JASMINE

Just A Simple Modular Intelligent Network Environment

Tiziano Ditoma
0252737
University of Rome – Tor Vergata
Computer Engineering, Master Degree
tiziano.ditoma@students.uniroma2.eu

Francesco Ottaviano
0259193
University of Rome – Tor Vergata
Computer Engineering, Master Degree
francesco.ottaviano@students.uniroma2.eu

Simone Mancini
0259568
University of Rome – Tor Vergata
Computer Engineering, Master Degree
simone.mancini@students.uniroma2.eu

Andrea Silvi
0258596
University of Rome – Tor Vergata
Computer Engineering, Master Degree
andrea.silvi@students.uniroma2.eu

Abstract

JASMINE is an open source project born for CINI 2018 Challenge on smart cities. The aim is to provide optimization for traffic management by adjusting traffic lights lamps' duration according to real-time traffic data coming from several sensors placed in proximity of traffic lights and from users' mobile devices. It is primarily based on Apache Software Foundation open-source solutions such as Apache Flink, for real-time data stream processing, and Apache Kafka as message broker. Furthermore a user-friendly interface allows to insert new elements in the system (e.g. traffic lights) and to analyze current traffic status. JASMINE uses database solutions such as MongoDB and Redis to efficiently store some relevant data.

Keywords*: Apache Flink, Apache Kafka, Redis, MongoDB, Data stream processing, publish/subscribe, sensors networks, smart cities, distributed system, cloud computing.*

I. Introduction

The purpose of the project is to develop an efficient and distributed solution to monitor and control real-time traffic status. Sensors placed near traffic lights send tuples containing relevant information to the system every 60 seconds and users' mobile devices provide location and user's speed data every 10 seconds.
Through sensors' data, JASMINE displays real-time most dangerous crossroads[1] and outliers ones[2]. It also provides information on the most congested paths[3].
We'll first analyze and justify our own design choices. Afterwards, we'll describe the architecture of the developed system and the solution provided to respond to the queries of the challenge.

[1] The first 10 crossed with highest speed;

[2] Those crossroads whose median of the number of vehicles that crossed them is superior than the global median of the vehicles that crossed all the crossroads;

[3] Most congested sequence of traffic lights (high number of vehicles and slow speed).

II. Design Choices

A. Frameworks and Programming Languages

JASMINE's control and monitoring system are coded in JAVA, well supported by Apache Flink. REST API implementations have been made easier by Spring framework utilization, chosen for its ease of use and connection to persistence level.
Node.js framework has allowed us to develop and implement a very simple and practical simulator useful to generate random data to test the system.
Regarding the front-end side (user interface), the framework AngularJS has been used and Bootstrap library has helped us with graphical elements.

B. Data Stream Processing

The first and the most important design choice is which data stream processing platform could be the most appropriate for the goals of the project. We have considered two alternatives, both owned by Apache Software Foundation: Apache Flink and Apache Storm.
The main reason that led us to chose Apache Flink (v. 1.5) is related to its time management. In fact, it supports stream processing and windowing with event time semantics. In particular, when events arrive out of order or delays occur it's easy to compute over streams.

C. Connector

As well as the data stream processing, connector choice has immediately been crucial and, as such, we analyzed different MOM solutions to handle communications between the components of the system.
Final choice fell on Apache Kafka instead of other MOMs.
We finally chose Apache Kafka because of its scalability and High Availability.
Furthermore, Flink's official site provides documentation for the Flink-Kafka connectors.

D. Persistence

Persistence level ensures that traffic lights and crossroads inserted into the system, through REST API, in JSON format are stored in a NoSQL database, chosen to provide high availability. We finally decided to use MongoDB as document-oriented database, because it well supports horizontal scalability and its queries executions speed ensures high performances.
To optimize reads and writes on the system's elements, MongoDB's feature for geographical indexing has been used: it consists on GeoJSON attributes utilization.

