Distributed Systems and Cloud Computing: Report

Simone D'Aniello
Data Science and Engineering
University of Rome Tor Vergata
Rome, Italy
daniello.simone.uni@gmail.com

Federico Ronci
Data Science and Engineering
University of Rome Tor Vergata
Rome, Italy
ronci.federico.1993@gmail.com

Marius Ionita
Data Science and Engineering
University of Rome Tor Vergata
Rome, Italy
ionita.maryus@gmail.com

***Abstract*—Nowadays, always more and more devices are connected through the internet, allowing a more powerful control among them and the possibility to gather a great number of information, useful to grant more efficient services.**

In this paper, well see how this innovation can be used to improve the service offered by traffic lights, making them capable to change timing based on the actual traffic state. For example, in Rome (city where this project has been built), people spend 35 hours per years in traffic, equals to the 23% of total time spent in car [5].

While we know that such an implementation would require a great amount of money, thats also true that this will be the natural evolution for cars and roads, since technology evolution cant be stopped.

I. Introduction

This project takes part in the CINI SMART CITIES UNIVERSITY CHALLENGE [1], which purpose was to build a distributed mechanism to control traffic lights timing while gathering useful information regarding the most dangerous intersection.

In order to accomplish the task, we needed a way to retrieve data from sensors placed in semaphores and making them eligible for collecting the readings and analyze them. A publish/subscribe system was our choice. However, we have lots of data coming from sensors in a continuous way, leading to the necessity for a Data Stream Processing Framework.

Finally, we want our system to be as fault tolerant and scalable as possible, meaning it must be distributed.

In the following section System we explain each component in the application. It may seem superfluous to show what is a semaphore or a crossroad and how they are organized but it is important to define the role of each node and how they communicate.

In the section Architecture the use of each framework is explained. While in System we explain physical nodes, in this section we explore the abstract architecture of the application.

In Choices is explained why some frameworks are used instead of others.

In the section Data injection and messaging we see how sensors data are simulated and produced. The standard through which the nodes communicate is shown too.

In Stream data processing and query resolution we show the core of the application explaining how queries are resolved. In Machine learning algorithms used for the decision process are shown.

In Emergency mode we see what happens when a node falls or the communication cant be established. In the section Storage is analyzed how data are stored in the database.

Summary is a digest of what we have seen until this moment. In the last section Test we show results and benchmarks.

II. System

In this section the system components and their interaction are shown.

At the beginning, each semaphore in the system carries out its task normally, but in the meantime a sensor placed on it registers information continuously, and send them to Cloud. Later, a controller uses the records received allowing the change of timing whenever is needed. For example, if in a crossroad a stream is much denser in a direction than in other one, the correspondent traffic light is allowed to keep its green state longer. This is made possible thanks to a Reinforcement Learning algorithm described later in this paper.

Semaphores are not the only entities capable to send information: in a world where smartphones are a global reality, it is natural to take advantage of their diffusion and network connection to gain more and more important data.

SemaphoreController module is built to manage everything that regards entities semaphore in our system, for example adding new semaphores, dealing with eventual malfunctions and their noticing to the user: It also makes advantages of reinforcement learning controller for timing semaphores in a crossroad.

Furthermore, it is the one involved in establishing an emergency mode if something should go wrong with the communications. In fact, if a message is not received by all traffic lights, unpleasant situations could arose. For this reason, messages are exchanged using a 2-Phase-Commit protocol enhanced by gossiping: each entity must know the decisions made by the controller.

But this is not the end: data sent by the sensors are also received by a component which performs stream processing, aggregating them and outputting useful statistics to retrieve, for example, dangerous semaphores, where traffic is denser. Everything is tied up by a publishing/subscribe framework, which allows an easy and efficient message exchange between interested components.

