Sensors data analyzer

Documentation

Authors:
Katarzyna Boczek 117317
Agnieszka Szymborska 117201

Table of Contents

Problem description

Introduction

Nowadays the subject of smart homes, areas and cities is more and more popular and becoming more and more important. Automatic control of light, heating, opening window, alarm systems, etc. is very useful for many people because it can save some energy, money and time. For example, it can help to reduce the loss of natural resources. Smart home management systems use wireless communication standards such as Wi-Fi, ZigBee or GSM. New standards appear on the market every now and then. They often offer larger ranges at lower costs. One of those standards is a LoRaWAN standard (Low Power Wide Area Network). It uses a LoRa protocol that offers a range of up to 5 kilometers in the city and 15 kilometers in open space.

Base project

Project described in this document is an extension of an intelligent sensors network in the master-slave architecture. It uses LoRaWAN standard to send and receive temperature and humidity data. That project was realized by three students of Poznań University of Technology (Bartosz Koperski, Michał Panowicz and Agnieszka Szymborska, supervisor: Mariusz Nowak PhD) from March 2016 to January 2017. In the created system, sensors (slave devices) send values of temperature and humidity measurements to the parent device (master device) using the radio protocol (range up to 50 meters). Then, the master device sends data received from sensors to the LoRaWAN gateway (LORANK8) using the LoRaWAN standard. Data received by the gateway are stored in TTN (The Things Network) server. They have to be saved in the local database so that they can be used later for online and offline analysis. The TTN server sends that data to a special topic from where they are read by the MQTT client that works in publish-subscriber pattern. The MQTT client is a script written in Python language that puts read data to the local database. Finally, they are displayed to the user on the chart. The whole process is shown on figure 1.

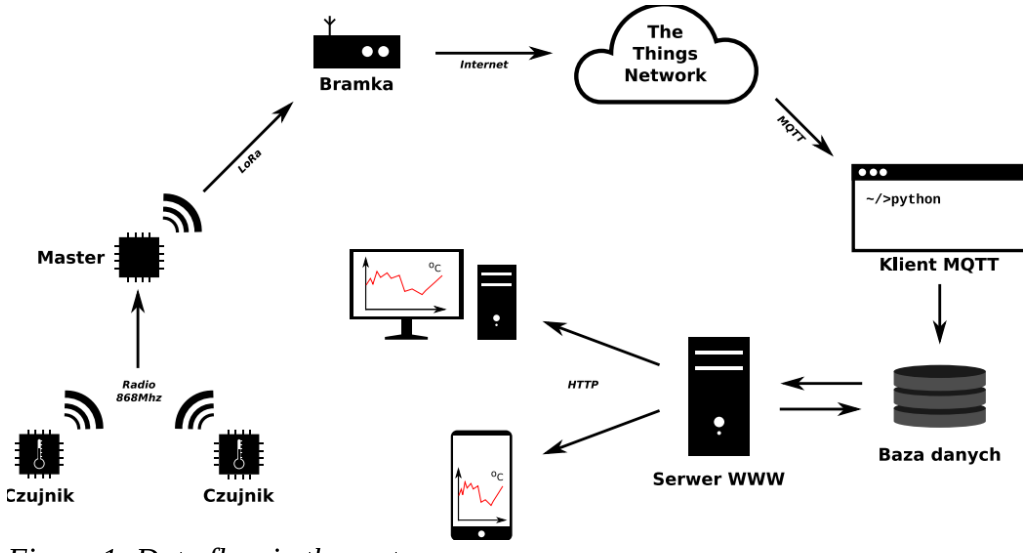

Figure 1: Data flow in the system

Assumed extensions

The project described here was realized during Data Warehouses and Analytical Processing course led by Robert Wrembel Assoc. Prof., PhD DSc on Poznań University of Technology. The main objectives of the project were:

- to design online processing data system architecture using external tools such as Apache Kafka and Apache Spark,
- to implement the system that sends information to the devices if the temperature in the room is too high or too low. This is important from the point of view of heating that room. If the temperature is too high the radiator should be turned off and if the temperature is too low it should be turned on,
- to implement the system that detects patterns in incoming events (sensors measurements).

