Sistemi Distribuiti e Cloud Computing - A.A. 2017/18 Progetto 1

A Urban Analytics System for Traffic Monitoring and Control

Bottini Stefano *Catullo Gentilcore Maria José* *Milani Alfredo* *Tuni Gabriele*

Introduzione

In tale articolo si propone una soluzione progettuale per la sfida proposta nell'ambito del contest CINI *Smart City University Challenge 2018* riguardante la realizzazione di un sistema distribuito per il monitoraggio e il controllo del traffico cittadino. Scopo di tale sistema è quello di rilevare in tempo reale le condizioni della viabilità urbana e attuare in base alle informazioni rilevate delle politiche di ottimizzazione della stessa. In particolare il sistema va ad agire sulla temporizzazione dei semafori, regolando la durata della luce verde in base alla congestione stradale e garantendo uno smaltimento il più possibile ottimo di essa. La sfida proposta è quella di gestire il grande flusso di dati proveniente dai diversi tipi di sensori previsti in modo efficiente ed efficace e di soddisfare tutti i requisiti richiesti. Si ricorda infine che la soluzione proposta non si basa su alcuna città esistente e dunque i dati dei sensori utilizzati, generati appositamente, possono non rispecchiare a pieno una situazione reale di viabilità cittadina.

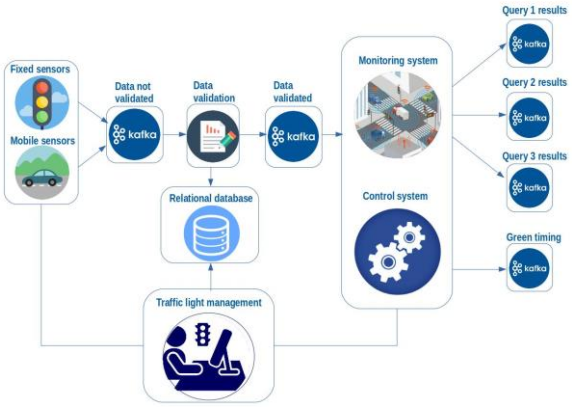
Abstract

La soluzione proposta è stata progettata per essere eseguita in un ambiente distribuito; ogni nodo può essere replicato in modo da garantire il grado di parallelismo o robustezza necessario per processare grandi quantità di dati. A tal scopo ci si è avvalsi, nella realizzazione dell'architettura, della piattaforma *Apache Storm* per *data stream processing*. Per il soddisfacimento dei requisiti richiesti, l'implementazione proposta prevede per la prima query il calcolo e l'aggiornamento in tempo reale di una classifica delle 10 intersezioni cittadine più pericolose in quanto maggiormente congestionate, sulla base di tre diverse finestre temporali. A tal scopo e nel resto del progetto sono state implementate e gestite le finestre temporali sfruttando le *tick tuple*, messe a disposizione da *Storm*. Per la seconda query, che ha lo scopo di individuare le intersezioni maggiormente congestionate, confrontando la mediana del numero di veicoli per singola intersezione con la mediana globale, sempre sulla base dei dati raccolti nelle tre diverse finestre, è stato utilizzato un meccanismo analogo al primo e si è sfruttata la libreria *TDigest* per il calcolo della mediana. La terza query che prevede l'individuazione delle sequenze di semafori critiche, ossia maggiormente trafficate, e per la quale è stato definito un indice di congestione legato al numero di veicoli e alle velocità degli stessi, fa riferimento ai dati raccolti negli ultimi 5 minuti e riporta la sequenza che risulta più critica in tale intervallo di tempo. Il sistema inoltre monitora lo stato delle lampade di ogni semaforo distinguendo tra i possibili livelli di errore e segnalando quando una di queste è guasta e il tipo di errore verificatosi. Viene inoltre regolata la durata del verde sulla base dei flussi di autovetture in arrivo ai diversi accessi delle varie intersezioni al fine di garantire un deflusso ottimo del traffico; a tal scopo si è applicato il noto metodo di *Weibster* per la determinazione dei parametri semaforici ottimi. Infine il sistema prevede un'interfaccia che permette ad un amministratore di autenticarsi e gestire i semafori all'interno della città fornendo delle API Rest per l'inserimento, la cancellazione e la visualizzazione dei semafori. Un'ulteriore funzionalità per l'autenticazione dei semafori stessi prevede che i dati che arrivano siano validati come provenienti da semafori registrati nel sistema e aventi i campi richiesti; una volta validati, i dati vengono salvati in una cache, al fine di ottimizzare le prestazioni.