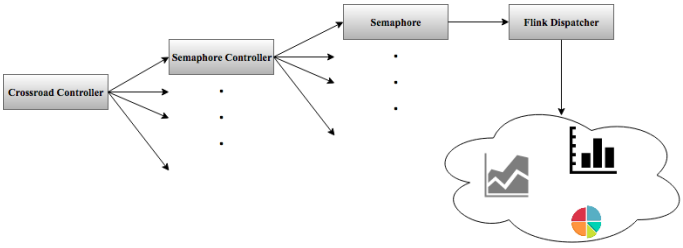

Fig. 1. System

III. Architecture

In this section we are going to explain projects architecture, i.e. nodes which must cooperate in order to make the system work. Data are generated by sensors which produce tuple containing different informations. These tuples are sent into a Kafka topic where are read both from a Flink instance and from a Java instance. Flink uses these data to compute aggregations and to send results on another Kafka topic. These results are read by a Java instance called Monitor in order to compute queries. Results of this computation are written into the document based distributed NoSQL database MongoDB. Analyzing the other stream, sensor tuples are read by the semaphore controller related to the semaphore which has sent data. Thanks to this messages the semaphore controller can decide which semaphore must be set to green (this phase is analyzed in details in the section Machine learning). This decision is sent on different Kafka topics where semaphores can listen to and be informed on which of them must change its state.

Front end is composed by a web page and a monitor instance. Front end has no communication with back end instances. The only way to retrieve informations about the system is reading from the database. In fact, the database contains all the necessary information for the user. These are discussed in Storage section.

As we have seen in System section the entire application is based on the following instances. Next list exposes which framework runs in which instance:

- **Semaphore**: It uses Kafka in order to read crossroad informations and send sensors data. It offers also a REST interface in order to be configured by user during instantiation.
- **Semaphore controller**: as explained before, it is involved in exchanging messages through 2PC protocol between semaphores, retrieves decisions made by the learning algorithm or everything that includes managing traffic lights interaction.
- **Crossroad controller**: This instance has access to storage system and stores semaphores status for which it is responsible together with its own on it. It also allows to add new semaphores or remove them.
- **Monitor**: it reads Flink results from Kafka, computes queries and stores the output on MongoDB
- **Monitor front end**: Reads data from MongoDB and offers an user interface to check semaphores in crossroads.
- **Monitor Back end**: it receives user requests on a REST interface;
- **Flink Dispatcher**: Reads sensor data from a Kafka topic and uses Apache Flink for stream processing. Sends results on another Kafka topic.

To sum up, the structure is quite simple. Data are sent on various Kafka topics from which are read by Flink and by Semaphore Controller. Flink uses these informations to compute queries and statistics. Semaphore controller instead uses them to decide which light to turn on.

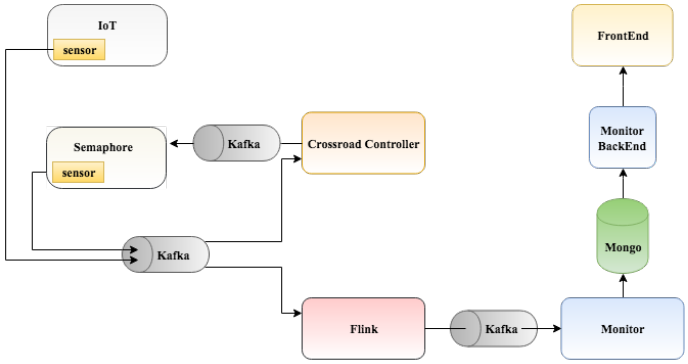

Fig. 2. Architecture

IV. Choices

In this section we explain why some frameworks are used instead of others.

Fig. 3. Choices

Our choice for Flink has been made mainly by its windows feature. Windows in Flink permit an easy control over time, after that results are outputted. For example, you can easily aggregate info for half an hour (or whatever you need) before seeing the results. As we need statistics for different time window, this framework has the better support. Not to mention its good performances in terms of latency and throughput, essential in a Data Stream Processing environment.

Apache Kafka is the most widespread framework for building real-time streaming data pipelines that reliably get data between systems or applications. Kafka and Flink, being both currently developed by the Apache Software Foundation [APACHE], can easily communicate using the Flink-Kafka connector. However, Apache Kafka is not used only for its interconnection with Flink. Lets make some comparison with the other messaging framework RabbitMQ. The latter distributed message queue system is designed to support protocols such as AMQP or STOMP. Rabbit MQ doesnt offer the exactly

once semantic and allows scaling up and down by simply adding and removing consumers. Kafka supports the exaclty once semantic and can scale further. Furthermore, tests show that Kakfa has lower latency than RabbitMQ.

