Project Ember

Open source project for a smart city illumination control system

Alessio Moretti
University of Rome Tor Vergata
Computer Engineering, Master Degree
(student id. 0239045)
alessio.moretti@alumni.uniroma2.eu

Federico Vagnoni
University of Rome Tor Vergata
Computer Engineering, Master Degree
(student id. 0245106)
federico.vagnoni@alumni.uniroma2.eu

ABSTRACT

This paper describes an open source solution to provide an efficient illumination system for a smart city. Project Ember was born during the DSCC[1] master course of Computer Engineering. The project currently uses Apache Flink for real-time data stream processing, Apache Kafka to handle messaging routing through the control system and the sensors, Elasticsearch to store efficiently statistics and data and to perform intelligent queries upon them, Python and Redis to prototype the local control unit to interface with streetlamps.

CCS CONCEPTS

•**Distributed Systems** → Autonomic systems; •**Computer systems organization** → Cloud Computing; Sensors network; •**Software engineering** → Message-oriented middleware;

KEYWORDS

Data processing, autonomic systems, Apache Flink, Apache Kafka, Redis, Elasticsearch, sensors network

1 INTRODUCTION

This paper is about an academic project born to be an efficient solution for the CINI[2] 2017 Challenge on smart cities illumination systems. In particular, the goal was to prototype and test a solution which was capable of (near) real-time data stream processing for monitoring records from street lamps, lumen sensors co-located with the street lamp itself and from traffic data produced by third-party APIs. We will explore this solution for the following use case: in a smart city context it is necessary to guarantee the maximum efficiency from lamps consumption while providing an optimal illumination within safety limits for pedestrians and drivers and according to local traffic intensity. To achieve that, it is necessary to project a grid of smart lamps capable of tuning their light level according to the right amount of energy necessary to provide city aware, safe and green consumption levels. This grid must be powered and managed via a reliable, highly available, processing-capable control system. Introducing Project Ember.

2 FRAMEWORKS AND TOOLS

We structured our environment using at first a publish/subscribe architecture with street lamps, lumen and traffic sensors as publishers and the stream processing framework as subscriber.

2.1 Data stream processing

The Apache Software Foundation makes available different alternatives, each one a refined version of the previous: Apache Storm is one of the most used data stream processing framework on the market and one of the most supported as well; Apache Spark is a refined version of Storm even though it is limited to fewer programming languages (Java, Scala and Python); Apache Flink[3] is the most recent project among the three and the most advanced. Flink gives the programmer the possibility to define just the topology of the operators, how they are linked, or how to set the windows timing (based upon the event time or upon the processing time spent inside the system). Flink handles the under-the-hood engine: multithreading, synchronization, parallelism, availability, cluster management. We chose the latest stable release of Apache Flink, 1.2.0, which comes with a well written documentation as well as multiple connectors for the most popular MOM[4] and storages. Flink calls a Source the very component that produces data and Sink the one that takes the processed data in order to store or to route them to another entity.

2.2 Connectors

To achieve scalability we had to analyze several options to let connect our Flink topology to messages routers and to the persistence level. The MOM chosen was Apache Kafka[5] works seamlessly with Flink thanks to the included connector plugins giving the possibility to simply personalize the connection according to our preferences: in particular in this solution Kafka is our preferred Source to handle data from the sensors. Talking about the Sinks, Flink supports many platforms and Kafka can be one of them (for example for the control output), but in order to persist and manage our data we wanted to use also a modern platform capable of organizing data for (near) real-time purposes.

2.3 Persistence level

A NoSQL approach was mandatory to us, to collect dynamic unstructured data typical of a sensor network, so we analyzed different products: in particular Elasticsearch[6] and Apache Cassandra. We chose Elasticsearch, as it is fully supported (not its last version indeed) by Flink 1.2.0 and it is a flexible, easy-to-deploy database with RESTful APIs. It leverage the ability to perform complex queries, even geographical and lexical ones, with good performances and scalability options. Elasticsearch is part of the Elastic Stack built by

[1] Distributed Systems and Cloud Computing
[2] Consorzio Interuniversitario Nazionale per l'Informatica
[3] Apache Flink official page: https://flink.apache.org/
[4] Messages Oriented Middleware
[5] Apache Kafka documentation: https://kafka.apache.org/documentation
[6] Elasticsearch official page: https://www.elastic.co/products/elasticsearch

Elastic.co which makes available another useful tool for visualizing data stored in Elasticsearch, Kibana[7].