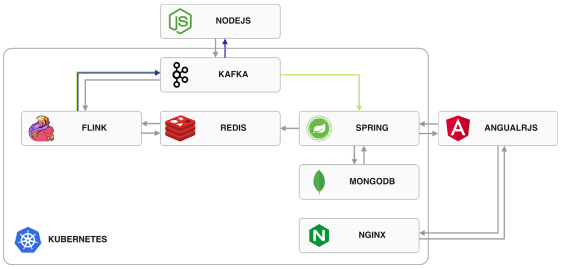
E. Redis

Redis is an open-source in-memory data store which can be used as database, as cache or as a message broker. In JASMINE Project, Redis allows to store and quickly find geographic coordinates of the cells the city has been divided into to have an efficient mapping with pings coming from users' mobile devices.

F. REST API

New components entries in the system are managed by REST API that allow, in addition to insertion, the deletion, the editing and the research. REST API are also used to inform the system that a broken lamp has been fixed and replaced.
Furthermore, connection between Back-end side and Front-end side is via REST API. A more detailed description will be provided in a dedicated section.

III. Architecture



Node.js is used to generate messages that simulate sensors placed on traffic lights and on mobile users. Kafka then manages these messages by inserting them in special topics that are read by Flink who manages the data stream. Redis is used by Flink to retrieve information on the grid that allows it to efficiently calculate the position of a mobile user in space.
MongoDB is used to persistently store all the information to be displayed in the frontend by AngularJS.

User Interface

JASMINE provides a user-friendly interface developed and implemented with AngularJS and using the open source toolkit Bootstrap. Thanks to the interface it is very simple and intuitive to manually add new components such as Traffic Light and Crossroads with their own geographic coordinates and it is possible to easily analyze real-time ranking related to the most dangerous crossroads, outlier crossroads and most congested path for three different required time windows (15 minutes, 1 hour, 24 hours).
It is also possible to get a real-time list of traffic lights with damaged light bulbs (alerts drop-down list) and making them working and available again through REST API.

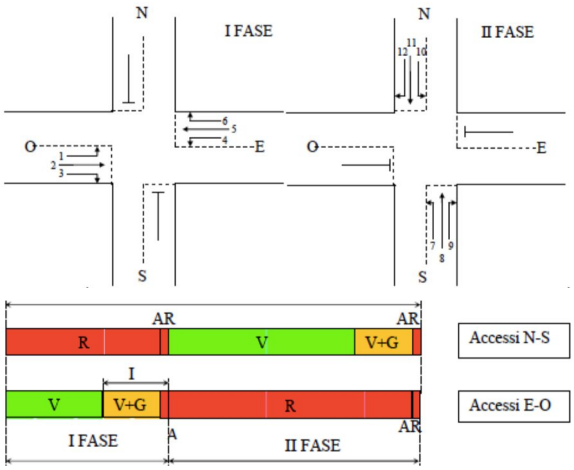
IV. JASMINE Queries

A. Control System

Control System's aim is to adapt green light duration and, as a consequence, red light too (yellow light duration is considered to be fixed) in order to make the urban traffic more fluid. In particular, yellow light has a duration of 4 seconds and the complete cycle lasts 200 seconds.
To achieve this goal, we have chosen to use a simplified version of Webster's algorithm.

Before proceeding with the application of Webster's method, the *phasing plan* of the crossroads must be evaluated; it is necessary to establish in how many phases the traffic-light cycle must be organized, which maneuvers are allowed in each phase and which the sequence of phases should be.
It is also necessary to know the extent of the pedestrian flows, as well as the geometry of the intersection. In the following version, the pedestrian flows and the problems connected to them will be neglected. Generally it is preferred, when possible, to use only two phases because this minimizes the time intervals loss, which occur every time the phase changes. For this reason and in order to simplify the philosophy of the controller, only two phases will be used for each crossroads of the system.[4]

[4] In the case of high turning flows, there may be crossroads congestion phenomena and therefore it is more convenient to use multiphase tables, if the intersection geometry allows the provision of a number of waiting lanes in each access at least equal to the phases of green for the same access. There are many criteria to establish when it is necessary to prepare a special phase for the left turn; generally this is done when the volumes of this maneuver exceed 150-200 vehicles / h and when the flows of crossing the opposite access are high.