Having settled the main framework, our next question was how to send data into the framework. Since were using an Apache product, it just seems logical to use another one. Thats why we choose Apache Kafka as a pub/sub system for data injection. In fact, Flink has built-in connectors for the streaming platform, making easy to write a data consumer from a Kafka topic. Kafka main feature are topics: in fact, it is possible to create more topics on a single running machine, and each topic is used for a different function for our system. For example, we have a topic for records produced by sensors placed on semaphores, and one for smartphone This messaging system is quite important because we can define different data fluxes which can be considered in an independent way, without risking an overlapping among them.

As we hinted before, communication between Frontend and Backend is realized through REST calls. For an easier implementation, we relied on the Spring framework. REST architecture is based on HTTP. A well-defined URL structure and the specification of the method (GET, POST etc.) are required. Spring grants users to define a mapping between URL and method, eventually passing a JSON body, through the use of annotation, simplifying a lot programmers job.

In the end, the problem of storing data arose. The reader may imagine that a simple RDBMS couldnt fulfill its duty here being too slow and incapable to scale for a distributed system. Thats why we chose a document-oriented DBMS, MongoDB to be precise. Results to store have different formats and the data flow requests complex queries. MongoDB offers a great flexibility, is easily scalable and has excellent performances.

V. Data injection and messaging

In this section we are going to show the message logic and tuple generation.

Data are generated in a simulated way: although we have a real dataset of existing semaphores, including latitudes and longitudes foreach, data produced by sensors (semaphores and mobile devices) are fictitious. We used java built-in pseudo-random-number-generator libraries, but obviously in a coherent way (meaning, for example, that a speed cant be 300 km/h).

Semaphores sensor tuples contains information related to the semaphore itself and the observed car flow(cars in unit time and mean speed).

Mobile devices records register current information about a car approaching a semaphore, meaning, among the others, its mean speed and coordinates.

These records are then sent into the correspondent Kafka topic for further analyze. Records in topics are consumed by controller to light up semaphores, or by Flink to compute queries. Each different query reference a different topic. Generally, tuples made by Flink or any kind of information that must be shared, is encapsulated in a Message class: its main purpose is to associate a code to the record for later filtering. For example, it may indicate who is the receiver, or if it is an error message. Following, a table with a list of codes, and their meaning.

code	from	to	aim
1	Semaphore	Controller	Add semaphore
-1	Semaphore	Controller	Remove semaphore
200	Semaphore	Controller	Send traffic condition
301	Controller	Controller	Start 2PC
302	Controller	Controller	Commit 2PC
-302	Controller	Controller	Rollback 2PC
311	Semaphore	Controller	OK 2PC
404	Controller	Semaphore	Signaling crossroad malfunction
621	FlinkDispatcher	MonitorBE	Add crossroad response
622	FlinkDispatcher	MonitorBE	Add semaphore response
623	Controller	MonitorBE	Get crossroad situation
70115	FlinkDispatcher	Monitor	Send average speed query 1 - 15 minutes
7011	FlinkDispatcher	Monitor	Send average speed query 1 - 1 hour
70124	FlinkDispatcher	Monitor	Send average speed query 1 - 24 hours
70215	FlinkDispatcher	Monitor	Send median speed for query 2 - 15 minutes
7021	FlinkDispatcher	Monitor	Send median speed for query 2 - 1 hour
70224	FlinkDispatcher	Monitor	Send median speed for query 2 - 24 hours
70315	FlinkDispatcher	Monitor	Send average speed query 3 - 15 minutes
7031	FlinkDispatcher	Monitor	Send average speed query 3 - 1 hour
70324	FlinkDispatcher	Monitor	Send average speed query 3 - 24 hours
2001	FlinkDispatcher	Monitor	FlinkDispatcher heartbeat

VI. Stream data processing and queries resolution

In this section data processing and queries computation are analyzed.

Records sent in input to Flink instances are in JSON format: this is a standard-driven choice, since it is our concern that our system should be as elastic as possible. With that reason in mind, we agreed that JSON format is the most diffused, utilized, easy to understand and easy to produce. Whenever a JSON string is read, a custom deserializer (that overrides Flink

flatmap) converts it into a Tuplex (where x is the number of arguments), which will be part of the stream to be analyzed.

To guarantee faster resolving, we took advantage of Flink .aggregate() method; in short, it allows to performs iterative computations using an aggregator Tuple: whenever a tuple arrives, the data it carries within are immediately used to update current mean or median (depending from the queries).

