IoT Live Streaming

Real time data flow processing using open source distributed systems

Pelagia Drakopoulou
Electrical and Computer Engineering
National Technical University of Athens
03118027

Maria Nefeli Kondyli
Electrical and Computer Engineering
National Technical University of Athens
03118147

***Abstract*—The Internet of Things (IoT) has enabled the integration of physical devices with network connectivity, leading to the generation of large volumes of real-time data. In this paper, we present a prototype of a real-time data flow processing system that leverages open-source distributed systems to process and analyze data. In order to achieve this we firstly generate virtual live streaming data which correspond to various sensors that can be found in a smart home. The system utilizes Confluent Kafka as a message broker for data ingestion, Spark Streaming for data processing and aggregations, and TimescaleDB as a time-series database for data storage. We also use Grafana to display insights into the data through charts and tables, and web sockets for live streaming data. We also discuss the capability of an IoT system to process and analyze large volumes of data in real-time, providing valuable insights that can be used for informed decision-making in various applications.**

I. Introduction

This paper presents a real-time data flow processing system that constitutes the prototype of a real IoT system. The system is divided in layers which will be thoroughly analyzed. More specifically, we will demonstrate the flow of data starting from the device layer, continuing through the messaging broker in order to be processed in the live streaming layer and finally be stored in the data/storage layer or even presented in the presentation layer.

The code used for this project can be found in this github repo: https://github.com/peldrak/adis_streaming_project.git

II. Device layer

A. General information

In this layer we generate the virtual data that will be used in our live streaming system. The data correspond to real time value measurements from sensors that can be found in a smart home. The sensors that we use can be divided into three categories. The 15-minute sensors produce data every 15 minutes, starting from the datetime 2020-01-01 00:15, which is the same for all sensors. The 1-day sensors produce data at the end of each day at midnight and the data correspond to the total consumption (of water or energy) for each day. The random time sensor which produces data 5 random times in a day.

B. Generation model

In order to model the above, we have composed a python script that produces data every second, and each second corresponds to 15 minutes in the virtual data time. For example, for a 15-minute sensor, if a record with timestamp 2020-01-01 00:15 is being produced, after 1 second a record with timestamp 2020-01-01 00:30 will be generated. Respectively, for 1-day sensors, data is being generated every 96 seconds which correspond to 1 day in the virtual data timestamps. It is obvious that the above model is being followed in our prototype in order to save time, otherwise it would be unsustainable for testing purposes.

C. Sensors

The 10 sensors used in this project are listed below:

- TH1: Temperature 15-minute sensor with value range (12-35 ^{o}C)
- TH2: Temperature 15-minute sensors with value range (12-35 ^{o}C)
- HVAC1: A/C energy consumption 15-minute sensor with value range (0-100 Wh)
- HVAC2: A/C energy consumption 15-minute sensor with value range (0-200 Wh)
- MiAC1: Rest devices energy consumption 15-minute sensor with value range (0-150 Wh)
- MiAC2: Rest devices energy consumption 15-minute sensor with value range (0-200 Wh)
- W1: Water consumption 15-minute sensor with value range (0-1 lt)
- Etot: Total energy consumption 1-day sensor with value increase (2600x24 ± 1000 Wh)
- Wtot: Total water consumption 1-day sensor with value increase (110 ± 10 lt)
- Mov1: Random movement sensor which generates data with value 1, 5 random times each day

D. Late data

Specifically for the W1 sensor, we also generate some late data in order to include in the prototype the possibility of error in a sensor. Every 20 seconds apart from the correct record we also produce a record which corresponds to 2 days prior to the correct time. We do the same, every 120 seconds, by producing records corresponding to 10 days back. Later on,

in the live streaming layer we will have to manage these late events accordingly.

E. Connection with Kafka

The python script uses the KafkaProducer library in order to send the data directly to Kafka, the message broker in our system. Every sensor sends messages to its own topic in confluent kafka, and the whole process is executed in parallel. We use avro format for the data that we produce and we have specified a global schema for all sensors which consists of the following fields: 'name': 'string', 'timestamp': 'string', 'value': 'float'. We finally make sure to follow the message format that confluent supports for avro serialized data, by adding a magic byte and four bytes which signify the schema id, in the beginning of each message.