A.1 Webster algorithm

The method proposed by Webster (1958) and Webster and Cobbe (1966) allows to obtain the cycle time and the green time duration for a single crossroads whose phase matrix is well known.

The quantities are introduced:

- R_i : red duration of the access representative of the generic phase *i* [*sec*]
- G : yellow light duration
- V_i : green light duration
- C_l : semaphore cycle time [*sec*]
- V_{Ei} :effective green duration of the representative access of the generic phase *i* [*sec*]
- AR : the duration of all red lights, when all the traffic lights are red for safety reasons
- L_i : lost time of the representative access of the generic phase *i* [*sec*]
- q_i : flow arriving at representative access in the generic phase *i*, in other words, the number of vehicles arriving at an access in a generic phase *i* $[\frac{vehicles}{hours}]$
- I_i : exchange time is the time not used to bind the crossroads
- S_i : saturation range of the representative access in the generic phase *i*, or better, maximum disposal flow during the VE period $[\frac{vehicles}{hours}]$. It starts from the hypothesis that each phase is represented by a single current, that for which the ratio between the incoming volume q and the relative saturation flow S is maximum.

To simplify the controller, the following assumptions have been made:

- The Lost Time $L_i = l_s + l_c$ is set at 4 seconds, l_s is the starting time and l_c is the clear time;
- The exchange time $I = G + AR$ is set at 6 seconds, where *G* is set at 4 seconds and *AR* is set at 2 seconds;
- The Saturation Flow *S* was assumed equals for all the crossroads and equals to 360 vehicles every 5 minutes.

In particular, the saturation flow *S* depends on several factors (access width, road slope, vehicle flow composition) and it is calculated as follows: $S = S_b \cdot k_1 \cdot \ldots \cdot k_n$, where S_b is the base saturation flow rate and $k_1 \cdot \ldots \cdot k_n$ are the various influencing factors:

$$S_b = 525 \cdot L \ \frac{vehicles}{hours}$$

where L is the useful width of access.

It was decided not to consider the factors $k_1 \cdot \ldots \cdot k_n$ and three-lane roads in each direction were considered, each lane 2.75 wide (according to the urban road code standard). Thus we get $S = S_b = 525 \cdot 8,25 = 4331 \ \frac{vehicles}{hours}$ and therefore $S_b = 72 \ \frac{vehicles}{minutes}$.

A.2 Steps of the algorithm

The steps of the algorithm are the following:

1. Find the representative currents, in the i-th phase only consider the access

values for which $\frac{q}{S}$ is maximum (e.g. if green phase 1 includes north access and south access and if $\frac{q_{Nord}}{S_{Nord}} > \frac{q_{Sud}}{S_{Sud}}$ then the current the north access will be representative, from this moment on the calculations of this phase will be made with respect to the values of the north access)

2. Calculation of the effective green according to the i-th phase for the representative flow:

$$V_{Ei} = \frac{\frac{q_i}{S_i}}{\sum_{i=1}^{n} \frac{q_i}{S_i}} \cdot (C_l - \sum_{i=1}^{n} L_i)$$

