarme in seguito al guasto di una lampada di un semaforo.

Query 1



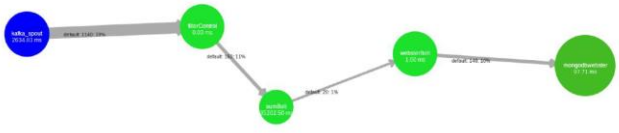
L'intento della query 1 è quello di calcolare la classifica dei TOP K Incroci (con K=10) con la velocità media maggiore, in finestre temporali da 15 minuti,1 ora e 24 ore.
Per far questo vengono calcolate più classifiche intermedie parziali per poi ordinarle in una sola classifica.
I principali bolt della topologia sono: AvgBolts, IntermediateRankBolts e il GlobalRankBolt.
Gli AvgBolts processano nella finestra temporale le medie delle velocità degli incroci, per poi passarle a degli intermediateRankBolts che calcolano tutte le classifiche intermedie che verranno poi passate al GlobalRankBolt che non deve far altro che unirle ordinatamente per creare un'unica classifica.
Il vantaggio di creare classifiche intermedie permette all'unico GlobalRankBolt di unire in modo ordinato le classifiche senza impiegare troppo tempo, evitando così che faccia da collo di bottiglia.

Query 2



Lo scopo della query 2 è quello di calcolare le intersezioni che hanno la mediana del numero di veicoli superiore al valore della mediana globale dei veicoli che hanno attraversato tutte le intersezioni.
Per calcolare la mediana esatta occorrerebbe ordinare tutti i valori del dataset, che risulterebbe molto inefficiente per lo scopo del data-stream processing, quindi è stato scelto di utilizzare l'algoritmo basato su tdigest, il quale permette di calcolare la mediana progressivamente e in forma approssimata senza la necessità di ordinare man mano il dataset.
I principali bolt della topologia sono: MedianBolts e il GlobalMedianBolt.
I MedianBolts calcolano la mediana del numero di veicoli per singolo incrocio, e un GlobalMedianBolt che calcola la mediana globale e memorizza gli incroci con la mediana maggiore della mediana globale.
L'utilizzo del tdigest evita i colli di bottiglia dovuti all'ordinamento dei dati, pur mantenendo un piccolo errore sul valore della mediana.

Query sistema di controllo



La query "Sistema di controllo" calcola i valori ottimi localmente (ossia per singolo incrocio) della durata della lampada rossa e della lampada verde di ogni semaforo e per ogni Incrocio.
Per calcolare le durate delle luci si utilizza l'algoritmo di Webster in modo da minimizzare il traffico stimato sugli incroci.

Supponendo che gli incroci siano sufficientemente distanti, quindi che le durate della luce Verde e della luce Rossa siano indipendenti tra gli Incroci, è ragionevole pensare che l'algoritmo di Webster sia utile non solo per il singolo Incrocio ma anche per sistemi di incroci, come i quadranti delle città.
Per applicare l'algoritmo di Webster occorre sapere il valore della saturazione che è costante per ogni semaforo e il flusso dei veicoli che attraversano ogni semaforo.
Per semplicità si è assunto che i valori di saturazione siano noti per ogni semaforo.
Per calcolare invece il flusso veicoli/s di veicoli sono stati implementati dei SumBolts che sommano il valore del numero di veicoli per ogni incrocio in finestre temporali. Per ottenere il flusso dei veicoli occorre dividere tale somma per la lunghezza della finestra temporale.
Una volta calcolato il flusso per ogni incrocio viene eseguito l'algoritmo di Webster per ottenere le nuove durate delle lampade Verdi e Rosse.

Query Stato dei semafori

La query "stato dei semafori" è una semplice topologia che ha lo scopo di capire quali Incroci contengono semafori con lampade guaste.
Per fare ciò la topologia filtra tutte le tuple ricevute in 2 gruppi disgiunti: le tuple che rappresentano semafori perfettamente funzionanti e le tuple che rappresentano semafori con almeno una lampada guasta.
La topologia non fa altro che ricevere le tuple, scartare quelle che non hanno lampade guaste e mantenere quelle con lampade guaste.

