

# Analysis and Design of Information Systems

Astrinakis Nikolaos  
dept. of Electrical and Computer Engineering  
National Technical University of Athens  
Athens, Greece  
el17008@mail.ntua.gr

Kaparou Alexandra  
dept. of Electrical and Computer Engineering  
National Technical University of Athens  
Athens, Greece  
el17100@mail.ntua.gr

Tampakakis Christos  
dept. of Electrical and Computer Engineering  
National Technical University of Athens  
Athens, Greece  
el17042@mail.ntua.gr

**Abstract**—The Internet of Things (IoT) is the next wave of innovation that promises to improve and optimize our daily life based on intelligent sensors and smart objects working together. Knowing how to securely transmit these data across multiple networks, collect, store and analyze them in order to turn them into useful information in real time is of major interest in recent works. This paper presents a live streaming system that acts as a prototype of an actual IoT system.

**Index Terms**—Internet of Things, IoT, Time-Series Data, Late Events, Window-Based Analysis, ActiveMQ, Apache Flink, Cassandra, Grafana

## I. INTRODUCTION

The Internet of Things (IoT) is an emerging paradigm that enables the communication between electronic devices and sensors through the internet in order to facilitate our lives. IoT uses smart devices and internet to provide innovative solutions to various challenges and issues related to various business, governmental and public/private industries across the world. [2] It is progressively becoming an important aspect of our life that can be sensed everywhere around us. In whole, IoT is an innovation that puts together extensive variety of smart systems, frameworks and intelligent devices and sensors (Fig. 1).

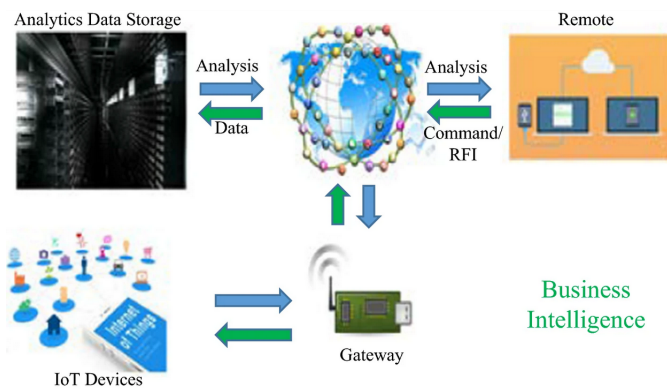


Fig. 1. General architecture of IoT

It is able to do so by taking advantage of stream processing, i.e. the computer science paradigm that deals with a sequence of data (a stream) and a series of operations being applied to each element in the stream. The most obvious benefit of stream processing is its ability to treat data not as static files (tables or other data structures) but as an infinite stream. In most cases, stream processing is associated with real-time processing, the method where data is processed almost immediately. [3] In real-time processing, the delay in data processing must be minimal, something that is difficult to implement with simple systems and adds an overload of data in case of system failure. Real-time processing consists of continuous input, processing, and analysis and output (reporting) of data. This processing paradigm aims the lowest-latency during the process. In this context, many frameworks are available for real-time big data processing like Storm, Spark, S4, Flink, Samza. The frameworks mentioned above are also characterized as fault-tolerant in order to minimize the risks of a system failure. This paper is about the processing of data flows in real time using the open source distributed systems ActiveMQ, Apache Flink, Apache Cassandra and Grafana.

## II. LIFECYCLE OF REAL-TIME BIG DATA PROCESSING

In today's big data platforms, in general sense, big data processing consists of the following stages: data acquisition, data storage, data analysis, and data reporting. In terms of real-time big data processing, the lifecycle consists of phases with continuity and time-limited. The lifecycle is given in Fig. 2 with related tools and tasks. [4]

In more detail the steps of the lifecycle are the following :

- **Data Ingestion** : This phase is the process in which big data are ingested from heterogeneous data sources. In real-time ingestion the data stream is started and continued systematically and it is then stored temporarily or sent to a stream processing framework.
- **Data Storage** : This phase covers the operations for storage of real-time data streams having different data