Fig. 1. Example of messages produced to TH1 topic.

Fig. 2. Example of messages produced to Etot topic.

III. Messaging Broker Layer

A. General information

The messaging broker layer is the main pillar of our system as it is responsible for the data flow between all open source systems that are being used in the project. At first, it receives the raw data from the device layer (producer) and distributes the data towards the live streaming layer and data/storage layer (consumers). We will see later on, that the broker also receives processed data from the live streaming layer in order to send the aggregated data to the data/storage layer. As an open source message broker we use Confluent Kafka which is a distributed streaming platform built on top of Apache Kafka, designed to help organizations manage and process real-time data streams at scale.

B. Scalability and fault tolerance

As we mentioned earlier, the scalability of IoT systems is vital due to the continuous increase in the data volume. We have to mention, here, that the confluent platform that we used has some restrictions in its free version, concerning the number of brokers that can be used in a cluster. However the purpose of this report is to demonstrate how IoT systems can be scalable and fault tolerant so we will not limit ourselves in the demonstration of the basic implementation. So, firstly we are going to describe the configuration of our system and later we will explain how we can make it better.

C. Comparison between basic and ideal configuration

Our implementation consists of a single-node kafka cluster with one broker which runs at localhost:9092. For every sensor we have created two topics, one for the raw data and one for the aggregated ones. We also have two extra topics for the late data. We have specified a partition in the topics equal to 1 as we did not find a specific reason to split the messages into partitions because we use different topics for everything. As our cluster consists of only one broker, the replication factor in our topics is equal to 1. It is obvious that this is the most basic implementation that we can do and is neither scalable nor fault tolerant. So, now we will discuss what we can do to improve it. First, we can use multiple machines, in order to have multiple clusters in our system. By doing this we ensure that if the volume of data increases significantly, then the work can be divided in different clusters and the execution time will not increase dramatically, as it would with only one cluster. The system then would be scalable. Subsequently, we can increase the number of brokers inside each cluster and therefore achieve a better QoS. Let's say we have 3 brokers in a cluster. Then we could specify the replication factor to be equal to 2. By doing this, the data of each topic is stored in 2 different brokers. Now, if one of the three brokers crashes, the messages won't be lost as all of them are stored in one more broker. Our system, then, would also be fault tolerant and a probable crash would not lead to data loss.

D. Schema Registry

The confluent platform also provides a Schema-Registry which runs at localhost:8081. We use it to post the schema of the messages we send to kafka, in order to be able to deserialize them. When we submit the schema for each topic, the schema registry returns its id. We use this id in our producer when we send the messages as we mentioned before.

E. Sink Connectors

In order to store both raw and aggregated data to the data/storage layer, we have installed a JDBC sink connector in our confluent platform. For every topic, we have configured a different sink connector which connects to timescaledb and autocreates the tables in which the data will be stored. The

connectors use io.confluent.connect.avro.AvroConverter as a value converter.

Fig. 3. Running connector.

IV. Live Streaming Layer

A. General information

Spark Structured Streaming is a scalable and fault-tolerant stream processing engine built on top of the Apache Spark Engine. It enables real-time processing of streaming data such as Kafka in our case. Structured Streaming processes the streaming data as a continuous, infinite table. The data is processed incrementally and in a fault-tolerant manner, meaning that if there is a failure, the system can automatically recover and resume processing from where it left off. Structured Streaming provides a unified API for batch and streaming data processing, which is built on top of SparkSQL.

B. Running Spark

We host spark locally and we start amaste and a worker in port 7077 as shown below:

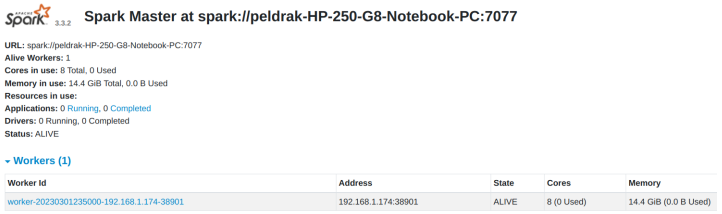

Fig. 4. Spark UI.

