Analisi del traffico sfruttando il fog computing

Luca Pepè Sciarria
Dipartimento di Ingeneria Civile e Ingegneria Informatica
Università di Roma Torvergata
luca.pepesciarria@gmail.com

Francesco Gambardella
Dipartimento di Ingeneria Civile e Ingegneria Informatica
Università di Roma Torvergata
francesco.gambardella@alumni.uniroma2.eu

***Abstract*—Oggigiorno il cloud computing è diventata una tecnologia consolidata e ampiamente usata. Anche se essa comporta molti vantaggi per gli sviluppatori risulta avere dei lati negativi quali alta latenza, mancanza di mobilità e location-awareness. Per ovviare a questi problemi è sorto un nuovo paradigma: il fog computing. Lo scopo principale del fog computing è quello di spostare i nodi di computazione il più possibile vicino agli utenti finali per poter diminuire la latenza di comunicazione. In questo articolo discutiamo l'applicazione del fog computing analizzando un caso pratico quale l'analisi del traffico in tempo reale, use case tipico nell'idea delle smart cities.**

Keywords—fog computing; distributed system; cloud computing;

I. INTRODUZIONE

La pervasività dei dispositivi mobili quali smartphone, l'idea dell'Internet of Things e la grande mole di dati che essi generano richiede un approccio diverso nella costruzione di un'architettura per lo sviluppo delle applicazioni. L'uso del solo Cloud computing, nonostante le caratteristiche di scalabilità e di alta disponibilità per l'utente finale oltre ad una gestione semplificata per la memorizzazione di informazioni e l'elaborazione di grandi moli di dati, può in questi casi non essere sufficiente e portare a soluzioni non efficienti, sopratutto nel caso di sistemi real-time che richiedono scadenze temporali stringenti. Un sistema di monitoraggio e analisi del traffico rientra proprio nella categoria di sistemi sopra discussa. infatti la possibilità di ricevere informazioni da diversi dispositivi. quali sensori IoT oppure smartphone in aggiunta all'esigenza di processamento di tali dati in tempi stringenti per poter rispondere a richieste sull'andamento del traffico in un dato istante, può necessitare di soluzioni alternative al cloud computing per avere un risultato soddisfacente. In questo articolo viene analizzato lo sviluppo di un'applicazione in grado di risondere a tali requisiti non funzionali. L'obiettivo del progetto è dunque la realizzazione di un sistema in grado di ricevere dati dagli utenti relativi alla loro posizione e velocità attuale per calcolare una media della velocità del traffico rispetto alla posizione geografica dichiarata. Tale dato deve essere dunque reso disponibile in tempi brevi per poter rispondere ad eventuali richieste sull'andamento del traffico avanzate dagli utenti che accedono al sistema tramite il proprio smartphone, tutto ai fini di migliorare l'esperienza alla guida degli stessi. Nello specifico, il progetto si compone di due applicazioni tra loro complementari: la prima applicazione ha lo scopo di costruire la topologia della città mentre la seconda utilizza la topologia per il calcolo in real-time della velocità su ogni strada.

Nel seguito verrà presentato il design del sistema (§2) in cui verrà descritta la creazione del grafo. Successivamente verrà descritto l'architettura del sistema (§3) e come i vari componenti interagiscono e quindi a come il tutto è stato implementato (§4). Verranno in fine mostrati i risultati e le prestazioni (§5). In conclusione, inoltre, vengono discussi eventuali sviluppi futuri (§6) e la guida all'installazione (§7).

II. DESIGN

In questa sezione si vuole illustrare il design del sistema. L'applicazione è suddivisa in due moduli: il primo responsabile della ricostruzione della topologia delle strade sfruttando le informazioni fornite dagli utenti riguardo i loro spostamenti geografici, mentre il secondo del calcolo della velocità media attuale delle singole strade riconosciute, sfruttando i risultati che il primo modulo ha generato. Di seguito è prima discussa la concezione e l'organizzazione dell'ambiente in cui l'applicazione deve operare per poterne poi spiegare come, con determinate scelte, si possa facilmente applicare l'approccio fog computing e trarne ampi vantaggi.