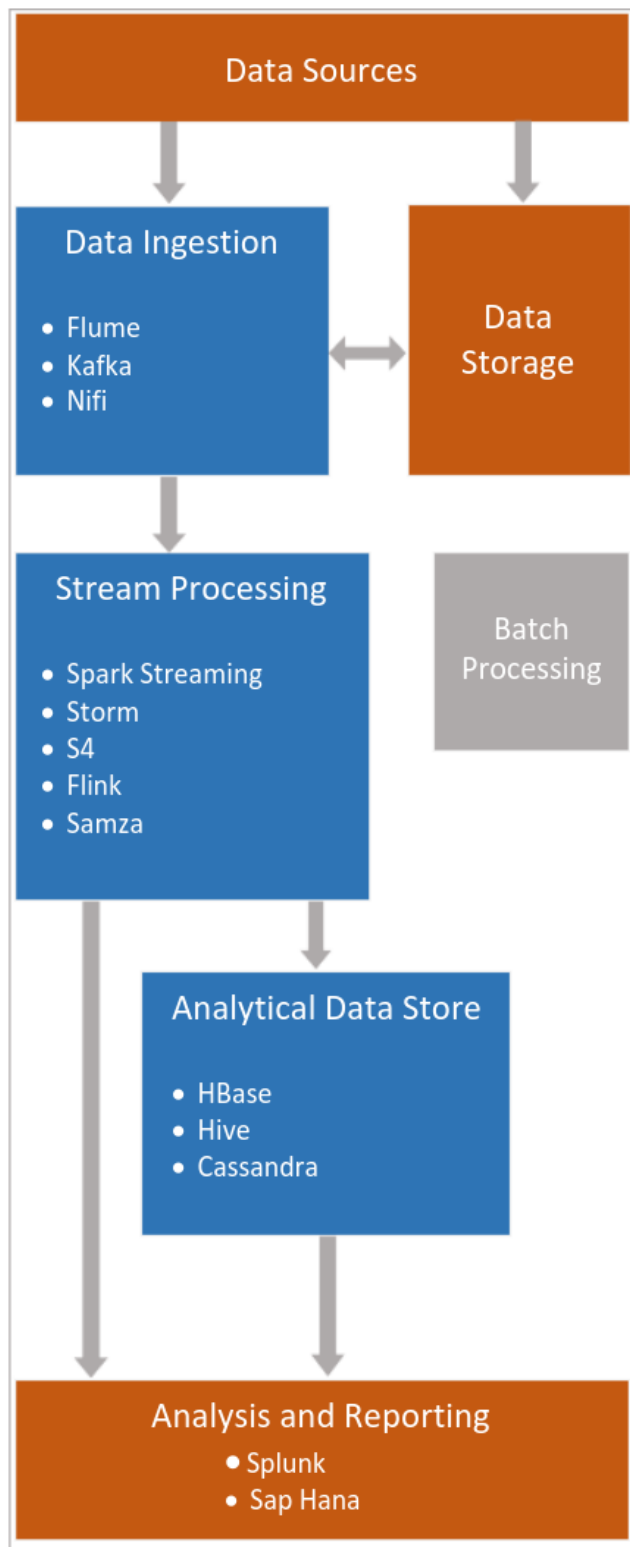


Fig. 2. Lifecycle of real-time big data processing

structures. In our project the phases *Data Ingestion* and *Data storage* are merged in one since the message broker ActiveMQ does exactly that, it ingests the data from the data source and stores them temporarily.

- *Stream Processing* : Stream processing focuses on the real-time processing of continuous streams of data in motion. A stream processing framework simplifies parallel hardware and software by restricting the performance of parallel computation. The pipelined runtime system of Flink in our project allows the execution of bulk/batch and stream processing applications.
- *Analytical Data Store* : In this step the data are stored permanently on a database in order to be used in the next step. The database can be both SQL and NoSQL, but the main restriction is that it has to be distributed. In our project, Apache Cassandra is a NoSQL database that provides an open source, distributed and decentralized/distributed storage system for managing large volumes of structured data spread out across the world.
- *Analysis and Reporting* : After being stored in the database, the real-time data can be captured, indexed, and correlated in a searchable warehouse, which can generate dashboards, visualizations, graphs, reports, and alerts. In this specific project the warehouse that was used is Grafana.

### III. PROJECT

#### A. Objective

The main objective of this project is the creation of a prototype for a working IoT system. The structure of the system we are about to describe could be used in various applications of an actual IoT system, such as a 'smart home'.

#### B. How it works

The live streaming system we had to realize, accepts virtual data as an input in real time from various sensors. Then, the data are processed so that they can be sorted by day, and afterwards, they are stored in a database. At this point the results of the processing can be viewed in a Dashboard provided by the database. The source code of this project can be found in the following link: <https://github.com/tampakc/InfoProjectFlink>

#### C. Data Processing

The way the data are processed, is divided in three parts. As referenced earlier, the first part is sorting the data by day, which is done to make data extraction easier. The second part is performing numerous operations on the data, and more specifically calculating the average, the sum, the minimum and the maximum values. Then, the third part is the detection of late events, the definition of which is described in the section below.

#### D. Late events

A very important part of our project is late events. A late event is defined as a message (in our case a number) that is being produced in a specific time inside the time window we have defined, but is received by Apache Flink after the window has passed. In defining a window we are given the choice of allowing a certain amount of *lateness*, which is a period of time after the window has ended that an event may arrive late and still be used in the aggregation. In our project we allowed no *lateness* in our windows, and as such, any event that arrives after its window has passed will be left out of the calculations. In this project our goal is to collect these events, acknowledge them as late and then display them per day in a table.

### IV. SOFTWARE

The software used for this project is the following:

- Data Generator
- ActiveMQ
- Apache Flink
- Cassandra
- Grafana

#### A. Data Generator

We created a random data generator using Java that sends numerical data in specified time intervals. Moreover, we configured the generator to send some late events at random times. The intervals at which the late events are sent are not correlated, so that they can be truly classified as late events.

#### B. ActiveMQ

ActiveMQ is one of the most popular open source tools that function as message broker. It is written in its entirety in Java, which means that it uses the Java Messaging Service (JMS), something that makes message exchange much easier. Message brokers are usually used for sending and receiving messages from various sources, so that the problem of message loss could be avoided. This problem would occur when during the use of some web service, the service would crash. In order to fix this, message brokers use a FIFO message queue, which allows the temporary storage of messages, allowing them to be sent only when the receiver is available. More specifically, ActiveMQ can receive and send messages through a variety of protocols (such as AMQP, MQTT, REST, Stomp a.s.o.) to clients that have been written in JavaScript, C, C++, Python, .Net a.s.o., something that helps establish ActiveMQ as a flexible and easy-to-use tool. The ActiveMQ topics function as subscription-based channels that can be used to create a 'one-to-many' communication. In this way, when the publisher (source) sends a message, every recipient (receiver), that has been registered to this specific topic will receive the message. In our project, the data are sent by a random data (number) generator, then received by ActiveMQ and subsequently they are ready to be used by Apache Flink.