Overview del Sistema

L'architettura di alto livello del sistema si compone di diverse parti che si interfacciano tra loro. I dati dei sensori posti sui semafori (sensori fissi) così come quelli posti all'interno delle autovetture (sensori mobili) vengono pubblicati su *Kafka*. Il sistema di monitoraggio, responsabile dell'individuazione delle situazioni critiche di viabilità prima citate, così come il sistema di controllo, responsabile della

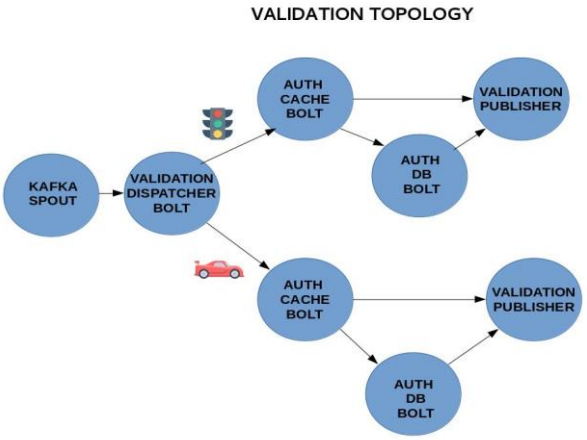
temporizzazione delle luci dei semafori, si interfacciano con *Kafka* sia per il prelievo dei dati che per la pubblicazione dei risultati. I dati pubblicati vengono resi disponibili per eventuali attuatori che intervengano sul sistema stesso per le modifiche. Infine vi è un ulteriore componente che si interfaccia con l'insieme dei sensori visto, che è quello di validazione e gestione dei semafori. L'architettura generale e ad alto livello del sistema è illustrata nell'immagine seguente.



Sistema di Monitoraggio & Sistema di Controllo

Validazione

Tra le principali funzioni del nostro sistema di data stream processing rientra la validazione delle tuple. Lo scopo di questa funzionalità è quello di scartare le tuple non valide, ad esempio quelle derivanti da sensori guasti o non autorizzati. Le tuple vengono validate dal punto di vista sintattico, ossia per essere accettate devono presentare determinati campi, corrispondenti alla specifica fornita.
La relativa topologia utilizza un meccanismo di caching per ridurre l'overhead sul database utilizzato per il riconoscimento dei sensori registrati nel sistema. L'id di una tupla precedentemente validata viene inserito in cache, permettendo alle tuple successive, aventi lo stesso id, di essere validate più velocemente. Le tuple validate vengono pubblicate in due *topic kafka*, diversi a seconda che si tratti di una tupla generata da un sensore mobile o di una tupla generata da un sensore fisso. In questo modo le altre topologie presenti nel sistema potranno leggere direttamente le tuple validate sui *topic kafka* citati (effettuando comunque un controllo minimo).



La topologia di validazione delle tuple è composta da 5 elementi fondamentali:

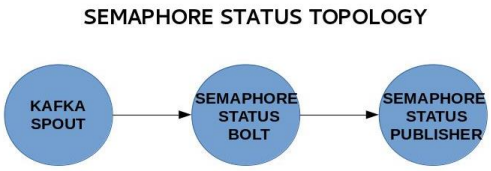
- ***Spout***: legge i dati da un *topic kafka,* le cui tuple (provenienti da sensori mobili e fissi indistintamente) non sono state ancora validate, e li invia al *dispacher bolt* ;
- ***Dispatcher bolt:*** verifica innanzitutto che la tupla sia completa di tutti i campi necessari e in caso di esito positivo distribuisce i dati al giusto *authenticator cache bolt* in base alla tipologia di sensore di provenienza della tupla;
- ***Authenticator cache bolt:*** controlla se la nuova tupla è già presente in cache; in tal caso vuol dire che è stata validata e viene quindi inviata direttamente al *publisher bolt*. Questo meccanismo viene utilizzato sia per i sensori fissi che per quelli mobili;
- ***Authenticator endpoint bolt***: controlla che la stessa sia presente nell'*end-point*, la inserisce in cache e la invia al *publisher*. Questa tipologia di bolt viene usata sia per i sensori fissi che per quelli mobili;
- ***Publisher bolt:*** pubblica le tuple validate su appositi *topic kakfa*.

Stato dei semafori

Un'altra funzionalità del sistema di monitoraggio riguarda il controllo dello stato dei semafori. Attraverso quest'ultima viene monitorato il funzionamento delle tre lampade di ogni semaforo e, in caso di malfunzionamento di una di queste, viene generato un allarme e vengono fornite informazioni

sulla posizione del semaforo interessato e sulla lampada guasta.
A tal proposito si è pensato di individuare il grado di malfunzionamento della lampada all'interno di un range di valori da 0 a 255. Questo intervallo di valori viene ulteriormente suddiviso in "sotto range" il cui numero totale indica il numero di livelli possibili di malfunzionamento previsti dal sistema. Per ottenere una granularità più fine sul livello di errore è possibile dunque aumentare il numero di "sotto range" previsti e specializzare maggiormente i tipi di malfunzionamento.
Nell'implementazione fornita vengono individuati tre livelli di malfunzionamenti possibili della lampada:

1. *funzionante* – non necessita né di manutenzione, né di sostituzione;
2. *difettosa* – necessita di manutenzione;
3. *guasta* – necessita di sostituzione.