A. Creazione del grafo

L'idea di fondo che ha permesso la ricostruzione della topologia delle strade ed una più facile gestione dei dati provenienti dagli utenti consiste nella suddivisione della mappa della città in diverse sezioni, così da creare una griglia su cui operare. Data la mappa della città essa è stata suddivisa come mostrato in Figura 1: la mappa è stata partizionata in diverse sezioni, ognuna delle quali grande 1km². Con questa suddivisione è stato possibile attuare una gestione mirata dei dati generati da ogni dispositivo. In particolare ogni sezione è a sua volta suddivisa in sottosezioni quadrate di lato pari a 10 metri. Tale scelta è stata fondamentale per gestire e sfruttare le informazioni riguardo gli spostamenti degli utenti. Trattando ognuna di queste sottosezioni come un nodo di un grafo, è possibile tracciare le traiettorie di movimento degli utenti tenendo in considerazione della loro transizione tra una

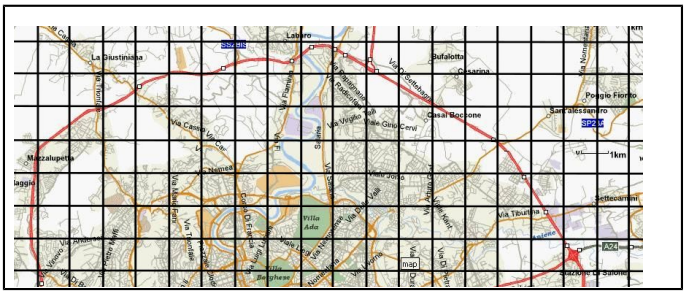

Fig. 1. Suddivisione in sezioni di Roma nord.

sottosezione ed un' altra: ogniqualvolta avviene questo passaggio il dispositivo si prepara ad inviare al sistema una coppia di coordinate *[(x1, y1);(x2, y2)]* che indica lo spostamento appena effettuato. In particolare la singola coordinata (x, y) permette di identificare una singola sottosezione della griglia presa in questione e di conseguenza un nodo del grafo. Tramite la coppia di coordinate è dunque possibile rappresentare lo spostamento percepito attraverso un arco orientato dalla prima coordinata alla seconda, rendendo così nota al sistema la presenza di una strada. La velocità media con cui l'automobile si è spostata tra questi due nodi viene usata per aggiornare il valore dell'etichetta che è associata all'arco. Da tutti i dati che si ricevono è possibile costruire un grafo per ogni sezione. Il partizionamento attuato porta con sé ulteriori vantaggi oltre alla semplice gestione della rete stradale: tramite questa scelta è infatti anche possibile svolgere una più semplice distribuzione del carico computazionale per il sistema, facendo sì che ogni sezione della griglia sia gestita da un opportuno nodo fisico, possibilmente prossimo geograficamente come prevede il paradigma di fog computing.

B. *Processamento del grafo*

Una volta che il grafo è completo esso avrà una struttura come in Figura 2 in cui è possibile vedere 2 tipi di nodi:

- *Nodi incrocio*: sono nodi che presentano 3 o più archi uscenti. Indicano l'inizio o la fine di una o più strade.
- *Nodi transitori*: nodi con al più 2 archi uscenti. Indicano nodi che appartengono ad una strada.

Sfruttando tale distinzione è possibile riconoscere e ricavare tutte le strade: si intende per strada un insieme di archi che vanno da un nodo incrocio ad un altro. In una strada sarà presente una ed una sola velocità media data dalla media pesata delle velocità su ogni arco che la compongono. Il motivo per cui è stato speso tale effort nel riconoscimento delle strade risiede principalmente nella convinzione che tramite esse si riesca a calcolare un valore medio più significativo, rispetto a quello puntuale che si avrebbe considerando un singolo arco, tenendo conto dei dati provenienti dalla totalità degli archi e quindi della distribuzione del traffico sulla strada stessa. Si vuole far notare che tale definizione di strada non ha lo scopo di generare una corrispondenza biunivoca tra le vie di una città e quelle che sono percepite dal sistema. Prendendo ad esempio la strada "via Casilina" di Roma, è interesse del sistema non riconoscerla nella sua interezza, così come avviene in uno stradario, ma saperla suddividere nei suoi diversi segmenti (cioè le effettive strade trattate dal sistema) che partono e terminano con un incrocio e che possono presentare velocità medie significativamentre diverse.

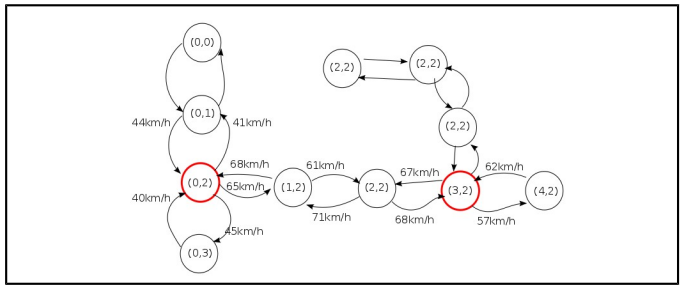

Fig. 2. Possibile grafo di una sezione. In rosso i nodi incrocio. Ad ogni arco è associata una velocità.