While it is easy enough to compute mean value iteratively, the same cant be said for the median (or quantiles in general). Thats why it is necessary an approximated value. For this reason, we used the PSquared algorithm [7]

Flink performs computation for a length of time established by the TimeWindow, after that results are outputted, while keeping aggregating new data. Output is produced every 15 minutes, 1 hour and 24 hours, and serialized-back into JSON format and placed in a Kafka Topic. A secondary component creates a ranking and make it visible to the user.

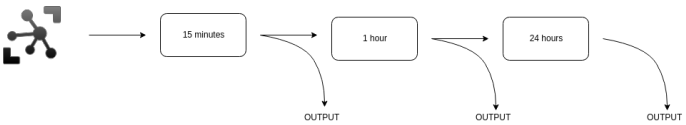

Fig. 4. Flink cascade process

VII. Machine learning

In this section we are going to explain machine learning algorithm used to solve queries.

Before starting our project, it only seemed natural to check the state of the art as regards the implementation of smart traffic lights in cities. Almost every scientific paper we found used the same approach: a Reinforcement Learning algorithm; more precisely a Q-Learning one.

The goal of Q-Learning is to learn a policy, which tells an agent which action to take under which circumstances. It does not require a model of the environment and can handle problems with stochastic transitions and rewards, without requiring adaptations. (from https://en.wikipedia.org/wiki/Q-learning)

By defining a Q-Learning algorithm, a reward function is needed. To keep it simple, we used an estimate of the queue at each semaphore. In a few words, the algorithm must answer these questions:

- How convenient is to turn on the traffic light?
- How disadvantageous is to keep turned off the traffic light?

Each semaphore entity contains two values, each one is an indicator about the questions presented before. By comparing each value of the semaphores, the most suitable to turn on is chosen. Learning formula is defined below:

$$Q^{(k+1)}(state^{(k+1),learning\_rate}) = (1-\alpha)Q^{k}(state, learning\_rate) + \alpha(queue\_size + \gamma + mawQ)$$

QueueSize parameter is obtained thanks to the informations registered by sensors placed in semaphores. Current state is updated each user-defined time slot, allowing more precise decisions as time goes on.

VIII. Emergency mode and malfunctions

In this section we explain what happens when a traffic light breaks or the connection between nodes is lost.

The emergency mode is a protocol which can assure that only one semaphore is set to green and which can avoid car crashes in case of lack of connection. This is very important because it allows our system to perfectly manage traffic even if something goes wrong, or data are not collected anymore. Sensor malfunction or difficulties for flink in dispatching messages should happen, emergency mode permits a stable (even if not perfect) semaphores work. This mode is realized defining a cycle time that gives a measure about how much should last green an red semaphore colour.

Since we underlined how dangerous it could be a broken traffic light, we thought about implementing a live notification system that could be intuitive and different from the most used solutions.

Telegram, a famous chat application, has everything we needed: in fact, we can rely on its infrastructure for sending messages and offers the possibility to create your own bot. In other words, we created a bot that send an alert message in a channel (a kind of group where only administrators can write, such that eventual spam is avoided and only emergency messages are sent), attaching ID of the broken semaphore, geo coordinates and eventual broken bulbs. This solution grants a free possibility to interact with whom is interested in, without worrying about creating the complete system (starting from registering devices, creating a fault-tolerant network and so on).

IX. Storage

In this section stored data format and use is analyzed. As we have explained in section Architecture the user must have access to the system state without communicating with back end architecture. For this reason is necessary to store the entire system structure. Informations saved are summarized in the following schema:

As you can see from the schema each item has a unique id and a type attribute. The latter is used to query the database in a simpler way. Each crossroad has an ID used to make query in MongoDB and another ID (computed using the crossroad street name) that is used in both back end and front end. The semaphore has an ID, which is assigned computing the hash function SHA256 on its latitude and longitude, and a integer malfunctions which represents the times the semaphore has recognized broken bulbs. Fields latitude, longitude and street are self explanatory.

Every controller is characterized by an unique ID and is defined by a type and an array containing a set of crossroads under its control. Every crossroad contained in the array is defined by other attributes like ID (assigned by Mongo), street

, a type field and a list of semaphores, which are part of the crossroads.