La topologia che monitora lo stato dei semafori è costituita dai seguenti elementi:

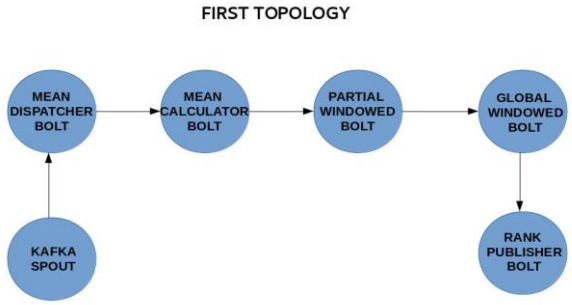
- ***Spout***: legge i dati da un *topic kafka* con le tuple validate provenienti da sensori fissi e li invia al *semphore status bolt;*
- ***Semaphore status bolt***: controlla se la tupla inviata dal sensore segnala il guasto di almeno una lampada del semaforo e in questo caso invia al *publisher bolt* le informazioni relative;
- ***Publisher bolt***: pubblica le informazioni ricevute su un apposito *topic kafka.*

Prima Query

La prima query che il nostro sistema di monitoraggio deve gestire riguarda la classifica delle 10 intersezioni aventi la maggiore velocità media di attraversamento. Viene richiesto di aggiornare in tempo reale tale classifica (ogni 4 secondi). Si richiede, inoltre, che la classifica venga calcolata secondo tre finestre temporali: 15 minuti, 1 ora e 24 ore.

Al fine di effettuare la classifica sulla base dei dati contenuti entro tre diverse finestre temporali è stato necessario implementare e gestire le stesse. Allo scopo sono state utilizzate le *Tick Tuple* messe a disposizione da *Storm* per scandire il tempo. A partire da queste è stata implementata una finestra temporale e sono stati gestiti tre tipi di eventi ad essa connessi:

1. *current*: eventi che si trovano nella finestra corrente;
2. *expired*: eventi che sono usciti dalla finestra corrente (scaduti);
3. *new*: eventi che sono stati generati a partire dall'ultima *Tick Tuple* ricevuta.



La topologia della prima query è composta da 6 elementi fondamentali:

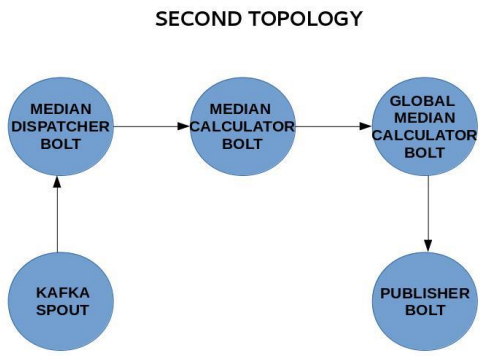
- ***Spout***: legge i dati da un *topic kafk*a con le tuple validate provenienti da sensori fissi e le invia al *dispacher bolt*;
- ***Dispatcher bolt***: invia le tuple al *mean calculator bolt* dopo averle incapsulate in un formato più idoneo per la gestione di dati provenienti da sensori fissi (a tale scopo si usa la classe *rich semaphore sensor*);
- ***Mean calculator bolt***: gestisce le informazioni secondo gruppi di intersezioni (*field grouping*) e calcola la media delle velocità relative ad un'intersezione solo se ha ricevuto 4 valori (cioè provenienti da tutti i semafori dell'incrocio); in questo caso invia le medie al *partial ranking bolt*. Per il calcolo della media il bolt usa una coda in cui salva i dati in base all'identificativo dell'incrocio a cui si riferiscono. Infine rimuove i dati relativi ad un

incrocio per il quale non è stato possibile calcolare la media all'interno della finestra temporale, in seguito di una mancata ricezione delle informazioni necessarie. Inoltre, se prima di poter calcolare la media di un'intersezione riceve due valori dallo stesso sensore allora scarterà l'intero incrocio presupponendo che siano dati obsoleti e che nuovi dati riguardo quella particolare intersezione saranno presto ricevuti;

- ***Partial ranking bolt***: si occupa di computare classifiche parziali aggiornandole con i dati arrivati nell'ultima finestra temporale e scartando quelli scaduti. Una volta ottenuta, la classifica parziale, viene inoltrata al *global ranking bolt*;
- ***Global ranking bolt***: è analogo al precedente ma opera su classifiche, anziché direttamente sui dati, fondendo le classifiche parziali e ordinando il risultato finale che viene inviato solo se diverso da quello precedentemente emesso;
- ***Rank publisher bolt***: pubblica le classifiche finali su un apposito *topic kafka*.