Calcolo del throughput e della latenza

Per il calcolo del throughput e della latenza sono state sfruttate le api RESTFUL di Storm che ci permettono di ottenere tutte le informazioni sulle topologie attive.
Una volta ottenute le informazioni sotto forma di json, esse sono state filtrate per prelevare solo i dati per la latenza e le tuple emesse.
Facendo più chiamate RESTFUL ad intervalli di tempo regolari è possibile raccogliere un insieme di dati utili per calcolare le statistiche principali.
L'insieme dei valori della latenza è preso cosi come è, infatti ogni chiamata RESTFUL ci dà il valore della latenza in un preciso istante temporale. Invece per quanto riguarda le statistiche sul throughput non è possibile ottenerle semplicemente prendendo il valore delle tuple emesse, in quanto tale valore aumenta progressivamente e sta ad indicare il numero delle tuple emesse dall'avvio della topologia, ossia dall'istante 0 all'istante t. Quindi una volta ottenuta la sequenza delle tuple emesse, per ottenere il numero di tuple emesse in un certo intervallo temporale [t, t+L] occorre fare la differenza tra 2 valori consecutivi della sequenza.
Per esempio, se indichiamo con TupleEmesse(t, t+L) le tuple emesse nella finestra temporale [t, t+L] e con TupleEmesse(0,t) le tuple emesse dall'istante 0 all'istante t, allora la formula finale per il throughput diventa:
Throughput=TupleEmesse(t,t+L)/L=(TupleEmesse(0,t+L)-TupleEmesse(0,t))/L.
Notiamo che se L=1 sec, abbiamo la classica formula del throughput, in quanto ogni secondo andiamo a vedere quante tuple vengono emesse, se L>1 otteniamo una formula che tiene conto del fatto che le tuple emesse sono state emesse in un intervallo temporale maggiore di un sec, e quindi vanno opportunamente divise per la lunghezza dell'intervallo.

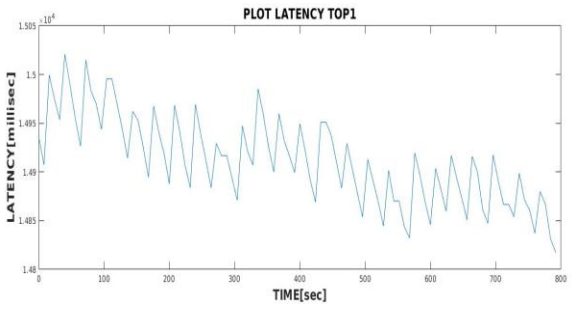


Figure 1: grafico della latenza in funzione del tempo della topologia 1.

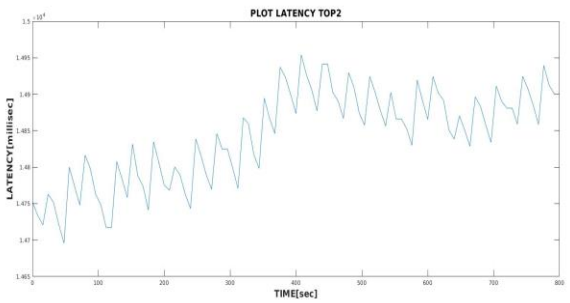


Figure 2: grafico della latenza in funzione del tempo della topologia 2.

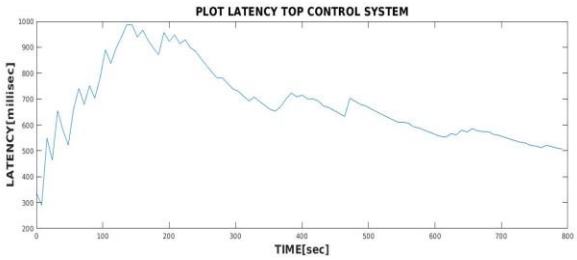


Figure 3: grafico della latenza in funzione del tempo della topologia del Sistema di controllo.