#### C. Apache Flink

Apache Flink is an open source live stream-processing and batch-processing framework written in Java and Scala. It is mostly used for the processing of data. In our project, specifically, it is used for the calculation of the minimum, maximum, average and sum operations on our data. Processing from Apache Flink can be performed both on bounded and on unbounded streams. In the first case, the data have a clear starting and ending point, which means that they can be processed after every single piece of data has been received and sorted first. Meanwhile, in the second case, the data have a clear starting point, but their ending point is not specified. An occasion such as this, for example, is data streams, which are constantly received and processed immediately after their reception by Apache Flink. Afterwards, whenever the final results are produced, they are sent to the database for storing.

#### D. Cassandra

Apache Cassandra is a noSQL Timeseries distributed database, which provides the user with high availability without the risk of failure or error. This database can store large quantities of data and it is distributed, meaning that the data are written in various different places, and not just one. As a result, there is minimum risk of an error, because even if a node fails it can be immediately replaced.

#### E. Grafana

Grafana is an open source tool that is used for the execution of timeseries data analysis and the display of said data in the desired form. It can access the data stored in Cassandra that have been produced by Apache Flink (min, max, average and sum operations that were mentioned earlier). After it accesses said data, it can display them in the desired state with the use of charts and other methods in the Grafana Dashboard. It can additionally display every single late event that has been detected in a table.

### V. STRUCTURE

In this section we describe the steps that we followed in order to install all the prerequisites and then use them to create the asked live streaming system.

### VI. DESCRIPTION AND ANALYSIS OF THE INSTALLATION

The following section is a description of the steps followed in order to install the necessary programs for the realization of the project. The environment we worked on was Ubuntu 20.04.

#### A. Installation of ActiveMQ

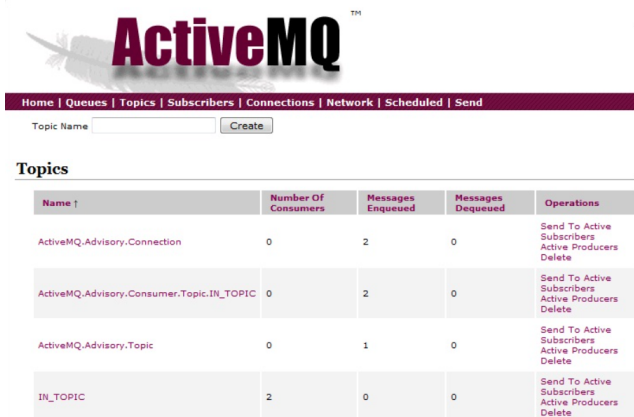
Starting with ActiveMQ, we visited the Apache ActiveMQ Website [5] so that we can make sure that we have every prerequisite program installed. This is done in order to ensure the proper function of the service. The only program required was the JRE 1.8, so we proceeded with the download. Following the installation of the JRE, we also had to set the JAVA\_HOME environment variable. Afterwards, we had to

download the compressed binary file and extract it to continue with the installation. For the rest of the process we followed a tutorial [6]. Following the steps of that tutorial, we created a directory named '/opt/activemq' and then moved the extracted files there, while at the same time we created a group account 'activemq' and a user for this group so that we can run the service. We also had to change the permissions for the directory that we created and to create a 'systemd' service file to control the ActiveMQ service. Finally we had to edit the jetty.xml configuration file and the installation was finished. Now, we had to reload the system daemon and start the ActiveMQ service, while verifying its status, making sure that the installation was completed successfully. The code for the installation is provided below:

```
sudo apt install openjdk-11-jre -y

wget http://archive.apache.org/dist/
  activemq/5.16.3/apache-activemq
-5.16.4-bin.tar.gz
sudo tar -xvzf apache-activemq-5.16.3-bin.
tar.gz
sudo mkdir /opt/activemq
sudo mv apache-activemq-5.16.3/* /opt/
activemq
sudo addgroup --quiet --system activemq
sudo adduser --quiet --system --ingroup
  activemq --no-create-home
--disabled-password activemq
sudo chown -R activemq:activemq /opt/
activemq
sudo nano /etc/systemd/system/activemq.
service
sudo nano /opt/activemq/conf/jetty.xml

sudo systemctl daemon-reload
sudo systemctl start activemq
sudo systemctl status activemq
```



Name ↑	Number Of Consumers	Messages Enqueued	Messages Dequeued	Operations
ActiveMQ.Advisory.Connection	0	2	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Consumer.Topic.IN_TOPIC	0	2	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Topic	0	1	0	Send To Active Subscribers Active Producers Delete
IN_TOPIC	2	0	0	Send To Active Subscribers Active Producers Delete

Fig. 3. WebUI of Apache ActiveMQ

Now that ActiveMQ is set up properly, we can access its WebUI (Fig.3), by visiting the URL localhost:8161/admin/.

## B. Installation of Apache Flink

Moving on with the Apache Flink installation, we start again by checking the prerequisites. There were none apart from JRE, which we installed earlier, so we start following a tutorial [7]. Initially, we downloaded the setup file and renamed the installation directory, changing our working directory to the aforementioned directory. Now what we had to do was start apache flink by running the start-local.sh file and after that check the status of the service. The code required for this is written below:

```
wget https://dlcdn.apache.org/flink/flink
-1.14.3/flink-1.14.3-bin-scala_2.11.
tgz
sudo tar xzf flink-1.14.3-bin-scala_2.11.
tgz
sudo mv flink-1.14.3/ flink
sudo cd flink
sudo jps
```

After the installation, we can now visit the WebUI (Fig.4) with the following url: localhost:8081.