3. Calculation of the green duration:

$$V_i = V_{Ei} + L_i - I_i$$

4. Calculation of the red duration:

$$R_i = C_l - (V_i - I_i)$$

A.3 Webster in Jasmine

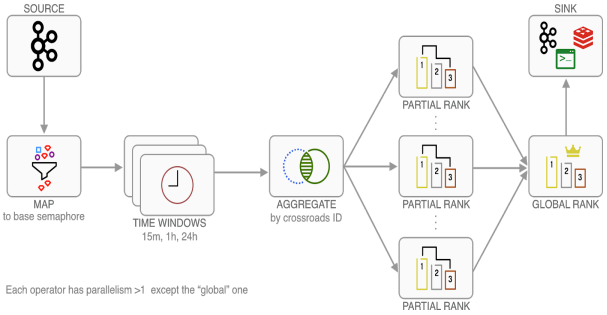
Actually within Jasmine only the value of the green light duration, changed according to Webster's algorithm, is sent to the crossroads, and according to this each traffic light calculates the value of its own red light duration.
The final goal of the controller is to consider the flow of vehicles and calibrate the traffic light cycle in order to facilitate traffic flow.
Currently a testing phase on the controller has not been foreseen because of the many assumptions we made and above all because of the random generation of the vehicle flow.

B. Monitoring System

An important purpose of the monitoring system is to generate an alert every time a damaged traffic light's bulb is detected by the system. The alarm provides information on its position and on the lamp's type.
The light bulb can be later repaired through the front-end side (expressed REST API to repair it). In the simulator, a simple function *exponentialRandomNumber(μ)* is used to simulate the lightbulbs switching off.
The stream of messages coming from the mobile sensors is immediately filtered. If one of these messages indicates that the respective semaphore has a damaged lamp, the message passes through the filter. Subsequently the unnecessary information is deleted for this message. Finally, the message is serialized and published on Kafka.
But the main goal of the project, and in particular, of the monitoring system is provide real-time information required by different 3 queries. Let's analyze them in details.

1) Query 1

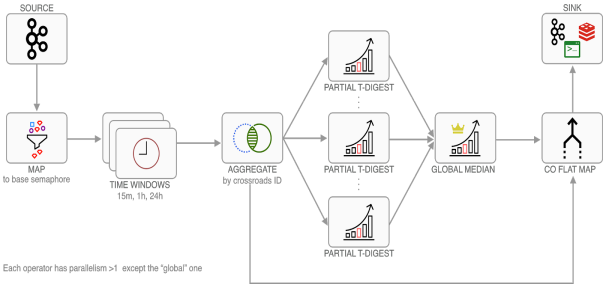
The goal of the first query is to compile a ranking of the 10 intersections crossed with greatest average speed. It's required to calculate it on well determined time windows (15 minutes, 1 hour and 24 hours) and update it in real-time. The aim is to provide a crossroads hazard index. The passage of a vehicle is simulated by *jasmine_simulator*, which is responsible for the publication of a tuple every 60 seconds for each traffic light. This tuple contains different information, some of them useful to characterize the position of a traffic light and its belonging to a crossroads, others concerning the values that extrapolate from the traffic status, such as the average speed of vehicles passing near the traffic lights.
We now show the query topology.



The first step is the filtering of the message stream coming from the traffic lights, removing the information that does not concern this query. In this case the information on broken light bulbs is removed.
From the new stream, the timestamp is extracted and three *sliding windows* of 15 minutes, 1 hours and 24 hours size are created, aggregating the various messages according to the ID of the crossroads which they belong to. In this way all the traffic lights that fall into the time window are grouped by crossroads.[5]
An additional level is added before the final ranking is drawn. In fact, partial classifications are calculated and, finally, they are aggregated in the global ranking.
In the calculation of the partial classifications, parallelism is raised in such a way as to keep a potential growth of these under control, while for the global ranking it is necessary to set the parallelism to one for consistency reasons.
Now the final ranking is filtered and if it has changed, compared to the previous computation, it is serialized and published on Kafka. Otherwise it is blocked because it is not required to provide an output.

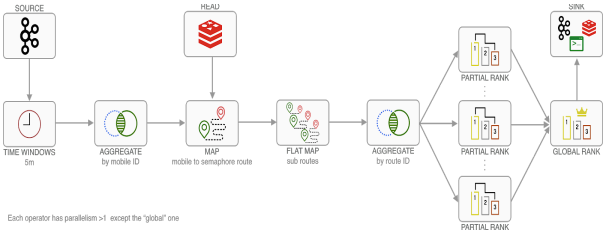
2) Query 2

The purpose of the second query is to show the crossroads having the value of the median of the number of vehicles, which have crossed the intersection, higher than the value of the global median of the vehicles, which have crossed all the intersections.
The median value is calculated on 15 minutes, 1 hour, 24 hours time windows. The goal is to provide a tool to identify intersections subject to greater congestion.
As in the first query, *jasmine_simulator* deals with the simulation using the same information of the previous query.