Seconda Query

La seconda query del sistema di monitoraggio riguarda l'identificazione delle intersezioni a maggior congestione. Queste ultime equivalgono alle intersezioni la cui mediana del numero di veicoli che le attraversano è superiore alla mediana globale del numero di veicoli che hanno attraversato tutte le intersezioni; anche in questo caso vanno considerate finestre temporali di 15 minuti, 1 ora e 24 ore.



La topologia della seconda query è composta da 5 elementi fondamentali:

- ***Spout:*** legge i dati da un *topic kafka* con le tuple validate provenienti da sensori fissi e li invia al *dispacher bolt;*
- ***Dispatcher bolt***: invia le tuple al *median calculator bolt* dopo averle incapsulate in un formato più idoneo per la gestione di dati provenienti da sensori fissi (classe *rich semaphore sensor*);
- ***Median calculator bolt:*** calcola la mediana delle intersezioni con lo stesso meccanismo del *mean calcultor bolt*. Le mediane delle singole intersezioni così calcolate saranno inviate al *global median calculator bolt;*
- ***Global median calculator bolt***: riceve dati da due stream, uno proveniente dai sensori fissi, l'altro risultante dal calcolo delle mediane delle intersezioni. Le operazioni principali svolte da questo bolt sono:
  - calcolare la mediana globale del numero di veicoli a partire dai dati provenienti dai sensori, aggiornandola in tempo reale;
  - confrontare le mediane delle intersezioni con la mediana globale calcolata e inviare al *publisher bolt* le intersezioni congestionate;
- ***Publisher bolt***: pubblica tutte le intersezioni ricevute su un apposito *topic kafka*.

Terza query

La terza query del sistema di monitoraggio progettato riguarda l'identificazione della sequenza di semafori che negli ultimi 5 minuti risulta maggiormente congestionata ossia caratterizzata da un alto numero di utenti che si muovono a bassa velocità. La query è stata realizzata sfruttando il flusso di dati proveniente dai sensori mobili.

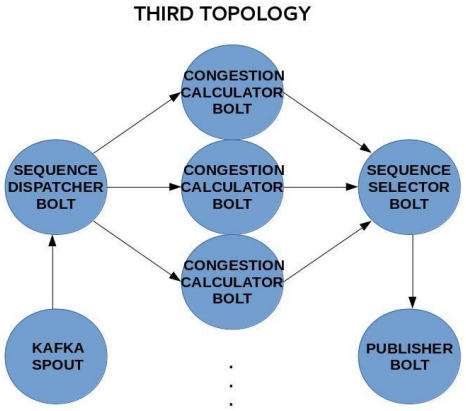
L'idea principale è stata quella di calcolare un indicatore di congestione della sequenza di semafori basandosi sulle velocità dei veicoli in transito su di essa e sul numero di veicoli che la percorrono. Nello specifico si è pensato di individuare dei range di velocità, attribuendo dei pesi a ciascuno di essi, con un peso maggiore per i range contenenti velocità più basse. L'indicatore di congestione di una sequenza viene dunque calcolato come media pesata, ottenuta

dalla somma del numero di veicoli appartenenti ad una determinata fascia di velocità moltiplicato per il peso attribuito a quella fascia. Formalmente si ha:

$$c = \sum_{i=1}^{m} n_i * p_i$$

c = grado di congestione
m = numero totale di fasce di velocità
n_i = numero di veicoli le cui velocità rientrano nella fascia i-esima
p_i = peso associato alla fascia i-esima

Nell'implementazione proposta si è deciso di individuare 10 fasce di velocità, considerando tale numero sufficiente affinché il calcolo dell'indice non risenta troppo del fatto che essendo la media un indicatore di centralità non tiene conto degli eventuali valori estremi (outliers) e dunque può produrre un risultato falsato. Aumentando la granularità delle fasce e redistribuendo i lori pesi si può arginare questo effetto.
Per la progettazione della query abbiamo fatto ricorso ad alcune semplificazioni e assunzioni. Innanzitutto le sequenze di semafori considerate non variano a runtime e vengono lette da un file di configurazione. Inoltre, la struttura complessiva che si delinea a partire da queste sequenze rispecchia una rete a griglia tipica di città come Manhattan. Su questa griglia sono state considerate solo sequenze longitudinali e latitudinali. Per individuare i veicoli in transito su di esse sono state utilizzate latitudine e longitudine, provenienti dai dati dei sensori mobili, confrontandole con le coordinate delle sequenze, considerate con un certo margine di tolleranza. Un'ulteriore assunzione è che le linee della griglia ricalchino meridiani e paralleli cosi da individuare facilmente la direzione di movimento di una vettura su una sequenza: quando un autoveicolo si muove lungo un tratto su una delle sequenze individuate, i dati da esso inviati saranno caratterizzati, al variare del tempo, da una variazione ridotta di una delle due coordinate e da una variazione consistente dell'altra.