System architecture

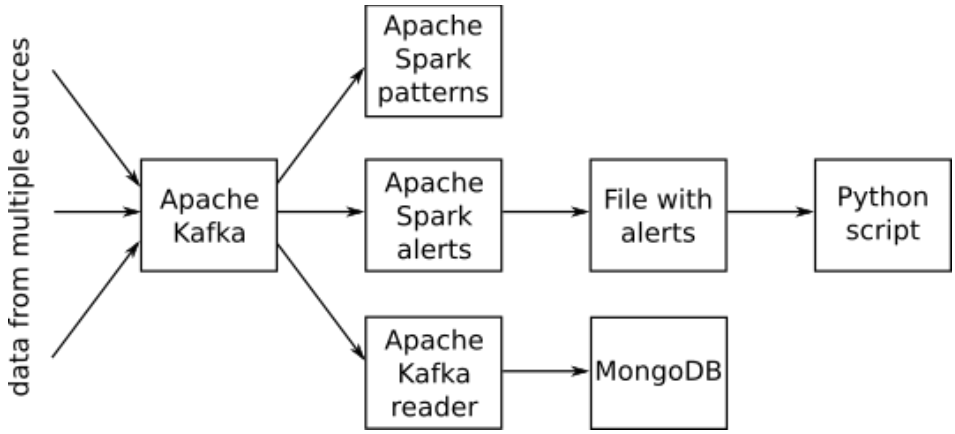

Figure 2: System architecture

Apache Spark version: 2.1.0
Apache Kafka version: 0.10.2
Python version: 3.5

Environment setup (Ubuntu 16.04 LTS)

To work with Apache Kafka and Apache Spark you will need Java JDK. If you don't have it you can download it from Oracle site here.

Python package installation

- pip3 install kafka-python

Apache Kafka installation

Firstly, you need to install ZooKeeper – an open source service for maintaining configuration information, providing distributed synchronization, naming and providing group services.

- sudo apt-get install zookeeperd

After installation, it will be started automatically on port 2181 by default. You can check it with the following command:

- netstat -ant | grep :2181

If the result looks like this:

- tcp6 0 0 :::2181 :::* LISTEN

The installation is valid.

The next step is to download Apache Kafka from the website here and to create folder /opt/Kafka for it:

- sudo mkdir /opt/Kafka

Now, you have to move uncompressed downloaded Kafka to created folder /opt/Kafka:

- mv -r ~/Download/<kafka_folder>/* /opt/Kafka/

Run server (from home folder level):

- sudo /opt/Kafka/bin/kafka-server-start.sh /opt/Kafka/config/server.properties

Create topic for Kafka:

- sudo /opt/Kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic <topic_name>

Expected result:

- Created topic <topic_name>

Apache Spark installation

Firstly, you have to install Spark's Simple Build Tool. Execute following commands:

- echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list
- sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 –recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
- sudo apt-get update
- sudo apt-get install sbt
- sudo apt-get install pandoc

Now, download Apache Spark from the website here. Follow the instructions below to uncompress package:

- tar xzvf <downloaded_package_name>
- mv <uncompressed_folder_name> /spark
- sudo mv spark/ /usr/lib/
- export SPARK_HOME=/usr/lib/spark
- export PATH=$SPARK_HOME/bin:$PATH

The next step is to configure Apache Spark. Go to configuration folder:

- cd /usr/lib/spark/python

and execute the following command:

- python3 setup.py install

Then, create conf file from template in /usr/lib/spark/conf folder:

- cp spark-env.sh.template spark-env.sh

Open the created file:

- nano spark-env.sh

Add the following lines to that file:

- JAVA_HOME=<path_to_jvm> (e. g. /usr/lib/jvm/java-8-oracle)
  SPARK_WORKER_MEMORY=4g

Create log4j.properties file from template in /usr/lib/spark/conf folder:

- cp log4j.properties.template log4j.properties

Open the created file:

- nano log4j.properties

Add following lines to that file:

- log4j.rootCategory=ERROR, console

Check configuration by executing the following command in /usr/lib/spark/bin folder:

- ./pyspark

MongoDB installation

Firstly, you have to import the key of MongoDB Repository:

- sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927

Create list file for MongoDB:

- echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list

Execute command:

- sudo apt-get update

Install MongoDB package:

- sudo apt-get install -y mongodb-org

Create file to manage the MongoDB service:

- sudo nano /etc/systemd/system/mongodb.service

Add the following content to the created file:

- [Unit]
  Description=High-performance, schema-free document-oriented database
  After=network.target
  [Service]
  User=mongodb
  ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf
  [Install]
  WantedBy=multi-user.target

Finally, check installation:

- sudo systemctl start mongodb
- sudo systemctl status mongodb

Project implementation

Alert system

The first aim of the project was to create an alarm system. Its main task is to detect the deviations from the accepted norm taking the set thresholds into account. For example it can detect too high or too low temperatures in the room. We used the Apache Spark to implement that system. A script reads the incoming data about the current temperature in the room from the Apache Kafka topic. Then, a fixed number of messages is stored in the window. The data collected in that window are grouped by the sensor identifiers from which they are derived. In the next step, for each of that groups the average value of temperature is calculated based on the data in the window. Calculated average values are compared to the set norm. If too much deviation is detected, a special message is generated and saved to the specified file. The actuator can read that message and take the appropriate action such as turning the radiator in or off.

Here is our script code:

```
# alert_system.py

from __future__ import print_function

import sys
import json
from os import mkfifo, path

from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

TIME_BETWEEN_READING = 1
BATCHES_IN_WINDOW = 3

TEMP_WANTED = 30
TEMP_DELTA = 0.5

FIFO_PATH = '/tmp/myfifo'

def write_partition(iterator):
      elements = list(iterator)
      if len(elements) > 0:
            print(elements[0])

def main(fifo):
      sc = SparkContext(appName="SensorDataAnalyzer")
      ssc = StreamingContext(sc, TIME_BETWEEN_READING)

      kafkaStream = KafkaUtils.createDirectStream(
            ssc,
            ["testing"],
            {"metadata.broker.list": "localhost:9092"}
      )

      # map each row from the batch to get only the json object
      lines_this_batch = kafkaStream.map(lambda x: json.loads(x[1]))

      # format received json object to print it on the screen
      formated = lines_this_batch.map(
            lambda x: '%s: Key: %s, Value: %s' % (x['time'], x['key'], x['value'])
      )
      # formated.pprint()

      # create window for last three batches
      window = lines_this_batch.window(BATCHES_IN_WINDOW * TIME_BETWEEN_READING)

      # map each element from window as a touple (key, value) where the value is also
      # a touple (sensorID, (temperature_measurement, 1))
      avg_of_window = window.map(lambda x : (x['key'], (x['value'], 1)))

      # group created touples by the key and sum up the corresponding fields
      # from the value
      avg_of_window = avg_of_window.reduceByKey(
            lambda x, y : (x[0] + y[0], x[1] + y[1])
```

```
	)

	# map each element and calculate the average temperature
	avg_of_window = avg_of_window.map(
		lambda (key, (elements_sum, elements_count)) : (
			key,
			elements_sum,
			elements_count,
			elements_sum / float(elements_count)
		)
	)

	# print the average
	#avg_of_window.pprint()

	# search too high average
	too_hot = avg_of_window.filter(lambda x : x[3] > TEMP_WANTED + TEMP_DELTA)
		.map(lambda x : '%s|TOO_HOT' % x[0])
	#too_hot.pprint()

	# search too low average
	too_cold = avg_of_window.filter(lambda x : x[3] < TEMP_WANTED – TEMP_DELTA)
		.map(lambda x : '%s|TOO_COLD' % x[0])
	#too_cold.pprint()

	to_print = ssc.union(too_hot, too_cold)
	#to_print.pprint()

	to_print.foreachRDD(lambda rdd: rdd.foreachPartition(write_partition))

	ssc.start()
	ssc.awaitTermination()

if __name__ == "__main__":
	if not path.exists(FIFO_PATH):
		mkfifo(FIFO_PATH)
	with open(FIFO_PATH, 'r+', 0) as fifo:
		try:
			#fifo.write('Starting\n')
			main(fifo)
		except KeyboardInterrupt:
			#fifo.write('Ending\n')
			print("\nThe End")
		fifo.close()
```