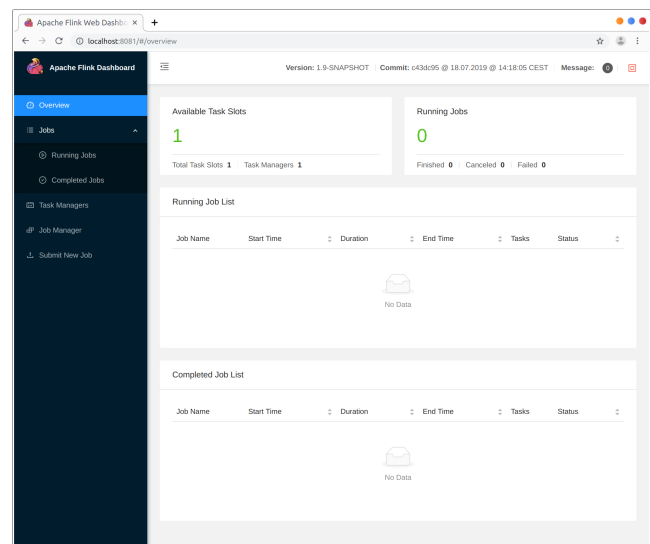


Fig. 4. WebUI of Apache Flink

## C. Installation of Cassandra

Our next step was the installation of Cassandra. We, again, checked the prerequisites to ensure the success of the installation, and just like before, every prerequisite program had been already installed during the two previous installations of ActiveMQ and Apache Flink, however this time, we had to install the 'docker' package in order for the installation to proceed. We followed the tutorial that is on the official Cassandra website [8], and as a first step, we had to use a command to pull the latest Cassandra version, something different to the other installations performed so far, since we didn't have to manually extract it. After that, we just ran a command to start a Cassandra cluster and then we ran a

command in order to verify that the service is running as expected and that no error occurred in the duration (Fig.5). The commands that were used, are:

```
docker pull cassandra:latest
docker run --name cass_cluster cassandra:
    latest
docker exec -it cass_cluster cqlsh
sudo systemctl status cassandra
```

Here, we can access the WebUI of Cassandra by entering the URL <https://localhost:3000/>.

```
tampakk@Chris-PC:~/Documents/flink_workspace/infosystems$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.0.0 | Cassandra 4.0.1 | CQL spec 3.4.5 | Native protocol v5]
Use HELP for help.
cqlsh> use infosystems;
cqlsh:infosystems> select * from daily;

window | avg | max | min | sum
-----+-----+-----+-----+-----
(0 rows)
cqlsh:infosystems> █
```

Fig. 5. Command Line Interface of Cassandra

#### D. Installation of Grafana

The final installation we had to perform was Grafana. First, we visited the official Grafana site [9] and carefully followed the instructions on the page to perform the installation. According to these instructions, we had to start by installing some packages that would help make the process easier. After that, we used some commands to install the latest Grafana version. Then we started the service and, just like before, we used a command to verify that the service was running as correctly. The commands that we used in order to achieve this are presented below:

```
sudo apt-get install -y apt-transport-
https
sudo apt-get install -y software-
properties-common wget
wget -q -O - https://packages.grafana.com/
gpg.key | sudo apt-key add -
echo "deb https://packages.grafana.com/
enterprise/deb stable main" | sudo tee
-a /etc/apt/sources.list.d/grafana.
list
```

After the installation was properly done and the status of the service was ensured to be active, we can access the cqlsh, which is the CLI for Grafana (Fig. 6) .

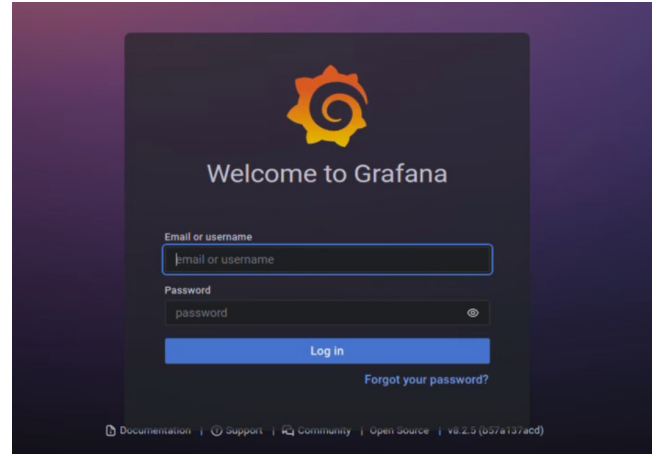


Fig. 6. WebUI of Grafana

## VII. DESCRIPTION AND ANALYSIS OF THE EXPERIMENT

The following section describes the steps we followed in order to realize the asked system. An UML Component Diagram of the system that shows every component and its usage can be seen in Fig. 7.

#### A. ActiveMQ

In ActiveMQ as we have mentioned before, we had to set up a Queue (Fig. 7) which our Java script would send messages to, and Apache Flink would be subscribed to (Fig. 8). Our Queue is named Random\_Numbers.

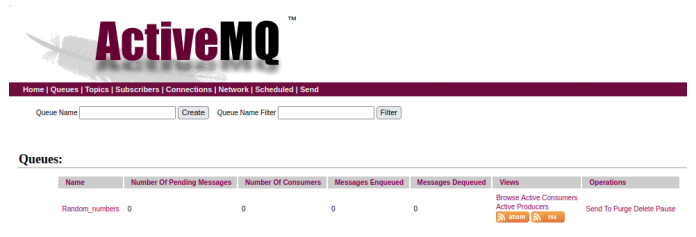


Fig. 7. Queue Random\_Numbers in ActiveMQ

Active Consumers for Random\_numbers

Client ID	Connection ID	SessionId	Selector	Enqueues	Dequeues	Dispatched	Dispatched Queue	Prefetch	Max pending	Exclusive	Retrospective
ID Chris-PC-43551-1646922039898-0-1				0	0	0	0	1000	0	false	false
ID Chris-PC-43551-1646922039898-1-1				0	0	0	0	0	0	false	false