La topologia della terza query è composta da 5 elementi fondamentali:

- ***Spout***: legge i dati da un *topic kafka* con le tuple validate provenienti da sensori mobili e li invia al *dispacher bolt*;
- ***Dispatcher bolt***: dopo aver incapsulato le tuple in un formato più idoneo per la gestione di dati provenienti da sensori mobili (classe *rich mobile sensor*), invia i dati al corrispondente *congestional computional bolt* associato alla sequenza corretta su cui il sensore mobile si sta muovendo;
- ***Congestional computional bolt***: raccoglie i dati dei sensori che transitano sulla sequenza da esso gestita e calcola l'indice di congestione relativo ad essa; vi è infatti un bolt di questo tipo per ognuna delle sequenze gestite dal sistema. Infine invia l'indice calcolato al *sequence selector bolt;*
- ***Sequence selector bolt***: raccoglie tutte le sequenze di semafori e i relativi indicatori di congestione, le ordina in base a questi ultimi e invia la sequenza con l'indice maggiore (ovvero quella con maggiore congestione) al bolt successivo. Invia nuovamente la stessa sequenza anche se cambia solo il suo grado di congestione;
- ***Publisher bolt***: pubblica la sequenza ricevuta su un apposito *topic kafka.*

Green temporization query

Il sistema di controllo fornisce la funzionalità di adattare dinamicamente la durata della luce del verde alle condizioni del traffico cittadino, al fine di ottimizzare la viabilità. E' stato assunto che ogni intersezione venga regolata da un ciclo semaforico a 2 fasi; la durata complessiva del ciclo è fissa e pari a 200 sec; di questi anche la durata della luce gialla è fissa e risulta essere di 4 sec. Di conseguenza è possibile intervenire esclusivamente sulla durata della luce verde (e di conseguenza della luce rossa). Per stimare la durata ottima della luce verde, per una singola intersezione e per ogni fase del ciclo semaforico, è stato seguito il metodo di *Weibster*; in particolare la formula seguente fornisce la durata ottima del verde effettivo per la fase *i* (intervallo di tempo fittizio equivalente al tempo reale di scorrimento del traffico nell'intersezione e comprendente il tempo di verde):

$$V_{E_i} = \frac{q_i/S_i}{\sum_{i=1}^{n} q_i/S_i} \; (C - \sum_{i=1}^{n} l_i)$$

V_{Ei} = durata verde effettivo nella fase i dell'intersezione considerata
q_i/S_i = rapporto tra flusso in arrivo e portata di saturazione dell'accesso più rappresentativo (con flusso maggiore) della fase i
l_i = tempo perso alla partenza nella la fase i
C= durata complessiva del ciclo semaforico
n= numero di fasi del ciclo semaforico per l'intersezione considerata

A partire da questa formula è possibile calcolare la durata ottima della luce verde per la fase *i*:

$$V_i = V_{E_i} + l_i + I_i$$

V_i = durata verde ottimo nella fase i dell'intersezione considerata
I_i = durata intervallo di cambio nella fase i del ciclo semaforico, dato da $I_i = \max[I_{ai}, I_{pi}]$
V_{Ei} = durata verde effettivo nella fase i dell'intersezione considerata
l_i = tempo perso alla partenza nella la fase i

Nelle formule sovraesposte sono state fatte alcune assunzioni e semplificazioni. Nel caso in esame *n* (il numero di fasi) è pari a 2, come anticipato. Ci siamo posti nell' ipotesi di larghezza dell'accesso inferiore a 5.5 m, per cui la portata di saturazione S per entrambe le fasi può essere considerata fissa; in particolare è stata ipotizzata una larghezza dell'accesso di 3.05 m per cui la corrispondente portata di saturazione risulta essere 1850 veic/h di verde. Nel calcolo del verde effettivo la quantità al numeratore q_i/S_i è quella relativa all'accesso più rappresentativo della fase *i*. Essendo dunque S_i fissa per ogni accesso e per ogni fase il numeratore sarà relativo all'accesso nella fase i con flusso di arrivo q_i maggiore. Il flusso per ognuno dei quattro accessi delle diverse fasi viene calcolato a partire dai dati dei sensori fissi con la seguente formula:

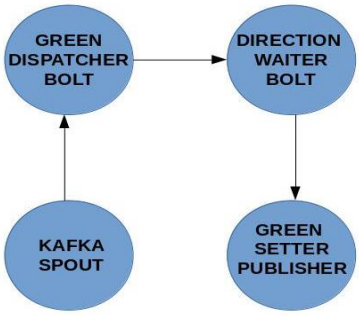
$$q_{ji} = \frac{n^\circ \; veicoli}{freq \; emissione \; tuple \; (sec)}$$

dove q_{ji} rappresenta il flusso all'accesso j per la fase i del ciclo semaforico e viene convertito in veic/sec. La durata totale del ciclo C è pari a 200 sec; infine l_i indica il tempo perso dalla corrente rappresentativa nella fase *i* e nel caso in esame tale valore è stato fissato pari a 4 sec (valore tipicamente usato).

Un'ultima assunzione riguarda il calcolo della durata del verde ottimo. La quantità I_i denota l'intervallo di cambio della fase *i* e nel caso di intersezioni semaforizzate in strade urbane, in cui le velocità siano comunque contenute, risulta essere pari ad I_{p_i}, intervallo necessario ad un veicolo che si avvicina all'intersezione, ad attraversarla. Valori tipici per tale parametro risultano essere intorno a qualche unità; nel caso in esame tale parametro è stato fissato a 7 sec.

Una limitazione riscontrata in tal proposito è stata che la durata del verde delle due fasi opposte di una stessa intersezione risulta non essere pari alla durata totale del ciclo. Questo può essere imputato al fatto che i flussi delle correnti ai diversi accessi di una stessa intersezione vengono calcolati a partire da un dato proveniente dai sensori fissi (il numero di veicoli), generato in modo casuale e che non tiene conto della relazione dovuta proprio alla regolazione semaforica che intercorre tra tali valori nei dati di una stessa intersezione.

GREEN TIMING TOPOLOGY



La topologia della query di temporizzazione del verde è composta da 4 elementi fondamentali:

- ***Spout:*** legge i dati da un *topic kafka* con le tuple validate e li invia al *dispacher bolt;*
- ***Dispatcher bolt***: invia le tuple al *direction waiter bolt* in modo che un bolt che ha già gestito i dati relativi ad un incrocio riceverà anche tutti gli altri dati che si riferiscono allo stesso incrocio (*field grouping*);
- ***Direction waiter bolt:*** invia i dati al *green setter publisher bolt* solo se ha ricevuto i dati relativi ad almeno una delle due coppie di accessi appartenenti alla stessa fase;
- ***Green setter publisher bolt***: calcola le durate del verde delle fasi dell'intersezione per le coppie di accessi ricevute, secondo la metodologia già esposta e pubblica tali valori su un apposito *topic kafka.*

Sistema di Gestione dei Semafori

Un ulteriore componente del nostro sistema concerne la parte di gestione dei semafori da parte di un utente autorizzato e registrato nel sistema. A tal scopo è stato fatto uso di un database relazionale MySql per il mantenimento delle informazioni riguardanti utenti e semafori. E'prevista una parte di front end che fornisce un' interfaccia grafica per l'inserimento di un nuovo semaforo, la rimozione di un semaforo esistente e la visualizzazione di uno o tutti i semafori presenti nel sistema; è previsto anche l'aggiornamento di alcune informazioni riguardanti il semaforo. Tutto ciò è stato realizzato tramite API Rest all'endpoint, dietro cui vi è il db citato. Le informazioni nel database vengono sfruttate anche per la validazione dei dati ricevuti.

PIATTAFORME DI SVILUPPO

Il sistema è stato progettato e sviluppato grazie all'ausilio dell'IDE *Intellij IDEA*, quest'ultimo con il supporto offerto dal tool *Maven* ha permesso di utilizzare agevolmente *Apache Storm* all'interno dell'IDE stesso. Sono stati utilizzati, come già descritto, il sistema *publish-subscribe Apache Kafka*. Sia *Storm* che *Kafka* si basano su *Zookeeper* per mantere informazioni di stato dei rispettivi cluster. Per la validazione e per la gestione dei semafori e dei loro sensori (operazioni *CRUD*) sono stati utilizzati un database relazionale *MYSQL* e il framework *Spring*, posto sopra di esso così da esporne i principali servizi come *API REST*.

Infine si è fatto ricorso a *InfluxDB* al fine di poter utilizzare *Grafana* per la presentazione grafica dell'output.
Per quanto riguarda il *deployment* su cloud è stato utilizzato il servizio *EC2* di *Amazon AWS.*
I componenti del sistema sono stati divisi nel modo seguente:

- nodo *Zookeeper* (single node)
- nodo *Apache Kafka* (1 nodo, 1 broker)
- nodo *Nimbus* (storm)
- nodo *Supervisor* (storm)
- nodo *back-end* (Spring e MySQL)
- nodo *monitoring-system*
- nodo *simulator*

Tutte le istanze sono state poste su una stessa *VPC* e dotate di un *public IP* al fine di garantirne la comunicazione.