C. Connection with Kafka

We are using spark structured streaming in order to perform aggregations in the data and extract info and insights about the everyday behavior inside the smart home. As a language we are using PySpark. In the consumer.py file we specify the kafka broker to connect to via localhost:9092. More specifically, the spark application subscribes to the kafka topics and reads messages that are being produced to these topics. It then stores the messages in dynamically increasing dataframes and can therefore perform aggregations on these dataframes. More specifically, we first have to convert the data from avro with the pyspark.sql.avro.functions and then we create dataframes that consist of the message value. After we perform the aggregations, we convert back the data to avro and send them to different topics in kafka with the writeStream command.

D. Aggregations

The aggregations to be performed are the following:

- AggDay[x] - Day aggregations for all 15-minute sensors: The aggregation method differs from sensor to sensor. Temperature sensors (TH1, TH2) are grouped by day and the aggregation method used is average. AC energy consumption sensors (HVAC1, HVAC2), other devices energy consumption sensors (MiAC1, MiAC2) and water sensor (W1) are also grouped by day and the aggregation method used is sum. Sensors measuring cumulative energy (Etot) and cumulative water consumption (Wtot) are also grouped by day, but in this case only in order to maintain the date and the aggregation method used is max.
- AggDayDiff[y] - Aggregations for all 1-day sensors that show the value difference between subsequent days:
  For this aggregation we create two new dataframes for each 1-day sensor that the value field contains the subtraction of the previous day entry from the current day entry.
- AggDayRest – Leakage aggregations between 1-day sensors and 15-minute ones:
  To calculate the leakage we first calculate the AggDayDiff for the 1-day sensors and then we subtract the sum of the AggDay of the 15-minute sensors. For energy leakage we calculate:
  AggDayRest = AggDayDiff[Etot] - AggDay[HVAC1] - AggDay[HVAC2] - AggDay[MiAC1] - AggDay[MiAC2]
  For water consumption leakage we calculate:
  AggDayRest = AggDayDiff[Wto] – AggDay[W1]

V. Data/Storage Layer

A. General information

TimescaleDB is an open-source relational database that is designed to handle large-scale time-series data. It is built as an extension to the popular PostgreSQL database, which means it can leverage the many features and tools available in PostgreSQL while also offering additional functionality specific to time-series data.

B. Tables

We are using TimescaleDB cloud and we have created a service in which ce can connect via cli by providing its credentials. The jdbc connector in kafka autocreates tables in timescale so there is no need for us to create manually the tables. In the db we store both the raw and aggregated data.

Therefore, each sensor has two specific tables in timescale, one for the raw and one for the aggregated data. We also have two additional tables for the 2-day and 10-day late data for W1 sensor.

C. Connection with the presentation layer

TimescaleDB also acts like a data source for grafana. By connecting timescale and grafana, the presentation layer has access to all the tables in our service. Also note that the size of the tables increases while more data records are being produced.

VI. Presentation Layer

A. General information

Grafana is an open-source analytics and visualization platform that is designed to help users easily analyze and visualize large amounts of data. One of the key features of Grafana is its flexible and customizable dashboard system, which allows users to create interactive, real-time dashboards to visualize their data. Grafana is commonly used in applications that require real-time monitoring and analysis of data such as IoT device management. In order to get real-time data there are websockets that allows our application to create secure connections with Grafana. WebSockets deliver real-time data without polling, when you're using the WebSockets data source plugin, Grafana panels will automatically update as new data comes in. This makes live monitoring easy, whether you're watching for a continuous stream of data or immediately want to see the results of alert data that arrives infrequently.