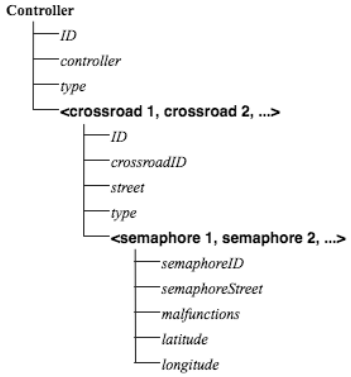

Fig. 5. Structure schema in mongoDB

MongoDB is also used to save query results and previous schema indicates the structure. Same schema is used for every query of the system and queries have an unique identifier and a list attribute called ranking which containes all objects in order. Every object in the ranking is identifed by an unique identifier. Ranking is made comparing value attribute, which is the result of Flink computation: for example, it may be the mean speed recorded by the sensors, the median of the cars, or values computed by data sent from mobile sensors. Other attributes serve the role to offer faster information to Frontend.

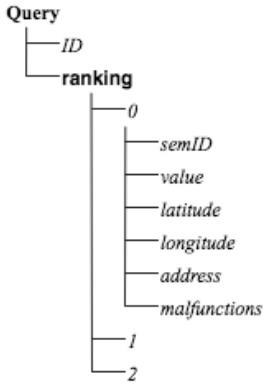

Fig. 6. Query schema in mongoDB

X. Summary

This project is designed to handle a complex traffic light system. It is composed by several components which communicate to guarantee that traffic lights are turned on in correct order, allowing the lowest wasted time while driving. In the meantime the system must return statistics about traffic and crossroads condition. It may be useful to summarize the entire data flow before talking about tests and results obtained.

The user driving near a traffic light sends data using a Kafka topic. The semaphore itself computes records statistics about vehicles and sends them another topic. Data are read both by the Flink Dispatcher and the semaphoreController. The first component uses messages to compute rankings based on the mean and median of the speed of vehicle passing through the crossroad. The semaphoreController uses data read by Flink as input of the Q-Learning algorithm. The semaphore to turn green is decided based on this algorithm. Thanks to this phase the user wasted time is minimized. The entire system status is saved on MongoDB. The user can control queries results and system active nodes using the web application.

XI. Tests

In this section tests made on each module are shown. These represents latency for all three queries, in particular these were performed on a computer with these characteristics: Hp i5-7600k 3.80 GHz, with operative system Ubuntu 18.04 and 8 GB RAM memory. All statistics are taken from Flink Dashboard UI.

A machine was used as sensor data generator in order to simulate data sent by 200 semaphores. Each semaphore is created with no delay and it send immediately information. Data are sent by each semaphore once per minute. Tuples are used to compute all the three queries and are sent in parallel in three different topics. Three Flink instances, one for each query, run on another machine and read data.

The difference between the mean and maximum latency can be explained as follows: often data arrive in burst, which cause a raise up of the time. However, we can also appreciate the fact that on average, latency is very low. This fact fact gives more confidence in the stability of the system presented.

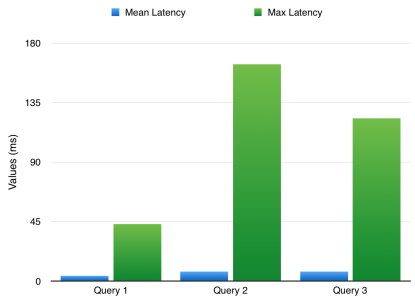

Fig. 7. Query schema in mongoDB

References

[1] CINI smart cities university challenge https://www.consorzio-cini.it/index.php/en/2014-07-15-07-42-21/9-italiano/news/1180-cini-smart-cities-university-challenge
[2] MongoDB. 2009. [ONLINE] Available at: https://www.mongodb.com/
[3] Apache Kafka. 2011. [ONLINE] Available at: http://kafka.apache.org/
[4] Apache Flink [ONLINE] Available at: https://flink.apache.org/
[5] Info data (in italian): http://www.infodata.ilsole24ore.com/2017/02/20/roma-la-citta-piu-congestionata-ditalia-anno-35-ore-perse-mezzo-al-traffico/
[6] Apache Software Foundation [ONLINE] Available at: https://www.apache.org/
[7] PSquared algorithm: https://www.cse.wustl.edu/ jain/papers/ftp/psqr.pdf