PRESTAZIONI

Sono state valutate le prestazioni del sistema sia in termini di latenza che di throughput, per ciascuna delle topologie precedentemente esposte. Per entrambi i valori sono stati utilizzati i dati riassuntivi forniti da *Storm UI* riguardanti le diverse topologie sottomesse. In particolare per il calcolo della latenza sono stati considerati i valori di *process latency* e *execute latency*. Tra questi, il valore più appropriato risulta essere quello dell'*execute latency*, in quanto, per come questa viene definito nella UI stessa, risulta essere il valore effettivo del tempo speso da ogni bolt nella fase di *execute*. La *process latency* considera l'intervallo di tempo trascorso da quando una tupla viene ricevuta a quando ne viene fatto l'ACK; siccome l'ACK può essere inviato prima che la tupla venga processata del tutto, tale valore risulta essere meno adeguato per il calcolo della latenza effettiva.

Un' ultima considerazione riguarda il fatto che la latenza dell'intera topologia può essere ricavata come somma delle latenze dei singoli bolt.

$$Lat = \sum_{i=0}^{n} LatE_i$$

n = numero nodi della topologia

$LatE_i$= latenza di esecuzione mediata sul numero di tuple processate finora del nodo i

Lat= latenza di esecuzione media della topologia

Infine precisiamo che per le queries che prevedono finestre temporali da 15 min, 1h e 24h, poichè una stessa tupla in ingresso esce tre volte dalla topologia seguendo le diverse sequenze di bolt (una per ogni

finestra), viene fatta una media aritmetica tra le tre latenze ottenute per ogni finestra.

Per quanto riguarda il *throughput*, si è considerato che, in una rete di nodi, quest'ultimo è limitato dal nodo più lento (*bottleneck*) ed è stato notato che, per ogni topologia, il nodo più lento è sempre quello posto prima del publisher in quanto è l'unico che ha una frequenza di emissione fissa e non processa le tuple appena le riceve. Considerato ciò, è stato quindi utilizzato il valore *transferred*, pari al numero di tuple trasferite per una certa topologia. Rispetto all'altro valore *emitted* messo a disposizione da *Sorm UI*, il precedente risulta più appropriato in quanto conta le tuple effettivamente trasferite da un bolt ad un altro. Ad esempio, nel caso del *global grouping,* una stessa tupla viene trasferita a più bolt, *emitted* considera tale tupla una sola volta, mentre *transferred* la considera una volta per ogni bolt che la riceve in *global grouping*. Infine per ottenere il valore del throughput, essendo questo definito come frequenza di output di una rete, è stato considerato un certo intervallo di tempo ed è stato fatto il rapporto tra il numero di tuple trasferite in tale intervallo e la durata dell'intervallo stesso in minuti, ottenendo così il numero medio di tuple trasferite in un minuto. Tale tasso di uscita risulta essere quello del nodo più lento (purchè ci si trovi in una condizione di regime).

$$THR = \frac{transferred_{btnk}}{time[min]}$$

$transferred_{btnk}$ = numero di tuple trasferite dal nodo che costituisce il bottleneck in un certo intervallo temporale
$time[min]$ = ampiezza in minuti dell'intervallo considerato

Si forniscono dunque i valori, calcolati nel modo appena esposto, di latenza e throughput per le diverse topologie. Tali valori sono stati ottenuti eseguendo il sistema in modalità cluster, in locale. L'intervallo di tempo considerato, in cui si suppone che il sistema sia andato a regime è pari a circa $time\,[min] = 15$