C. *Aggiornamento della velocità*

Avendo a disposizione le strade, il calcolo e l'aggiornamento della velocità sulle stesse risulta essere relativamente semplice. Tale passaggio può essere riassunto distinguendo due fasi: la prima, in cui avviene l'aggiornamento della velocità media dell'arco preso in input, che viene eseguito mediando in modo pesato il valore più recentemente ricevuto con il valore attuale, questo allo scopo di avere informazioni quanto più possibile inerenti alla condizione del traffico corrente; la seconda si preoccupa invece di riportare le modifiche avvenute sul singolo arco alla strada cui esso appartiene. In quest'ultimo caso si è pensato di pesare i contributi dei singoli archi in base al loro rapporto rispetto alla lunghezza totale della strada, intesa come il numero totale degli archi che la compongono.

III. ARCHITETTURA

In questa sezione verrà illustrata l'architettura proposta per l'applicazione. Si andrà prima a discutere l'architettura software per poi successivamente presentare l'architettura di sistema, mostrando quindi come le diverse componenti logiche dell'applicazione vengono istanziate sui nodi e come questi sono a loro volta collocati fisicamente.

A. *Architettura software*

L'architettura software è suddivisa come mostrato in Figura 3. I vari componenti che la compongono sono:

- Kafka
- Storm
- Redis
- ElasticSearch

Kafka. Kafka permette l'utilizzo di un sistema publish-subscribe; in questo modo è possibile avere un maggior disaccoppiamento tra le sorgenti dei dati ed i consumatori presenti nell'applicazione proposta. Inoltre, esso costituisce l'unico punto di accesso al sistema per fornire le informazioni; ciò consente dunque una semplificazione della gestione dei produttori presenti nei client, i quali necessitano la sola conoscenza di questo componente.

Storm. *È* il componente responsabile del data stream processing. La scelta di tale framework è stata adotatta in quanto naturalmente integrata con kafka: è possibile fare in modo che gli stessi broker di kafka abbiano funzione di spout per storm e svolgano direttamente una funzione di data injection. In particolare sono state realizzate due diverse

topologie, ognuna relativa ad uno dei moduli dell'applicazione di cui si è parlato in precedenza.

- *Topologia A:* è la topologia che implementa la logica di creazione del grafo. È composta da tre livelli: un primo livello costituito dai ***kafkaSpout***, mentre i successivi due da *splitterBolt* ed ***areaBolt***. La componente kafkaSpout riceve dati da kafka, che si presentano con il formato di jsonArray composto da un numero variabile di oggetti: *[{"x1", "y1", "x2", "y2", "speed"}, . . . , {. . .}]*; ogni oggetto è quindi costituito da una coppia di coordinate indicante un arco e la relativa velocità come mostrato nel formato di esempio, in cui la coppia ("x1","y1") rappresenta la sezione di partenza e la coppia ("x2", "y2") quella di arrivo, secondo l'idea presentata nella sezione *Design (§2)*. A questo punto l'array viene inoltrato al livello degli **splitterBolt**, il cui compito consiste nello "spacchettare" l'array ricevuto ed inviare i singoli oggetti al livello successivo di bolt. Infine **areaBolt**, ricevuti i dati provenienti dagli **splitterBolt**, si preoccupa di implementare l'effettiva logica di aggiornamento del grafo; ciò avviene sia a livello locale, aggiornando una struttura "in-memory" per il singolo worker, sia ad un livello "globale" interfacciandosi con la componente di persistenza di *Redis.*

- *Topologia B:* Come per la topologia A, questa presenta un primo livello di **kafkaSpout** per le funzioni di data injection, a cui segue un livello di **splitterBolt**; le operazioni svolte dai due livelli sono le medesime di quelle presentate nella topologia A. Successivamente sono presenti ulteriori tre livelli di bolt composti rispettivamente da ***meanCalculatorBolt***, ***updateMeanStreetBolt***, ***producerBolt***. Il **meanCalculatorBolt** ha il compito di processare la tupla relativa al singolo arco ricevuta dallo **splitterBolt**, estraendone i dati di interesse; in seguito è sua responsabilità propagare l'informazione al livello successivo, inviando una tupla composta dall'identificativo dell'arco appena processato e della nuova rispettiva velocità. **UpdateMeanStreetBolt** riceve dunque i dati prodotti dal **meanCalculatorBolt** e si preoccupa di aggiornare il valore della velocità media della strada contenente l'arco inoltrato secondo le modalità già introdotte in principio. Per poter svolgere tale mansione il più efficientemente possibile, si è deciso di fare in modo che le modifiche eseguite dalla componente avvengano in locale, su dati prelevati da Redis durante la fase di *prepare*, in modo tale da evitare ritardi, durante la fase di *execute*, relativi alle