B. Dashboard setup

For our project we create a new dashboard that will contain charts and tables for all the sensors of the IoT system. For each sensor we create two panels. The first one represent the entries in our database in a table. The second panel is Time Series chart that represent the value of each entry in time. To do so we select the database and the table that we want the information from and we write the SQL query that returns the specified data. In the upper right side of the dashboard there is a datepicker that allows the user to get the data for specified dates. There is also a refresh button to get the latest entries in the database.

Some of the tables and charts for the sensors are shown below:

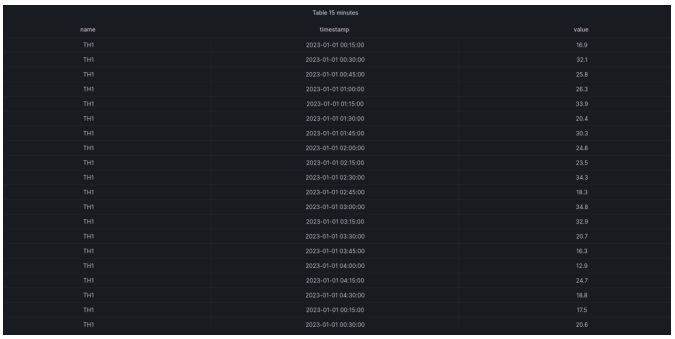

Fig. 5. Table for TH1.

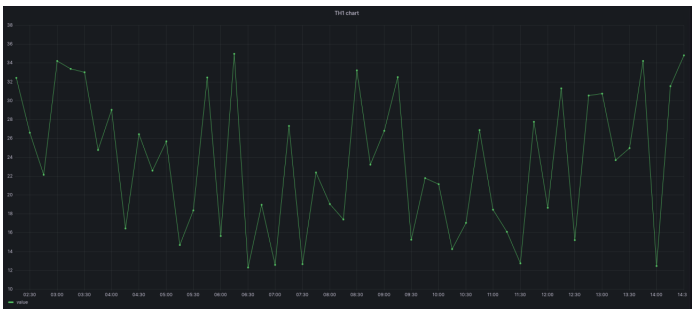

Fig. 6. Chart for TH1.

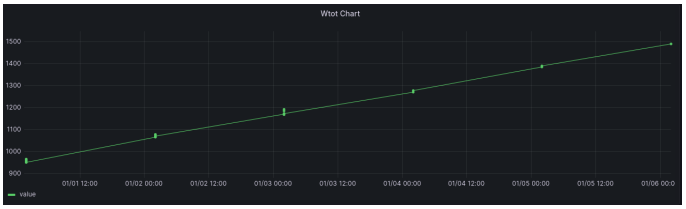

Fig. 7. Chart for Wtot.

VII. Conclusion

In conclusion, the real-time data flow processing system presented in this paper serves as a prototype for a complete IoT system. The layered architecture of the system has been thoroughly analyzed, demonstrating the flow of data from the device layer to the presentation layer. The system showcases the importance of a robust messaging broker and live streaming layer for efficient data processing and storage. The presented prototype can be further developed and customized to suit the specific needs of different IoT applications. Future work can also focus on optimizing the system's performance by exploring different data processing techniques and algorithms.

References

[1] https://kafka.apache.org/
[2] https://spark.apache.org/docs/latest/index.html
[3] https://docs.timescale.com/
[4] https://grafana.com/
[5] https://www.confluent.io/blog/converters
[6] https://spark.apache.org/docs/structured-streaming-kafka-integration.html
[7] https://towardsdatascience.com/kafka-python-explained
[8] https://sagarkudu.medium.com/set-up-a-kafka-cluster-in-local-with-multiple-kafka-brokers
[9] https://github.com/indiacloudtv/pyspark-structured-streaming
[10] https://www.databricks.com/blog/event-time-aggregation
[11] https://www.timescale.com/blog/create-a-data-pipeline
[12] https://www.scaleway.com/en/docs/tutorials/visualize
[13] https://www.confluent.io/hub/confluentinc/kafka-connect-jdbc
[14] https://docs.confluent.io/kafka-connectors/self-managed/userguide
[15] https://www.confluent.io/blog/consume-avro-data-from-kafka-topics