- FIRST TOPOLOGY

$$THR = 30498\,\frac{tuple}{min}$$

$$Lat = 0.80\,ms$$

- SECOND TOPOLOGY

$$THR = 33796\,\frac{tuple}{min}$$

$$Lat = 0.53\,ms$$

- THIRD TOPOLOGY

$$THR = 2721\,\frac{tuple}{min}$$

$$Lat = 88\,ms$$

- GREEN TIMING TOPOLOGY

$$THR = 44472\,\frac{tuple}{min}$$

$$Lat = 0.291\,ms$$

- SEM. STATUS TOPOLOGY

$$THR = 32789\,\frac{tuple}{min}$$

$$Lat = 0.3\,ms$$

- VALIDATION TOPOLOGY

$$THR_{SEM} = 17716\,\frac{tuple}{min}$$

$$Lat_{SEM} = 18.83\ ms$$

$$THR_{MOB} = 34614\,\frac{tuple}{min}$$

$$Lat_{MOB} = 26.75\ ms$$

Di seguito riportiamo alcune considerazioni interessanti riguardo i risultati sovraesposti. Notiamo che per la seconda topologia, se tutte le tuple validate venissero pubblicate su kafka (perché soddisfacenti i criteri della query), il numero di tuple in uscita da essa al minuto sarebbe il triplo (tre finestre) di quello della *validation topology*. Ciò risulta non vero in quanto non è plausibile che tutte le tuple rispettino il criterio riguardante la mediana per essere pubblicate. Riguardo le prestazioni della *validation topology*, gli indici sono stati calcolati separatamente per i sensori fissi e quelli mobili. Questo poiché una determinata tupla attraversa in modo esclusivo uno dei due rami della topologia, a seconda che sia generata dai sensori mobili o da quelli fissi. Inoltre la latenza di questa topologia risulta essere influenzata dall'accesso al database, ma viene comunque ammortizzata dalla presenza della cache. La latenza calcolata per i bolt relativi ai sensori mobili risulta essere, come atteso,

superiore a quella calcolata relativamente ai sensori fissi, data la maggior frequenza di emissione dei primi. Per quanto riguarda la topologia che controlla lo stato dei semafori il valore della frequenza di uscita risulta essere paragonabile a quelli delle altre topologie. Questo significa che il numero di lampade guaste è molto elevato; tale comportamento può essere interpretato come una conseguenza di come è stato progettato il simulatore e del fatto che esistono diversi livelli di malfunzionamento delle lampade.

Va precisato, infine, che la frequenza di emissione della terza topologia può aumentare notevolmente qualora si consideri un numero di sequenze di semafori maggiore di quello predefinito, così come incrementando il numero di sensori mobili. Per il calcolo dei valori esposti il numero di tali sensori è stato settato a 400.

CONFIGURAZIONE E ESECUZIONE

Per configurare ed eseguire correttamente il sistema di monitoraggio e controllo, i componenti essenziali del sistema che bisogna avviare sono: *monitoring system*, *simulatore* e *back-end*. Prima di tutto occorre collegarsi al seguente URL

https://github.com/alfredo-milani/trafficcontrol

e scaricare il progetto. Dopo aver configurato opportunamente *Zookeeper*, *Kafka*, *Storm* (*Nimbus* e *Supervisor*), a seconda che si voglia far eseguire il sistema in modalità locale o cluster, occorre accedere alla cartella contenente i file jar necessari *(jar-monitoring- control-system*).

Simulatore

Per avviare il simulatore occorre modificare il file di configurazione *config.properties,* posto nella cartella contenente tutti i jar, settando opportunamente l'indirizzo IP e la porta di ascolto di *Kafka*. Per far partire i sensori, bisogna eseguire i comandi seguenti:

java -jar [path_sensors-simulator.jar] -c [path_config.properties] -st [mobile/semaphore]

L'opzione *-st* permette di scegliere il tipo di sensore da simulare; inoltre è possibile settare con *-n* il numero di sensori desiderati e con *-w* il tempo di attesa in secondi prima che i sensori inizino a generare dati.

Back-end

Per avviare il back- end è necessario installare *MYSQL* e creare un database.
A questo punto occorre accedere al file *application.properties* (collocato in Boot-INF/classes), dentro il jar relativo (*admin-traffic-control*), ed inserire l'URL del proprio datasource, precedentemente creato, e le relative credenziali di accesso. Dopo aver aggiornato il file jar è possibile avviare il back-end tramite il seguente comando:

java -jar [path_admin-traffic-control.jar]

Monitoring System

Per avviare il Monitoring System bisogna modificare il file di configurazione *config.properties* contenuto nel jar relativo (*monitoring-system.jar*) e aggiornare il jar stesso.

In tale file occorre innanzitutto occorre configurare il parametro *mode* con il valore *local* o il valore *cluster* e specificare IP e porta di ascolto di *Kafka*. Qualora si voglia sfruttare il database per l'autenticazione dei sensori bisogna inoltre modificare negli URL degli endpoint "*semaphore sensors endpoint*" e "*mobile sensors endpoint*" indirizzo IP e porta di ascolto del *back-end* (la porta di default è 8200). Qualora non si voglia sfruttare il database per autenticare i sensori è possibile settare nello stesso file il parametro *debug_level*, ponendolo pari a 5.
Infine, è necessario settare opportunamente il parametro *sequences_semaphores_file* inserendo il path del file json contenente le sequenze predefinite di semafori necessarie nella terza query. Tale file è inizialmente posizionato all'interno della stessa cartella contenente i diversi jar. Infine per far partire il sistema, occorre eseguire il seguente comando:

[storm_path]/bin/storm jar [path_monithoring-system.jar] it.uniroma2.sdcc.trafficcontrol.TopologyStarter