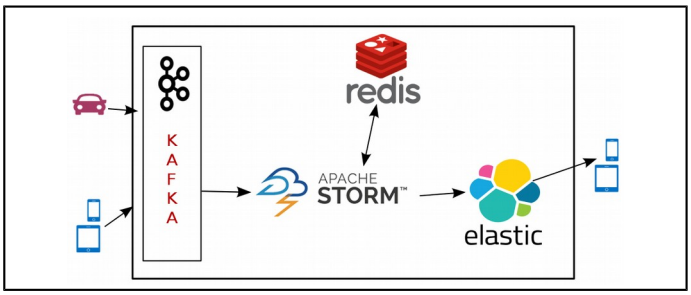

Fig. 3. L'architettura del sistema prevede Kafka quale *entry point* mentre ElasticSearch fornisce i dati agli utenti. Kafka esegue anche il compito di Spout e quindi di *fata injection*. Redis memorizza i dati relativi alla topologia dela città ella composizione delle singole strade.

connesioni e alle operazioni sul database. La componente di Redis risulta dunque essere funzionale all'inizializzazione del Bolt, il quale può dunque essere in grado, in caso di ripresa da fallimenti, di recuperare i dati relativi alle strade su cui deve operare e riprendere la propria esecuzione senza ledere alla logica del sistema (questo grazie anche alla natura dei dati del traffico che vergono velocemente al valore di regime pur partendo da uno stato Idle). In alternativa, è stata anche variata la possibilità di aggiungere una nuova componente Bolt "parallela" a quella di Elastic Search, responabile di ricevere i dati dell'updateMeanStreetBolt e renderli persistenti su Redis, ma al momento non si ritiene essere strettamente necessaria. Al termine dell'albero della topologia Storm è presente il producerBolt il quale, ricevendo in input dal livello precedente la chiave di una strada e la sua velocità media, è responsabile di riportare le informazioni ricevute alla componente di ElasticSearch, rendendole persistenti e disponibili agli utenti finali che potranno eseguire le loro query.

Redis. È un database noSql, di tipo chiave-valore, in-memory. È stato scelto principalmente per la sua velocità operazionale e come supporto al calcolo effettuato da storm, per introdurre una componente di persistenza al sistema. Nel dettaglio, per ogni regione sono presenti tre hashMap:

- *graphArea*, per la memorizzazione dei nodi e relativi archi presenti in una sezione. La chiave risulta essere una stringa identificativa del nodo mentre il valore è il nodo stesso, con i rispettivi archi, che viene salvato come stringa in formato json;

- *edges*, usato per tenere traccia delle corrispondenze tra gli archi e le strade di appartenenza. In partricolare viene memorizzato per ogni arco la chiave identificativa della strada;

- *streets*, memorizza per ogni strada, identificata tramite un intero, la corrispettiva velocità media attuale.

A livello di architettura software, tale componente si sviluppa in tre processi master tra cui si distribuisce il lavoro, ognuno dei quali presenta un grado di replicazione pari a 3 per aumentare la tolleranza ai guasti del sistema. Si vuole far notare che la consistenza offerta da Redis è di tipo finale, ma ciò non costituisce problema agli scopi dell'applicazione, dato che i valori delle velocità risultano essere per conto loro approssimati e l'esigenze degli utenti non richiedono una

stretta precisione; inoltre una possibile inconsistenza verrebbe mascherata dall'applicazione grazie alla pesatura con cui avvengono gli aggiornamenti.

ElasticSearch. Permette l'indicizzazione dei documenti, facendo in modo di rispondere alle esigenze degli utenti finali riguardo alla condizione del traffico corrente (ricavabile dai valori delle velocità medie delle strade) e la conoscenza della topologia della rete stradale. Analogamente alla componente di Kafka, questo costituisce l'unico punto di accesso al sistema per gli utilizzatori, non più per fornire dati, bensì per risolvere le query mosse dagli utenti sfruttando il protocollo *http*. I documenti salvati in ElasticSearch prevedono una struttura in formato json. Ogni documento rappresenta una strada e contiene i seguenti dati:

- *streetKey:* chiave attraverso cui una strada è identificata univocamente;
- *edges:* un array di oggetti json ognuno dei quali rappresenta un arco appartenente alla strada;
- *section:* sezione a cui la strada appartiene
- *speed:* valore della velocità media attuale rilevata sulla strada in questione.