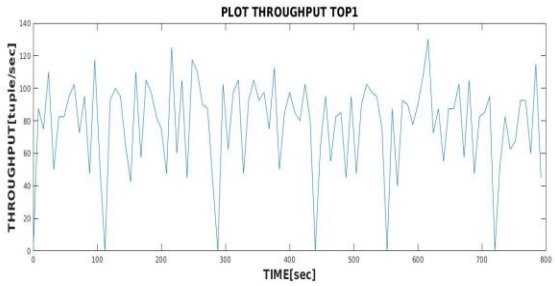


Figure 4: grafico del throughput in funzione del tempo della topologia 1.

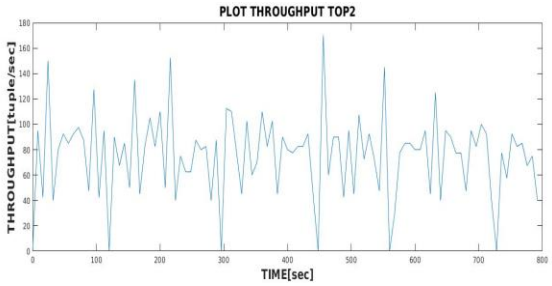


Figure 5: grafico del throughput in funzione del tempo della topologia 2.

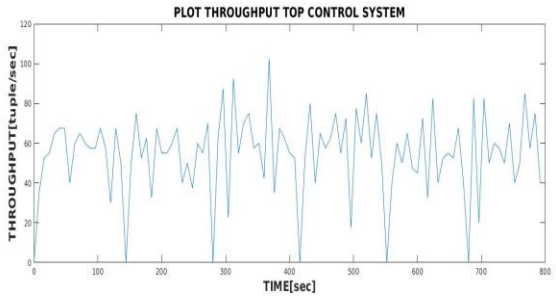


Figure 6: grafico del throughput in funzione del tempo della topologia del Sistema di controllo.

Conclusioni

I requisiti richiesti sono stati rispettati grazie all'utilizzo dei servizi e dei framework precedentemente descritti. I principali problemi riscontrati nello sviluppo dell'applicazione sono stati molteplici.

ll più importante è stato l'integrazione dei framework ApacheStorm e ApacheKafka ed il loro deploy sulla piattaforma Cloud di Amazon.

Un'altra difficoltà è stata la comprensione di Kubernetes e di EKS, non avendo alcuna esperienza con il deploy di un'applicazione su una piattaforma cloud.

Tali problemi sono stati risolti soprattutto tramite i numerosi progetti Open-Source presenti sulla piattaforma GitHub e anche grazie alle documentazioni ufficiali offerte dai creatori dei framework.

Per eventuali sviluppi futuri è possibile implementare in modo migliore le finestre di ApacheStorm, non sfruttando l'implementazione fornita da quest'ultimo e inserire un sistema publish-subscribe oppure un sistema a code di messaggi per interfacciare l'output delle query con altri servizi, in modo più efficiente.

RIFERIMENTI

Storm:
https://docs.microsoft.com/it-it/azure/hdinsight/storm/apache-storm-overview
http://storm.apache.org/releases/current/Powered-By.html
https://stackoverflow.com/questions/30699119/what-is-are-the-main-differences-between-flink-and-storm

Kafka:

https://dzone.com/articles/kafka-clients-at-most-once-at-least-once-exactly-o
https://medium.com/@ajmalbabu/kafka-0-9-0-clients-db1f43257d30
https://www.confluent.io/blog/enabling-exactly-kafka-streams/

MongoDB:

http://www.webmt.it/blog/quando-usare-un-database-document-oriented

EKS:

https://aws.amazon.com/it/eks/features/

Kubernetes:

https://kubernetes.io/