Fig. 8. Consumer of Random\_Numbers

#### B. Apache Flink

Apache Flink requires the use of specialized connectors to communicate with other components. These come in the forms of Data Sources and Data Sinks. As our service requires Flink to receive data from ActiveMQ and store the results in Apache Cassandra, we will need to make use of the respective source and sink for which the dependencies will have to be added to our project and packaged with the jar file that will be uploaded to the Task Manager. Something that needs to be noted is



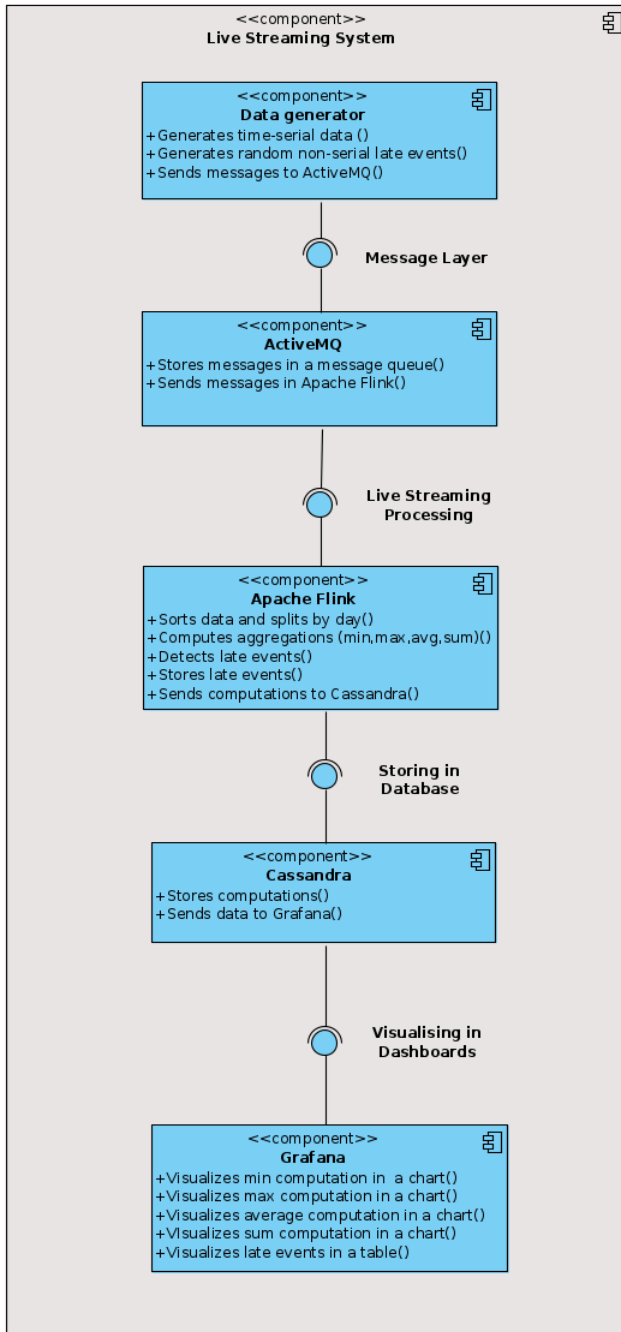


Fig. 9. UML Diagram for the structure of our project

that we decided our time-window to be 6h and not 24h. The reason for that is that if we had a 24h time-window, then in order to collect a great amount of data per time-window we had to wait for up to 3 days, which made the processes of collecting, categorizing per day and aggregating the data very time-consuming.

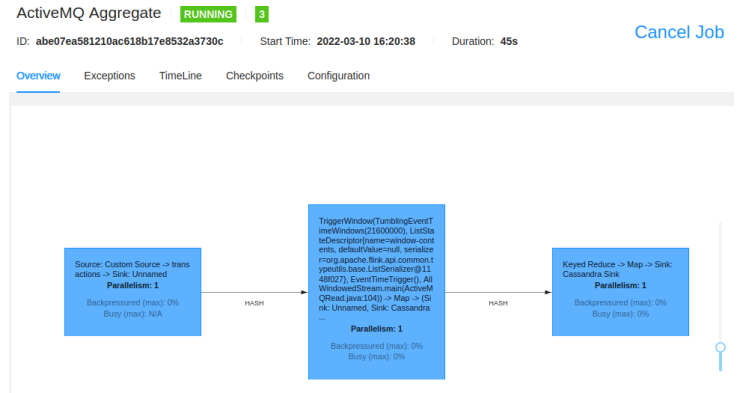


Fig. 10. Apache flink task overview

### C. Apache Cassandra

As far as Apache Cassandra is concerned, we had to use CQL in order to add the data provided by Apache Flink to our database. Our first step, was creating a new keyspace called 'infosystems' and use a command so as to use this keyspace we just created. Afterwards, we had to create two tables (or columnfamilies); one for our statistics and another one for our late events (Fig. 11 and Fig. 12). We had to declare the types and names of the data each of the tables consisted of, as well as their primary keys. The columns for the 'statistics' table were the 'window' column, which is a timestamp that describes the time window in which data were received, 'sensor\_id', which is an integer defining the sensor that each piece of data was received by, and 'avg', 'max', 'min' and 'sum', that are integers which describe the average, maximum, minimum and sum respectively of the data received in a specific time window. The primary key for this table is a composite key, consisting of the "sensor\_id" column acting as a partition key and the "window" column acting as a clustering key. The "late" table consists of the "sensor\_id" column, identical in datatype and function to its counterpart in the "statistics" table, a "time" column indicating the production time of the event that was sent late, and the "value" column containing the value of the sensor at the time. Here the primary key is again a composite key, consisting of "sensor\_id" as the partition key and "time" as the clustering key.