Tramite queste informazioni è possibile dunque effettuare le seguenti query a favore degli utenti:

- data la chiave di una strada, è possibile accedere ai dati a questa collegati. Al momento la chiave viene generata in modo artificiale seguendo un meccanismo di enumerazione incrementale durante la fase di inizializzazione del sistema quando si processano i vari archi per definire le strade; in futuro si vuole fare in modo che tale chiave sia in qualche modo connessa al nome della strada in questione, in modo tale da poter svolgere ricerche per nome;
- data una sezione restituirne tutti i dati relativi cioè l'insieme di tutte le strade in essa contenute con relative velocità ed archi associati;
- dato un arco, essere in grado di risalire alla strada a cui esso appartiene. Questa query permette di trovare la velocità di una data strada di cui non si conosce l'id ma soltanto l'arco (uno dei) che la compone.

B. *Architettura di sistema*

A livello teorico, è stata progettata un'architettura seguendo i paradigmi del "Fog computing", in cui si vuole che i nodi fisici siano il più possibile prossimi agli utenti. Come anticipato nel paragrafo 2 della trattazione, idealmente si vuole tenere conto della suddivisione dell'area della città in più sezioni, ognuna delle quali vuole essere di responsabilità di uno specifico nodo (possa esso essere un server od un cluster). In questo modo ogni dispositivo che vuole inviare o ricevere dati, può farlo contattando il nodo a lui più vicino, riducendo così al minimo le latenze di comunicazione. In particolare, si costituisce uno scenario in cui in corrispondenza di ogni sezione, è possibile trovare: un nodo fisico appartenente al cluster di Kafka responsabile del data injection; una componente cluster di storm responsabile del processing dei dati relativi alla sezione in cui giace; un nodo fisico, potenzialmente replicato, appartenente al cluster di Redis; ed infine un nodo appartenente al cluster di ElasticSearch per rispondere alle query mosse dagli utenti che transitano nella sezione in esame. In atto pratico, tuttavia, non è stato possibile eseguire un tale deploy; il sistema è stato dunque strutturato come segue:

- un nodo fisico per istanziare la componente Kafka
- un clustrer per la componente storm, composto da un nodo ospitante zookeeper, un nodo per ospitare il processo *Nimbus* responsabile della distribuzione del lavoro ai Supervisor, ed ulteriori due nodi *Supervisor* che ospitano i processi *worker.*
- un nodo fisico su cui è stato istanziato il cluser logico della componente Redis descritto in precedenza
- un cluster composto da tre nodi fisici per la componente di ElasticSearch.

IV. Implementazione

In questa sezione si tratteranno i dettagli implementativi con cui sono state realizzati i concetti principe che hanno caratterizzato lo sviluppo dell'applicazione. È stato adottato il processo di sviluppo Agile, con iterazioni di 2 settimane a cui è seguito uno Sprint Review Meeting, in cui è avvenuto l' incontro con gli Stakeholders.

A. *Partizionamneto della mappa*

Per realizzare la corrispondenza tra una specifica sezione e la componente storm responsabile al processamento dei relativi dati, sono state sfruttate le informazioni ottenibili direttamente dalle coordinate GPS. Per rendere chiaro tale passaggio si necessita tuttavia una digressione ed un approfondimento sulle specifiche dei dati GPS. Le coordinate si presentano come una coppia di valori decimali rappresentanti latitudine e longitudine. Un generico valore per latitudine o longitudine si presenta sotto il formato:

xy.abcde

in cui è possibile attribuire il seguente significato ad ogni cifra decimale: sia *d* l' i-esima cifra decimale, per i = 1, 2, 3, 4, 5, essa rappresenta il valore di $10^{(5-i)}$ metri. Sotto questa rappresentazione, è dunque possibile ricavare, dati due valori, anche la distanza spaziale tra loro, e da qui anche la grandezza di una sezione. A questo punto, attuare una distinzione tra sezioni di appartenenza date le coordinate e nota la dimensione della sezione è relativamente semplice; in particolare, nel caso del sistema realizzato in pratica, in cui è stata definita una quadrisezione della città, è possibile realizzare il partizionamento facendo in modo di associare ogni Spout di storm al range di valori delle coordinate che ricoprono l'area di cui è responsabile e ciò può essere facilmente ottenuto attuando delle discriminazioni sui valori delle prime due cifre decimali, attraverso cui è possibile sancire i confini di ogni sezione. Si vuole far notare che quanto detto adesso non vuole costituire una inconsistenza con quanto presentato nel paragrafo 2 della trattazione: è stata infatti già discussa la differenza tra la progettazione teorica sperata per il sistema e quella che è stata