You can run it using the following command from home folder level:

```
sudo /usr/lib/spark/bin/spark-submit --master local[2] --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0 alert_system.py 2>&1
```

Pattern system

Another purpose of the project was to write a script to detect patterns in the incoming data. Just like before this task was also done using Apache Spark. Each message received from the Apache Kafka stream is inserted to the window (implemented as a list). When the window is full the values in it are compared with one another to see if they fit to the wanted pattern.

Here is our script code:

```
# patterns_system.py

from __future__ import print_function

import sys
import json

from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

TIME_BETWEEN_READING = 1

WANTED_PATTERN = []

WINDOW = []

def checkPattern():
	patternFound = True

	for i in range(len(WANTED_PATTERN)):
		if WINDOW[i + 1] - WINDOW[i] <= 0 and WANTED_PATTERN[i] == 'u':
			patternFound = False
			break;
		if WINDOW[i + 1] - WINDOW[i] >= 0 and WANTED_PATTERN[i] == 'd':
			patternFound = False
			break;
		if WINDOW[i + 1] - WINDOW[i] != 0 and WANTED_PATTERN[i] == 's':
			patternFound = False
			break;

	if patternFound == True:
		print("Znaleziono wzorzec! ", WINDOW)

def insertToWindow(rdd):
	global WINDOW

	for element in rdd.collect():
		# add element to the window
		WINDOW.append(element['value'])

		# if window is too long, delete the oldest element
		if len(WINDOW) > (len(WANTED_PATTERN) + 1):
			WINDOW = WINDOW[1:]

		# if window is full, check if it corresponds to the pattern
		if len(WINDOW) == (len(WANTED_PATTERN) + 1):
			checkPattern()

def main():
	sc = SparkContext(appName="SensorDataAnalyzer")
	ssc = StreamingContext(sc, TIME_BETWEEN_READING)

	kafkaStream = KafkaUtils.createDirectStream(
		ssc,
		["testing"],
		{"metadata.broker.list": "localhost:9092"}
	)

	# map each row from the batch to get only the json object
	lines_this_batch = kafkaStream.map(lambda x: json.loads(x[1]))

	lines_this_batch.foreachRDD(insertToWindow)

	ssc.start()
	ssc.awaitTermination()
```

```
def encodeWantedPattern():
	global WANTED_PATTERN

	directionChanges = sys.argv[1].split('-')

	for change in directionChanges:
		direction = change[0]
		for _ in range(int(change[1:])):
			WANTED_PATTERN.append(direction)

if __name__ == "__main__":
	try:
		encodeWantedPattern()
		print(WANTED_PATTERN)
		main()
	except KeyboardInterrupt:
		print("\nThe End")
```

You can run it using the following command from home folder level:

```
sudo /usr/lib/spark/bin/spark-submit --master local[2] --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0 patterns_system.py <pattern>
```

where <pattern> is a pattern which you want to find. It must have the following structure:

```
<type><number>- … -<type><number>
```

<type> is a letter that represents the type of trend (d – down (lower temperature), u – up (higher temperature) or s – stable (the same temperature)). <number> is a number of measurements in the trend. For example a wanted pattern can look like this: d2-u2-d1 and the sample values that corresponds to that pattern are: 20, 19, 18, 19, 20, 19.

Summary

Although we had a lot of problems with the configuration of new tools that we had never used before, the main objectives of the project had been realized. Apache Kafka and Apache Spark are not easy to get started because their documentation is incomplete but they are still very useful tools. They let us create analyzing system with very little effort in writing code. Apache Kafka is very helpful when we have to integrate data from multiple sources into one stream. If we did this project again we would choose another programming language, e. g. Scala or Java because there is more tutorials and examples for these languages. In addition, Apache Spark does not longer support Python. We hope that our system will improve intelligent sensors system created before.