As we see from the image that shows the topology of this query, the first part is identical to the query 1.
After creating the three "sliding windows" and grouping by crossroads ID, a new aggregate is made and the global median is updated.[6]
At this point, a comparison between the global median stream and the crossroads stream is performed through a coFlatMap.
The result of the comparison will lead to publish on Kafka all crossroads above the updated median.

3) Query 3



[5] In details the *aggregate* we are doing on the windows requires a grouping phase according to a key, *keyBy* operation, (in this case the key is the ID of the crossroads). From the original stream, sub-streams are created and Flink will manage them in parallel, gaining in efficiency.

[6] The update of the median was made through the use of the T-Digest library which allows to efficiently update the value of the median value by value.

The goal of the third query is to identify the sequence of traffic lights that in the last 5 minutes is more congested (being characterized by a high number of users moving at low speed), exploiting the flow of data from mobile sensors. The mobile sensors give information on the position (in terms of latitude and longitude) and on the users/vehicles speed. The main effort of the query is to make the mapping between the position of the vehicles with the mobile sensors and the nearest traffic light, this is necessary to approximate the vehicle's route through the traffic lights. This will be explained in detail later.

The simulation of mobile sensors is done through *jasmine_mobile*, which is responsible for publishing messages from mobile sensors on Kafka.

The first step is to extract the timestamp from the messages belonging to the stream of mobile sensors. Thanks to the timestamp, it is created a *sliding window* of 5 minutes where the various messages are grouped (keyBy method) according to the ID of the mobile sensor. The result is a series of points in the space with associated timestamps grouped by mobile sensors' ID. It is necessary to associate to these points a route through the traffic lights network.

For each point in the stream I perform an efficient search on Redis to find the nearest traffic light. Redis is used to divide the urban area into a grid of cells (whose size is configurable). Each cell refers to a traffic light, if a point belongs to a cell it is associated with the reference traffic light of the cell. This operation brings a compression of the stream. In fact if two points refer to the same cell only the closest point is considered. At the completion of this mapping a traffic light path for each ID is provided.

Being the sliding window 5 minutes long and since the mobile sensors send messages every 10 seconds, we will (except errors) routes to the most composed of 5 traffic lights. Each path is then subdivided into all the possible sub-paths, the new timestamps are re-aggregated and a unique ID is assigned to each sub-path. The ID is created automatically simply by concatenating the ID of the traffic lights that belong to the sub-path. For each route a congestion value is associated. It is a function of the average travel speed and the number of vehicles on the route. At this point it is sufficient to draw the ranking among all the sub-paths and extract the maximum. This value is subject to a last filter, if the new value corresponds to the old one, it is not necessary to produce an output, otherwise the old value is updated. The path is serialized and published on a Kafka topic.

Reduce computation by windows chaining

A different solution has been implemented in the management of time windows to reduce the computation on the data streams that arrive at the system. The use of this solution can be configured in *applicationProperties*. The *windows chaining* utilization, instead of *sliding windows,* allows exploiting the previous window computation to compute the next window.

In other words, instead of computing the data stream three times for different time windows (15 minutes, 1 hour, 24 hours), it is used the previous window to compute the next one. This gain must be managed in order to consistently compute the mean and median values: in fact in this implementation the crossroads contain additional information with this purpose. For example, it would be an error to compute the global average computing the average of the averages.

V. AWS

To deploy the system, AWS platform has been chosen. We also decided to use Kubernetes, Google's open source system, as orchestration tool.

Cluster's creation and management has been entrusted to Kops framework (Kubernetes Operations).