praticamente possibile realizzare attualmente con i mezzi a disposizione. Ne consegue che è stato necessario modificare anche il piano di partizionamento della mappa cittadina per potersi adattare all'effetiva istanziazione dei nodi fisici e cercare di mantenere un adeguato carico di richieste per ogni nodo. Per rendere più semplice l'uso dell'applicazione è stato dunque inserito come parametro della stessa. La possibilità di poter decidere la dimensione delle sezioni con cui ripartire il grafo può costituire un punto di forza del sistema, il quale è potenzialmente in grado di adattarsi a diversi scenari e ambienti con diversa scala geografica. Proseguendo il discorso, andando a considerare invece le variazioni sulla quarta cifra decimale dei dati GPS, è possibile tenere conto delle transizioni degli utenti da una sottosezione ad un'altra. Così facendo è possibile realizzare sottosezioni quadrate di lato 10 metri. Tale valore risulta essere il minimo utilizzabile senza il pericolo di incorrere in errori di localizzazione: infatti il GPS dei dispositivi mobili risulta avere in modo consolidato una precisione alla quarta cifra e qualora fosse possibile andare oltre, la precisione della posizione espressa dalle ultime cifre risulta comunque poco attendibile. Inoltre la scelta di avere sottosezioni di lato 10 metri si è dimostrata anche ottimale in termini di processamento per la creazione del grafo, mostrando la migliore resa in termini di capacità di riconoscimento (sebbene sempre in modo approssimato) della topologia stradale.

B. *Routing dei dati*

Sfruttando quanto appena detto, anche la realizzazione del routing dei dati relativi ad una certa sezione, verso il corrispondente "ramo" della topologia storm non risulta essere troppo complesso. Il punto focale della questione è l'opportuna definizione dei *topic* in Kafka. L'idea di fondo è fare in modo che per ogni sezione della mappa cittadina venga definito univocamente un topic di Kafka: ciò viene realizzato sfruttando come topic-name le coordinate identificative della sezione, che, per quanto detto, si può intendere siano di facile ottenimento, implementando un semplice mapping uno a uno tra topic e sezione. Dato un arco è possibile risalire alla sezione a cui esso appartiene, e dunque al topic di inoltro, andando ad analizzare le prime cifre delle coordinate del punto sorgente (o per essere più conformi, della sottosezione atomica) da cui è generato; a questo punto è compito della componente kafkaSpout, la quale si registra al topic della specifica sezione, svolgere la funzione data injection per il "ramo" della topologia Storm a cui è connessa.

C. *Indicizzazione dei file e query in ElasticSearch*

Seguendo la linea guida usata per il topic name di Kafka, l'indicizzazione dei file per la componente si ElasticSearch risulta essere altrettanto intuitiva. infatti ogni file, contenendo i dati di una specifica strada è indicizzato con la chiave della strada stessa.

È possibile rispondere alle seguenti query:

- data una sezione, è possibile ricavare le strade che la compongono; questa ritornerà i documenti relativi ad ogni strada;
- data una strada, è possibile capire la sezione in cui si sviluppa, gli archi che la compongono e la velocità media percepita;
- dato un arco, è possibile reperire la strada di appartenenza con relativa velocità media;

V. Prestazioni del sistema

Per analizzare le presatazioni sono state esegute sperimentazioni tramite dei client implementati ad hoc, realizzati in modo tale da poter stressare sufficientemente la componente di stream processing del sistema e la componente di ElasticSearch. In particolare, i test sulla componente di processing sono stati esguiti sfruttando due modalità differenti di invio di dati. In entrambi i casi, l'intera sperimentazione è avvenuta tenendo attiva una componente ulteriore di ElasticSearch che effettua query in modo continuativo. L'infrastruttura fisica su cui è avvenuto il deploy dei componenti prevede l'utilizzo di istanze Ec2 di tipo *m4.large* per l'istanziazione delle due componenti Supervisor di Storm, della componente Kafka e Redis, mentre sono state utilizzate istanze *t2.micro* per le componenti Zookeeper e Nimbus relative a Storm, e per il cluster di tre nodi di ElasticSearch. In totale si ha un sistema comprensivo di 4 *large* e 5 *micro*.