2.4 Extras

To develp a local control unit to manage the city grid we used Flask and Redis. Flask[8] is a micro-framework for Web Server built in Python and Redis[9] instead is simple and efficienty key-valye data store and works as a database, cache and message broker. Redis serves us as a cache, allowing us to interact with control unit history with it with very simple APIs via the endpoints exposed by Flask.

2.5 Programming languages

Java is the programming language that links all these components together being used by Flink as well as Scala, such as Elasticsearch APIs. We also used Python for the control unit development as well as to realize the simulated data source for testing.

3 ARCHITECTURE OVERVIEW

In this section we will cover how the system communicates between each of its components and modules and the assumptions we made to prototype and test the architecture. In figure 1 a high-level architecture overview is provided. Before proceeding, we want to focus on the output from the real-time[10] control system: it is produced into the MOM and consumed by control units (how will be discussed later), closing a feedback loop. This behavior and the capabilities to maintain high-availability across the clusters make the system itself near to the features of an autonomic system.

[7]Kibana official page: https://www.elastic.co/products/kibana
[8]Flask official page: http://flask.pocoo.org/
[9]Redis official page: https://redis.io/
[10]We will define the system as "real-time" in this paper even if is not validated for such a control system, but it is capable of near real-time data streams processing

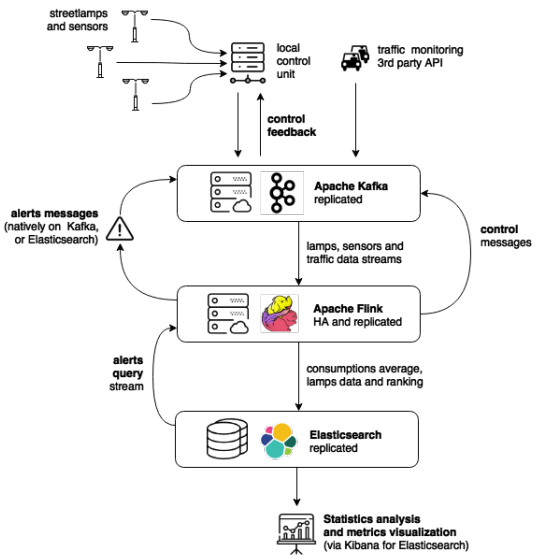

Figure 1: Project Ember architecture overview

3.1 Sensors network

First of all let us consider how the sensors network sends its data to the control system. According to project specifications the street lamps sends to the system a JSON formatted string containing all the information their microcontrollers collect as a tuple[11]. Those data are sent every 10 seconds. A lumen sensor is placed on the lamp and it sent data with same rate as well giving us information about the daylight luminosity level and the id of the lamps it is placed upon.

3.2 Control unit

The control unit gives us the possibility to create a new indirection level that is placed among the lamps and Flink. The lamps talk, using the city intranet, to the local control unit which maintains a mapping between the lamps ids and their IP addresses inside the local city network. The control unit manage the correct routing of the messages from the lamps to the stream processing operator via Kafka, as well as the control feedback received via the same MOM. This solution was introduced both for the indirection it introduces and the plug-and-play registration of each new lamp via RESTful APIs, as well as the capability to store the sensible data at the edge of the network to decrease latency, giving us some features of the fog computing paradigm.

3.3 Apache Kafka cluster

To handle the thousands of data necessary to manage the infrastructure, we thought to use a cluster of replicated nodes running the same Apache Kafka instance, using Apache Zookeeper[12] to maintain the cluster available via redirection from a public IP. As we will see later, the MOM allows us to register topics for lamps, sensors and traffic monitoring data to retrieve them as a stream in Flink, and to create automatically topics for each control unit in order to retrieve the control feedback for each sector of the city grid. In addiction we included the possibility to route via Kafka alerts messages (to be consumed later by a custom operator).

3.4 Apache Flink cluster

This is the core of the architecture. The Apache Flink framework was used to create a data stream (near) real-time processing system to handle the thousands of tuples per seconds from the city grid and to produce for each of them a control output, as well as aggregations by streets or identifiers to produce statistics, ranks (by last replacement) and to store them for further analysis. Flink is also used to continuously monitor data from Elasticsearch and to produce alerts for any kind of failures that is collectable from each lamp history. To provide the necessary resilience and to let the processing system to scale, it is intended to be deployed as an highly-available cluster, composed of a JobManager (replicated and managed via Apache Zookeeper), acting as a master, and some TaskManagers, acting as slaves, into which is replicated every stream according their computational power (one core is one execution slot), producing

[11]in particular a unique integer id, address and model, consumption, intensity level, power on status and last replacement (as a timestamp in seconds) plus the timestamps in seconds of the instant the tuple was sent
[12]Apache Zookeeper official page: https://zookeper.apache.org

several degrees of parallelism proportional to the cluster size. Moreover, it is handled via the JobManager any transition from a set of parallelized stream into a single time window to perform any bath computation (as for example average computation).