The entire system is based on pods containing Docker images of different components: Zookeeper, Kafka, MongoDB, Redis, JDK8, Nginx, Flink-jobmanager and Flink-taskmanager. The first four elements are instantiated by Kubernetes' StatefulSet which allows to generate *n* replicas of the same container with a common state. This choice is justified by the need to ensure that a crash of a StatefulSet element does not imply the loss of the state.

Other pods are instantiated using the Kubernetes Deployment that guarantees pods regeneration when a crash occurs.

Kafka, JDK8 and Nginx are images created ad hoc by JASMINE team and they are now available on DockerHub under Zed25 profile. The need to have controlled and preconfigured images' behaviours led us to the decision of creating our own Docker images.

JDK8 upload REST API (developed with Spring) jar and Nginx contains the frontend code.

It is possible to configure the cluster through .yml files that serves to generate the different components of the system.

It is also possible using templates to create configuration files indicating the changes.

Using the "*./jasmine_create_base_cluster.sh*" and "*./jasmine_deploy.sh"* files it's possible to deploy the cluster. Three Amazon instances are generated: a t2.micro as master and two m4.large (2 vCPU, 8GiB RAM) as nodes.

The choice for the nodes is due to the Amazon limitations on EC2 instances requests. It could be sufficient having nodes with lower power.

On these three instances the following are created: 3 Zookeeper, 1 Kafka, 1 MongoDB, 1 Redis, 1 Nginx, 1 JDK8, 1 Flink jobmanager and 8 Flink taskmanager. EC2 m4.large instances and 8 Flink taskmanagers provide 16 computational slots to JASMINE, widely sufficient to execute our jobs with parallelism 4.

However it is possible, if it is necessary, scale horizontally adding pods for Flink and, subsequently, m4.large instances to the cluster.

This approach can be applied to each component of the system.

Furthermore, to ensure HA, it is possible to spread the cluster on three Availability Zone Amazon makes available in the same region.

VI. Testing and Deployment

Some metrics are measured through Flink dashboard UI. In particular we have evaluated throughput and latency when a 50 crossroads input (200 traffic light/s) is pushed into the system. We tested JASMINE system on 15 seconds time window and with a parallelism 4.

Throughput results have been satisfying, in fact we obtained about 1348 tuples/s with this configuration so we focused on latency data.

The following table reports our data.

Query	Latency (s)
1	0.488
2	0.315
Damaged TL	0.147

We could improve these data increasing parallelism value. It is also possible to do scale-up improving instances types (e.g. from m4.large to m4.xlarge). This will bring more computational power on a single instance and a better network connection. Another solution, more immediate and less expensive, is scaling-out the system increasing EC2 Amazon instances.

We pointed out that the latency is strongly influenced by time windows because Flink keeps tuples in function of the slide of the windows.

VII.Configuration

Documentations related to the configuration of the different components of the system is on github profile "Abyssi" in the following repositories:

- jasmine_core
- jasmine_rest
- jasmine_simulator
- jasmine_frontend
- jasmine_semaphore
- jasmine_mobile

VIII. Conclusions

We described the design, implementation and evaluation of our Flink-based solution for CINI Smart City University Challenge 2018. The aim was to choose the architecture and the relationship between the individual operators with the aim of parallelizing and distributing the workload.

The complete system is always subject to possible and desirable improvements. For example, it is considered necessary to manage consistently the updating of the duration of the green lights for each semaphore in a crossroads. In this regard it is necessary to introduce a consensus algorithm between the semaphores belonging to the same crossroad, such as Paxos, to be tolerant to possible failures. Another improvement could be to insert an intermediate layer between the traffic lights that belong to a crossroads and the central system. This additional level could precompute the tuples generated by the traffic lights' sensors, reducing the overall load.

Finally, it is important to implement security mechanisms able to guarantee that the tuples accepted by the system are not malicious tuples. In the Smart City view this is a very important topic in order to avoid possible attacks on the entire traffic light system.