A. *Test Effettuati*

Un primo test *Test 1* è stato eseguito sviluppando un client che cerca di inviare consecutivamente mille richieste alla componente di Storm ad intervalli di 0.2 secondi, ognuna delle quali distribuita in modo uniforme rispetto ai quattro settori implementati e consistente di una lista di quattro archi. La scelta della dimensione della richiesta non è casuale: infatti si ritiene che essa sia il picco che effettivamente un cliente possa inviare al secondo. Per spiegare meglio, è stato considerato come tetto massimo di velocità con cui un utente può muoversi in macchina un valore di 150 Km/h, pari a circa 41 m/s; ciò significa che in queste condizioni l'utente preso in considerazione non sarebbe in grado di inviare oltre quattro archi. Si vuole far notare che la condizione scelta per il test è dunque un plausibile scenario reale di massimo stress a livello di mole di dati sottoposti al sistema, e che in un caso puramente verosimile tali richieste sarebbero mediate con altre di dimensione minore. Per cercare di testare ulteriormente i limiti del sistema, è stato in realtà svolto una ulteriore sperimentazione seguendo le stesse modalità sopra spiegate, ma aumentando il numero di richieste consecutive a quattromila.

Un secondo test *Test 2* consiste nell'uso di cinque client paralleli che inviano ognuno 100 tuple consecutivamente senza nessuna pausa tra un invio ed una altro. Anche in questo caso i dati sono distribuiti in modo uniforme tra i diversi rami della topologia di storm. L'idea di fondo è di riprodurre un workload il più possibile realistico generando un insieme di richieste che giungono al sistema effettivamente in contemporanea, creando allo stesso tempo uno scenario meno "impulsivo" rispetto a quello riprodotto dal Test 1.

B. *Risultati dei Test*

Per quanto riguarda l'analisi della performance del sistema, sono state utilizzate le informazioni riportate dalla UI di storm. Nelle Tabelle 1 e 2 sono riportati i risulati ottenuti dal *Test 1*. Consumer è il bolt che riceve dati dallo spout e li suddivide; Mean calcola la media della velocità nel singolo arco usando il valore appena ricevuto inviando poi il totale a Street il quale fa

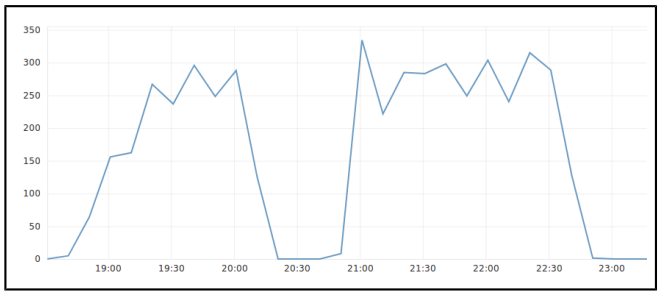

Fig. 4. Il grafico mostra il numero di scritture medie effettuate su ElasticSearch. L'asse delle x rappresenta l'orario con un unità pari a 10 minuti. Sull'asse delle y vi è il numero di scritture.

la stessa cosa a livello di strada; infine Producer riceve la velocità aggiornata per una data strada e modifica il relativo file in ElasticSearch. Analizzando gli output relativi al primo test, è possibile calcolare che il sistema ha mostrato nell'arco dei trenta minuti presi in ossrvazione un throughput medio del valore di circa 304 tuple/s che completano l'intero albero della topologia, con un tempo di latenza medio complessivo di 22.258 ms. Per tempo di latenza si intende il tempo che intercorre tra l'istante in cui una tupla è immessa nella topologia dalla componente Spout e l'istante in cui invece termina la propria esecuzione nell'ultimo livello di Bolts. Osservando invece i risulati ottenuti con la variante da quattromila richieste consecutive, è stato calcolato un aumento del throughput e della latenza del sitema, che raggiungono valori di circa 348 tuple/s e 57.595 ms. L'elevata latenza è dovuta principalmente all'ultimo Bolt che si interfaccia con Elastic Search per svolgere le mansioni di update sui dati, i cui tempi di esecuzione risultano essere preponderanti rispetto a quelli osservati negli altri Bolt come mostratio nelle tabelle stesse. Sfruttando inoltre gli strumenti di monitor offerti dal servizio di Elastic Search è possibile apprezzare l'andamento dei througput a tempo continuo del Test 1 e del Test 2 in Figura 4 e 5. È possibile notare, in particolare, come i due scenari mostrino un comportamento simile e come però il secondo test abbia prodotto un throuhput con meno varianza e più continuo, come era scopo dello stesso.