3.5 Elasticsearch and Analytics

The ranks, the consumption mean by id and by address, have to be visualized and showed in real-time, so Elasticsearch comes in help. Flink offers an interface for Elasticsearch, that, simply providing the cluster configuration, connects to it and allows to send data in byte encoded json format strings. Elasticsearch organizes data by so called Indexes and Types. The Index is the equivalent of Database in a NoSQL approach and the Types the equivalent of Tables. Elasticsearch keeps tracks of data assigning them a mapping table so it can be capable of understand primitives or complex data types. Kibana comes in our help for representing them. By a simple file of configuration you can use Kibana to create your own dashboard by defining all the data to be analyzed specifying the Type it has to use, how rearrange the attributes, how to order and in which style visualize them. Elasticsearch and Kibana has been designed to easily interact each other, keeping synchronized the dashboard across Kibana clients, and be simply deployed as well. Elasticsearch can be extended without any amount of effort, simply defining new nodes for the cluster and joining the master node. Elasticsearch serves also a particular role: being designed as a RESTful engine as well, it allows to specify complex query and resolves them efficiently, using at its core Apache Lucene, an open source machine learning library.

4 DATA STREAM PROCESSING

5 A MESSAGE ORIENTED APPROACH

Apache Kafka plays its MOM role connecting all the components of our system so let's see how they are related to each other thanks to it: control units and Flink produce a lot of data in streams that are organized in topics: lamps, lumens and traffic, being specific. A system such as this can be sufficient but let's think: being a smart lamps grid the system has to emit orders directed to a single lamp among thousands. Kafka can manage runtime topics creation but a topic for each lamp is meaningless and inefficient; requiring the data stream processing system to know the right position of the lamp among network is infeasible as well. Let we analyze better the problem.

5.1 Sensors to Kafka

Lamps are supposed to be given, with a micro-controller built inside them capable to connect to the city intranet, to understand the power level proper of the bulb model and how such parameters relates one another in order to obtain the right luminosity. A lamp produce a JSON formatted string containing: the lamp id, the model of the light bulb, the timestamp of the last replacement instant of time, the current power consumption, the luminosity level, the control unit where it is registered, etc.

The lumen sensor is placed upon the street lamp and is managed by the same micro-controller; lumen sensors common for entire streets are managed as well. The lumen produce a JSON formatted string containing the same id of the lamp where it is placed upon (or a nonce for the common sensors), the luminosity value recorded, the street where it is located and the timestamp of the time instant when the data has been recorded.

The traffic sensor is realized by third party APIs and it is registered to Kafka producing a JSON formatted string containing the street monitored, the traffic value and the timestamp of the recorded time.

5.2 Kafka to Adapters

Once Flink calculates the state a street lamp should have it sends control directives to custom topics. Each lamps, in fact, is linked to a control unit and this information is included in the JSON sent by lamps to the system. That's where the control directive is routed, a topic that is identified by the string that identify the control unit responsible of a particular street lamp; so we can, on one hand, free Kafka of an incredible number of topics and, on the other hand, free smart lamps to register themselves on their ones. As we said even the alerts of lamps not functioning or not communicating for a too long time can be routed to Kafka. Such alarms are simply transmitted to it using the connector provided by Flink that requires a byte encoding of the Alert object and the specification of the topic where they are directed to, a custom module can be easily engineered via a Kafka consumer to handle the alerts properly (our project let the user specify via configuration file to store the alerts into Elasticsearch instead).

5.3 Controlling the lamps

Each lamps is registered to a particular control unit (we will cover how in the next section), which takes care to register itself to the topic of its own, so it can read the responses and understands to which lamp direct them. Once the response will be made available the control unit reads the messages and convert them to a JSON object so it can access their attributes; it determines the id of the lamp and check in the Redis database to find the IP address related to that lamp id. That's the core of the indirection level of the whole architecture. Kafka and the control unit makes possible all of this, making our system totally plug and play for a better large scale deployment.

6 THE CITY GRID

6.1 Local streetlamps control

6.2 Control units at scale

7 TESTS AND PERFORMANCES

7.1 The city simulator

8 NEXT STEPS AND CONCLUSIONS

8.1 Security

8.2 Deployment

8.3 Conclusions