Here it should be noted that the sensor\_id column in this project is a dummy column made necessary by the usage of Grafana, and it could be a detriment if it were a real service. In the beginning our schema didn't include this column because it was not necessary, our service collected data from a single source, and as such an ID column would be superfluous. However Grafana expects that a single table

will hold data from multiple sensors, so it is required that we have a “sensor\_id” column from which will isolate the relevant rows with a WHERE clause. In order for our app to work correctly under these conditions, “sensor\_id” had to be a partition key, otherwise the WHERE clause can’t be used with it. This means however, that since all data have the same ID, they will all end up in the same node, thus we lose the distributability of Cassandra, and the workload is not distributed. This would only be a detriment if the app did indeed collect data from a single source, if it were a real app, the data would statistically be hashed equally to all nodes as the number of sources increased.

```
cqlsh:infosystems> describe table statistics;

CREATE TABLE infosystems.statistics (
  sensor_id int,
  window timestamp,
  avg double,
  max int,
  min int,
  sum int,
  PRIMARY KEY (sensor_id, window)
) WITH CLUSTERING ORDER BY (window ASC)
```

Fig. 11. Statistics table

```
cqlsh:infosystems> describe table late;

CREATE TABLE infosystems.late (
  sensor_id int,
  time timestamp,
  value int,
  PRIMARY KEY (sensor_id, time)
) WITH CLUSTERING ORDER BY (time ASC)
```

Fig. 12. Late table

#### D. Grafana

First of all, we had to add Apache Cassandra to the list of available datasources of Grafana, since there was no such option by default. We did that by installing a plugin from the Grafana page of available plugins and specifying the Host address of Apache Cassandra and the required credentials of the database as shown in the Fig. 13.

We then had to specify the following attributes:

- Keyspace : The Keyspace name defined in Cassandra (here **infosystems**).
- Table : The Table which holds the data that we will visualise in a chart (here **statistics** and **late** respectively).
- Time Column : The column storing the timestamp value, it’s used to answer the “when” question (here **window**).
- Value Column : The column storing the value we would like to show (here **avg**, **max**, **min** and **sum** respectively).

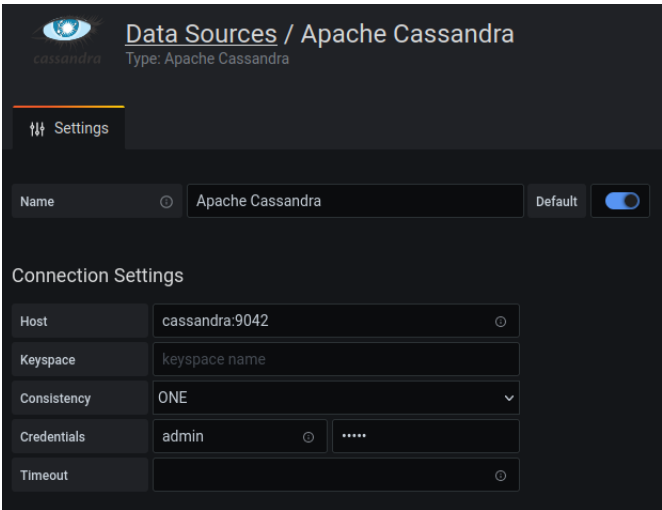


Fig. 13. Defining Cassandra as the Datasource of Grafana

- ID Column : The column to uniquely identify the source of the data (here **sensor\_id**).
- ID Value : The particular ID of the data origin we would like to show (here **1**).

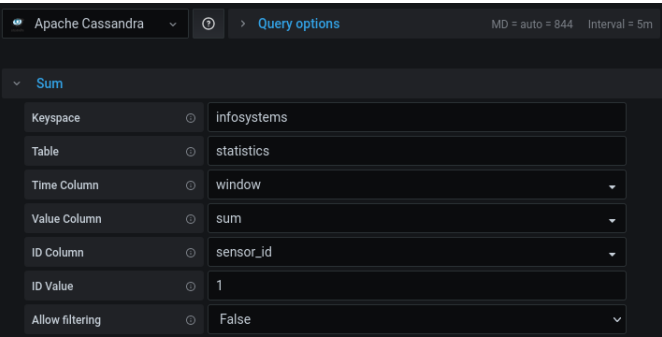


Fig. 14. Specifications for the statistics chart

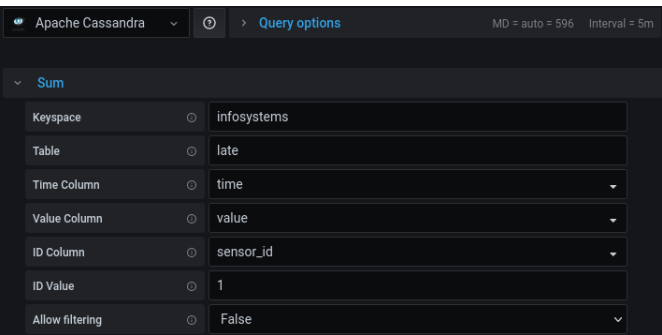


Fig. 15. Specifications for the late chart

### VIII. PROBLEMS AND TROUBLESHOOTING

During the realization of this project, there were some problems that had to be dealt with before it could be finalized.