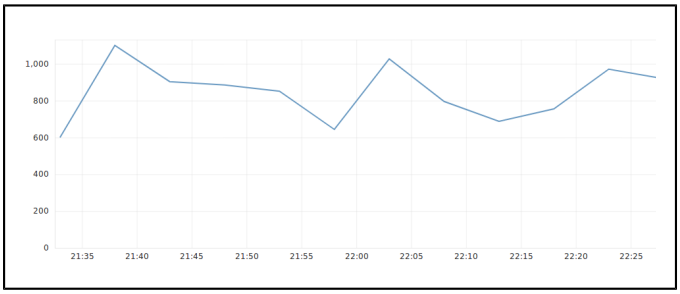

Fig. 5. Andamento della somma delle scritture eseguite su Elastic Search.

Bolt	**Statistiche**		
	Tuple emesse	***Execute Latency (ms)***	***Process Latency (ms)***
consumer	547160	1.408	1.400
mean	547860	0.205	0.210
street	416380	0.486	0.511
producer	0	15.570	14.924

Tab 1. La tabella mostra le tuple emesse, la latenza dovuta alla fase di esecuzione e per il processamento di ogni tipologia di bolt in un intervallo di 10 minuti..

Intervallo Temporale	***Emitted***	***Complete Latency (ms)***	***Ack***
10m 0s	2170240	13,261	181007
30m 0s	6576620	22,258	547580

Tab 2. La tabella mostra le tuple emesse, la latenza dovuta al procesamento delle tuple e il nuemero di tuple a cui è seguito un ack che quindi hanno completato l'intero albero di esecuzione. Il numero di tuple fallite non è mostrato poiché pari a 0 nella sessione di test presa in esame.

VI. Sviluppi futuri

L'applicazione risulta essere completa per quanto concerne la sua funzionalità di stima del traffico attuale sfruttando i dati relativi alle velocità, tuttavia possono ancora essere introdotte delle aggiunte. È possibile utilizzare i dati ricevuti per la costruzione di una base di dati in cui memorizzare oltre alla velocità media anche il numero di macchine per ogni strada in diversi istanti della giornata (ad esempio ogni 15 minuti). Con il database a disposizione è inoltre possibile produrre uno storico dei dati e dunque introdurre tramite machine learning meccanismi di anomaly detection per rilevare probabili incidenti o lavori in corso nelle strade della città riscontrando variazioni non consuete nei dati. Infine, dato il forte disaccoppiamento del sistema, è possibile fare in modo di aggiungere nuove funzionalità relative al processamento di dati di nuova natura, provenienti non solo da device mobili cellulari, ma anche da sensori, magari anche interni alle automobili, come è plausibile pensare dato il trend crescente delle tecnologie IoT.

VII. Guida all'installazione

È possibile eseguire l'applicazione tramite jar; tuttavia è necessario configurare prima le diverse componenti che appartengono al sistema, ovvero i nodi Kafka, Storm, Redis ed ElasticSearch. Una volta istanziati, basta semplicemente prendere nota delle informazioni seguenti e fornirle ad un file chiamato "Application.properties" da salvare nella stessa cartella da cui si esegue il jar:

- indirizzo del nodo di Zookeeper da contattare per inviare dati alla componente di Kafka
- indirizzo di tre nodi (il minimo necessario per avere un cluster di Redis) che appartengono al cluster di Redis e relativi numeri di porta di ascolto; se il cluster è composto da più di tre nodi basta comunque comunicarne tre: il client Redisson usato per

interfacciarsi con Redis è in grado, dati tre nodi, di conoscere tutti i restanti non esplicitamenbte dichiarati.

- indirizzo del nodo di ElasticSearch da contattare per accedere al cluster.

A questo punto, è necessario eseguire il jar direttamente dalla componente Nimbus di Storm, che è stata istanziata in principio, tramite il comando *storm jar <path_to-jar> <MainClass_path>* per avviare la topologia e rendere attivo il sistema.

REFERENCES

[1] www.aws.amazon.com

[2] www.kafka.apache.org/

[3] www.storm.apache.org

[4] OpenFog consortium, “OpenFog Reference Architecture for Fog Computing”