Obviously the main focus of the work was directed towards the architecture and the communication of the various components. The system makes various assumptions that represent any possibility of improvement.

References

[1] Apache Kafka Official Website http://kafka.apache.org

[2] Apache Flink Official Website https://flink.apache.org

[3] Apache Flink Documentation https://ci.apache.org/projects/flink/flink-docs-stable/

[4] Kubernetes documentation https://kubernetes.io/docs/home/

[5] Kops documentation https://github.com/kubernetes/kops

[6] Spring documentation https://spring.io/docs

JASMINE

Just A Simple Modular Intelligent Network Environment

Tiziano Ditoma
0252737
University of Rome – Tor Vergata
Computer Engineering, Master Degree
tiziano.ditoma@students.uniroma2.eu

Francesco Ottaviano
0259193
University of Rome – Tor Vergata
Computer Engineering, Master Degree
francesco.ottaviano@students.uniroma2.eu

Simone Mancini
0259568
University of Rome – Tor Vergata
Computer Engineering, Master Degree
simone.mancini@students.uniroma2.eu

Andrea Silvi
0258596
University of Rome – Tor Vergata
Computer Engineering, Master Degree
andrea.silvi@students.uniroma2.eu

Attachment - API

Introduction

This section, included as an attachment to the documentation, is intended to explain in detail REST APIs exposed by our system.
It has been decided to insert this documentation as an attachment in order to divide this and the paper describing the system.

The first section describes the main system entities in detail, followed by a description of the implemented REST APIs.

Optionally, as an additional interface functionality that allows us to view changes in real time, the possibility of using sockets to receive messages via the STOMP protocol (*Simple Text-Oriented Messaging Protocol*) has also been introduced and will be shown in the final section.

NOTE: Time window milliseconds have been considered as key for information retrieval

Entities

LightBulbColor	[RED, YELLOW, GREEN]
LightBulbStatus	[ON, OFF, INACTIVE, DAMAGED]

LightBulb	{
	color (LightBulbColor),
	status (LightBulbStatus)
	}

GeoPoint	{
	latitude (double)
	longitude (double)
	}

Semaphore	{
	id (String),
	location (GeoPoint),
	greenTime (int),
	lightBulbs (Array[LightBulb]),
	crossroads (Crossroads),
	semaphoreId (String)
	}

Crossroads	{
	id (String),
	semaphores (Array[Semaphore])
	}

RichSemaphore	(Semaphore) {
	speed (double)
	}

RichCrossroads	(Crossroads) {
	averageSpeed (double),
	medianVehiclesCount: double
	}

OutlierCrossroads	(Crossroads) {
	timestamp (long),
	medianVehiclesCount (double),
	timeWindowMilliseconds (Integer)
	}

SemaphoreRoute	(Array[RichSemaphore])

SemaphoreRouteLeaderboard	{
	id (String),
	list: (Array[SemaphoreRoute]),
	timeWindowMilliseconds (Integer)
	}

CrossroadsLeaderboard	{
	id (String),
	list: (Array[RichCrossroads]) ,
	timeWindowMilliseconds (Integer)
	}

REST API

Semaphores

Create One	
URL	/api/v1/semaphores/
Method	PUT
Path Variables	/
Request Params	/
Body	Semaphore
Return	Status: 200 Body: “CREATED” Status: 400 Body: Exception Message

Get One	
URL	/api/v1/semaphores/{id}
Method	GET
Path Variables	id: String
Request Params	/
Body	Semaphore
Return	Status: 200 Body: Semaphore Status: 404 Body: “SEMAPHORE_NOT_FOUND" Status: 400 Body: Exception Message

Update One	
URL	/api/v1/semaphores/{id}
Method	POST
Path Variables	id: String
Request Params	/
Body	Semaphore
Return	Status: 200 Body: “UPDATED” Status: 400 Body: Exception Message