### A. Flink's API and Stream classes

Having worked previously with PySpark RDDs, Flink's API worked somewhat differently than we had expected. Instead of the result of each operation performed on the stream being a dataframe on which any other operation can be performed, operations performed on a DataStream result in a different class, on which different operations can be performed. This means that Flink's API sticks more closely to the map-reduce workflow, whereas with RDDs we could perform two mappings or two reductions consecutively. This was a problem for instance when splitting the data into windows. Data arrived into our system in the form (timestamp, value). Late events needed to preserve this form, however the events that were to be aggregated needed to be mapped to correspond to the final schema. This meant that the mapping should happen after windowing, however Windowing results in an AllWindowedStream, which doesn't support the map operation but mostly aggregations and reductions, that our stream wasn't ready for. This issue was resolved by making use of AllWindowedStream's apply() method, which worked essentially like a map() function, with the convenient benefit of exposing information about the window that owns every event, making it easier to have access to a window's starting time without having to perform calculations based on the timestamp of the event and the window length.

### B. Java to CQL Datatypes

The goal of this project was to take events generated every 15 minutes and aggregate them, storing them permanently afterwards in Apache Cassandra. The job script for Apache Flink can be either written in Java or Scala. For the purposes of this app we chose Java. The aggregation of the data was to be done preferably within a time window of a day, turning them into daily data. Thus the initial schema we chose to store the results in Cassandra was daily (day date, avg double, min int, max int, sum int). This turned out to be quite troublesome, because the data had to be transformed from a Java datatype, into a CQL datatype. For these transformations, Flink's API made use of certain *codecs* which handled the mapping from Java to CQL. For the primitive types there was no issue, however for the CQL Date type, the transformation proved troublesome. The only codec that Flink recognized for CQL Dates was `com.datastax.driver.core.LocalDate`  $\longleftrightarrow$  CQL Date, but using that datatype in our calculations gave rise to another issue. When trying to transport data from Flink to Cassandra, the data had to be serialized first, using the `KryoSerializer`. The serializer however proved unable to perform this operation for the abovementioned datatype. Thus it became impossible for us to use the schema proposed above in Cassandra, containing the day column, because there was only one codec available for it and the corresponding Java type couldn't be serialized. One proposed solution was to make use of POJOs, Java objects that follow certain conventions that make them usable by various frameworks, which would allow them to be serialized and sent to Cassandra. This did not resolve the issue and as such we had to try a different approach. We resolved the

issue by making use of an alternative schema: daily(window timestamp, avg double, min int, max int, sum int), which had 2 advantages over the previous one. Firstly it allows for more granular windowing, because previously windows could only be multiples of a day, whereas with the new schema windows could be as small as we needed them to be, greatly increasing the flexibility of the app, and allowing for easier testing and debugging by producing results without having to run for 24 hours. Secondly it made use of Java's default Date type, for which both a codec was available and serialization was possible.

### C. Connectivity of Apache Cassandra to Grafana

Another problem that we encountered was about the connectivity of Apache Cassandra to Grafana. In order for the data stored in Apache Cassandra to be displayed on Grafana, a plugin was required, so we found a plugin, created by HadesArchitect that connects the two services [10]. We proceeded with the installation of the plugin and used Cassandra as a data source. However, when we filled in the necessary fields for the connection to begin, an error message saying "Query Data Error" was displayed. We then found out that this plugin had a bug, that was preventing the connection between them of being established. This bug would occur if the version of the plugin was lower than 1.4.4, something that was weird, considering that when we installed the plugin there was no update option, so we were stuck using the 1.0.0 version and we had to find a way to solve this problem manually. We then followed the steps described on the page of the plugin for the execution of a dummy scenario [11] in a provided demo program, after which, the plugin was automatically updated to the latest version.

## IX. ANALYSIS OF RESULTS

In this project, we were required to extract information from the randomly generated data and compute specific aggregations. The aggregations and the corresponding resulting charts are shown below:

- Minimum (min): With the min operation, we compute the minimum value per day that a sensor has detected.
- Maximum (max): With the max operation, we compute the maximum value per day that a sensor has detected.
- Sum: With the sum operation, we compute the sum of the values per day that a sensor has detected.
- Average : With the average operation, we compute the average of the values per day that a sensor has detected.

As seen from the graphs below, each graph has data aggregated per 6 hours. It should be noted that the windows should align with 00:00, 06:00, 12:00 and 18:00 but this clearly doesn't happen in the graphs. That is because the windows do align with those hours in GMT+0, but Grafana is displaying them based on local time, which is GMT+2, thus aligning with 02:00, 08:00, 14:00 and 20:00. Furthermore, we chose to use Grafana's staircase display option to more accurately represent the