Delete One	
URL	/api/v1/semaphores/{id}
Method	DELETE
Path Variables	id: String
Request Params	/
Body	/
Return	Status: 200 Body: “DELETED” Status: 404 Body: “SEMAPHORE_NOT_FOUND” Status: 400 Body: Exception Message

Restore One	
URL	/api/v1/semaphores/restore/{id}
Method	GET
Path Variables	id: String
Request Params	/
Body	/
Return	Status: 200 Body: “UPDATED” Status: 400 Body: Exception Message

Get Page	
URL	/api/v1/semaphores/
Method	GET
Path Variables	/
Request Params	page: Integer pageSize: Integer (Optional)
Body	/
Return	Status: 200 Body: { “totalPages”: Integer, “content”:Array [Semaphore]} Status: 400 Body: Exception Message

Get Damaged Page	
URL	/api/v1/semaphores/damaged
Method	GET
Path Variables	/
Request Params	page: Integer pageSize: Integer (Optional)
Body	/
Return	Status: 200 Body: { “totalPages”: Integer, “content”:Array [Semaphore]} Status: 400 Body: Exception Message

Crossroads

Create One	
URL	/api/v1/crossroads/
Method	PUT
Path Variables	/
Request Params	/
Body	Crossroads
Return	Status: 200 Body: “CREATED” Status: 400 Body: Exception Message

Read One	
URL	/api/v1/crossroads/{id}
Method	GET
Path Variables	id: String
Request Params	/
Body	/
Return	Status: 200 Body: Crossroads Status: 404 Body: “CROSSROADS_NOT_FOUND” Status: 400 Body: Exception Message

Update One	
URL	/api/v1/crossroads/{id}
Method	POST
Path Variables	id: String
Request Params	/
Body	Crossroads
Return	Status: 200 Body: “UPDATED” Status: 400 Body: Exception Message

Delete One	
URL	/api/v1/crossroads/{id}
Method	DELETE
Path Variables	id: String
Request Params	/
Body	/
Return	Status: 200 Body: “DELETED” Status: 404 Body:”CROSSROADS_NOT_FOUND” Status: 400 Body: Exception Message

Get Page	
URL	/api/v1/crossroads/
Method	GET
Path Variables	/
Request Params	page: Integer pageSize: Integer (Optional)
Body	/
Return	Status: 200 Body: { “totalPages”: Integer, “content”:Array [Crossroads]} Status: 400 Body: Exception Message

Get Outliers	
URL	/api/v1/crossroads/outliers/{millis}
Method	GET
Path Variables	millis: Integer
Request Params	/
Body	/
Return	Status: 200 Body: Array[Crossroads] Status: 400 Body: Exception Message

Semaphore route leaderboards

Get One	
URL	/api/v1/semaphore_route_leader_boards/{millis}
Method	GET
Path Variables	millis: Integer
Request Params	/
Body	/
Return	Status: 200 Body: SemaphoreRouteLeaderboard Status: 400 Body: Exception Message

Crossroads leaderboards

Get One	
URL	/api/v1/crossroads_leader_boards/{millis}
Method	GET
Path Variables	millis: Integer
Request Params	/
Body	/
Return	Status: 200 Body: CrossroadsLeaderboard Status: 400 Body: Exception Message

Generate (intended to be used only for simulation)

Generate Crossroads	
URL	/api/v1/generate/{size}/{delete}
Method	PUT
Path Variables	size: Integer delete: Boolean
Request Params	/
Body	/
Return	Status: 200 Body: “DONE” Status: 400 Body: Exception Message

STOMP

Top 10 crossroads	
Topic	top-ten-crossroads-{millis}-topic
Topic Variables	millis: Integer
Message	CrossroadsLeaderboard

Outlier crossroads	
Topic	outlier-crossroads-{millis}-topic
Topic Variables	millis: Integer
Message	OutlierCrossroads

Top semaphore route	
Topic	top-semaphore-route-topic
Topic Variables	
Message	SemaphoreRouteLeaderboard

Damaged semaphore	
Topic	damaged-semaphore-topic
Topic Variables	
Message	Semaphore