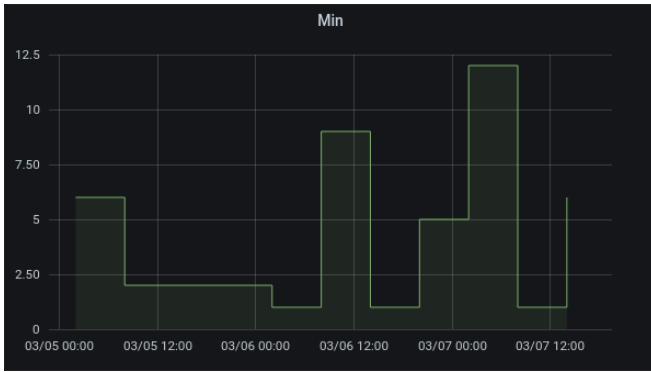


Fig. 16. Min aggregation

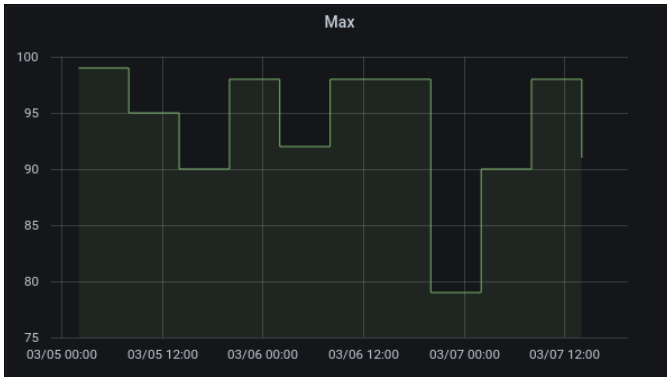


Fig. 17. Max aggregation

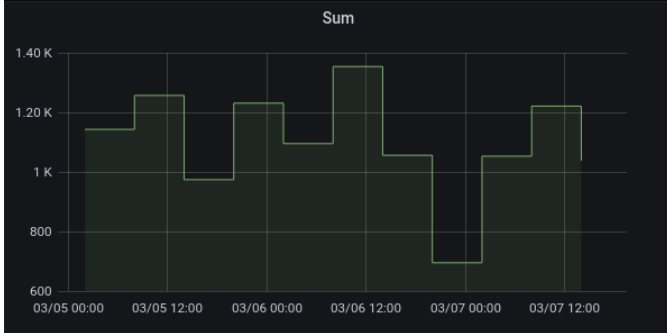


Fig. 18. Sum aggregation

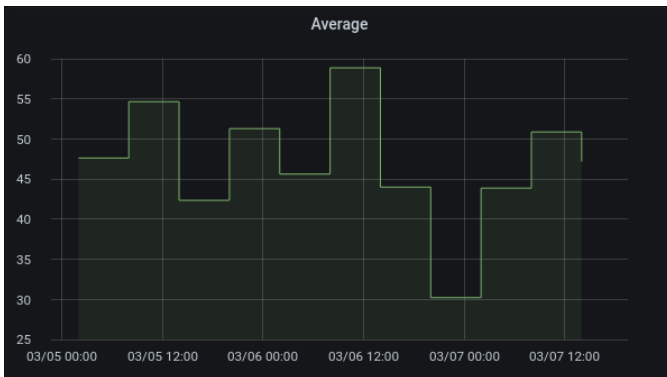


Fig. 19. Average aggregation

data, because each data point represents the beginning of a window, thus the span between two consecutive data points should be constant and equal to the first of the two..

We also had to visualise all the late events in a table. The corresponding table is shown below:

Late Events	
Time	
2022-03-04 22:53:44	96
2022-03-05 09:23:46	57
2022-03-05 16:08:47	7
2022-03-06 08:53:50	89
2022-03-07 01:38:53	18
2022-03-07 17:23:56	99
2022-03-07 19:23:56	98

Fig. 20. Late events

We can see that the difference between two consecutive late events varies wildly. On the low end, the difference between the last two events is only 2 hours, while on the high end, the difference between the 3rd and 4th events is a little under 17 hours. There appears to be no consistency to how much time is between two consecutive delays, however we expect that as the system lasts for longer the average time will be

$$30 * 15/60 = 7.5$$

hours.

## X. CONCLUSION

This paper provides a brief overview of a live streaming system. The Internet of Things is weaving a new worldwide web of interconnected objects. In this paper, we have presented an IoT technology stack that consists of multiple layers, including device hardware, connectivity, data management, applications and analytics. We introduced some tools that helped us in the process of creating the live streaming systems. Such tools are : ActiveMQ, ApacheFlink, Apache Cassandra and Grafana. We also described the steps we followed to install these tools as well as the problems we encountered when trying to use them. Last but not least, we provided the resulting charts and a brief overview of the shape of the charts.

## REFERENCES

- [1] Keiichi Yasumoto, Hirozumi Yamaguchi, Hiroshi Shigeno (2015) Survey of Real-time Processing Technologies of IoT Data Streams.
- [2] Sfar AR, Zied C, Challal Y. A systematic and cognitive vision for IoT security: a case study of military live simulation and security challenges. In: Proc. 2017 international conference on smart, monitored and controlled cities (SM2C), Sfax, Tunisia, 17–19 Feb. 2017.
- [3] Dmitry Namiot, Manfred sneps-snepp, Romass Pauliks (2018) On Data Stream Processing in IoT Applications
- [4] Fatih Gurcan, Muhammet Berigel (2018) Real-Time Processing of Big Data Streams: Lifecycle, Tools, Tasks, and Challenges

- [5] Official Apache ActiveMQ Website, <https://activemq.apache.org/getting-started>
- [6] Tutorial used for the installation of Apache ActiveMQ, <https://www.vultr.com/docs/install-apache-activemq-on-ubuntu-20-04/>
- [7] Tutorial used for the installation of Apache Flink, <https://data-flair.training/blogs/apache-flink-installation-on-ubuntu/>
- [8] Official Grafana site with instructions on how to install Cassandra, [https://cassandra.apache.org/doc/latest/cassandra/getting\\_started/installing.html](https://cassandra.apache.org/doc/latest/cassandra/getting_started/installing.html)
- [9] Official Grafana site with instructions on how to install Grafana, <https://grafana.com/docs/grafana/latest/installation/debian/>
- [10] Cassandra DataSource for Grafana, <https://github.com/HadesArchitect/GrafanaCassandraDatasource/releases/tag/2.0.0>
- [11] Cassandra DataSource for Grafana Demo, <https://github.com/HadesArchitect/GrafanaCassandraDatasource/wiki/Quick-Demo>
- [12] Ahmed El Hakim (2018), Internet of Things (IoT) System Architecture and Technologies, White Paper.