VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS (CO2039)

Assignment

Extended Modules for Talented Engineer

Advisor: Nguyen Quang Hung
Students: Nguyen Thanh Nhan - 2012522.
Tran Ha Tuan Kiet- 2011493.
Nguyen Duc Thuy - 2012158.
Le Xuan Huy - 2011266.
Dang Dang Khanh - 2011383.
Ho Truong Duc Tien - 2012196.

HO CHI MINH CITY, March, 2022

Contents

Chapter 1

Member list & Workload

No.	Fullname	Student ID	Workload	Percentage of work
1	Nguyen Thanh Nhan	2012522	- Team Leader - Research MQTT, Thingsboard - Create sample sensors - Deploy ThingsBoard	100%
2	Nguyen Duc Thuy	2012158	- Set up cluster - Manage all demos - Write report, presentation	100%
3	Le Xuan Huy	2012522	- Writing report - Research Apache Spark - Write Spark Job codes	100%
4	Tran Ha Tuan Kiet	2011493	- Research Apache Kafka - Build Kakfa component	100%
5	Dang Dang Khanh	2011383	- Research Nginx - Write sensor's codes	100%
6	Ho Truong Duc Tien	2012196	- Research ThingsBoard and Kafka - Write report	80%
7	Cu Thanh Bang	2012682	- Research HDFS - Write report	20%

Chapter 2

Necessary knowledge

2.1 MQTT

2.1.1 Overview

MQTT is a light-weight, open and simple publish/subscribe **messaging transport protocol**[1]

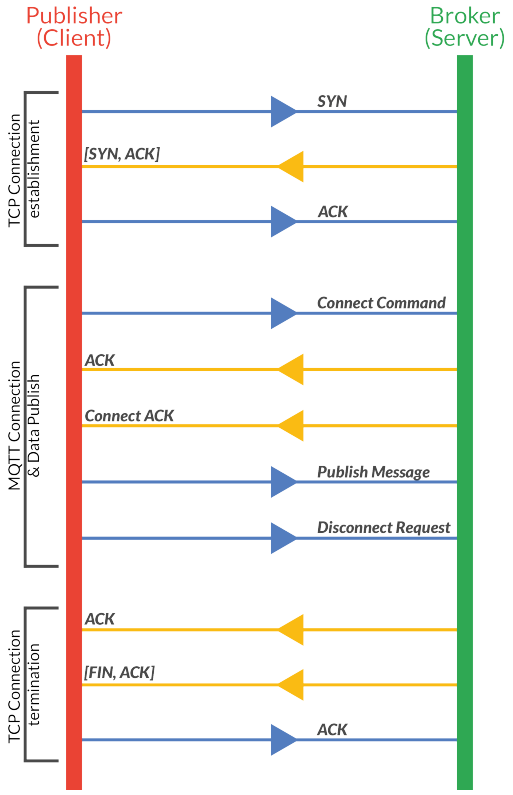


Figure 2.1: MQTT Protocol

2.1.2 Message Types

2.1.2.1 CONNECT

The MQTT client initiates the connection by sending a CONNECT message to the broker. To initiate a connection, the client sends a command message to the broker. If this CONNECT message is malformed (according to the

[1]Some MQTT definitions also mention MQTT message broker. But in this project, we only care about the network protocol part.

MQTT specification) or too much time passes between opening a network socket and sending the connect message, the broker closes the connection.

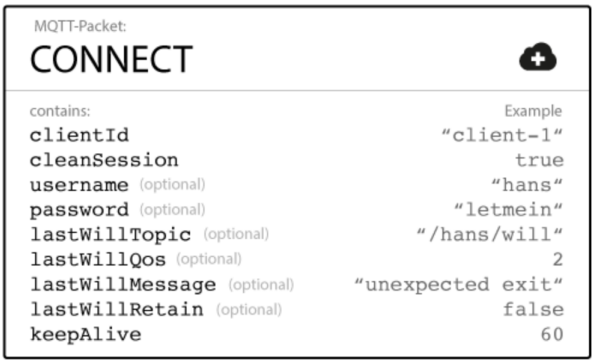


Figure 2.2: MQTT CONNECT example

2.1.2.2 CONNACK

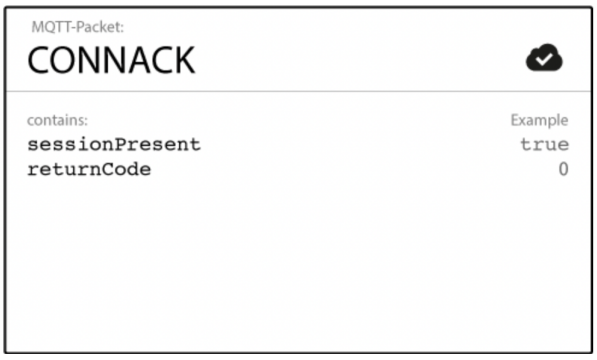


Figure 2.3: CONNACK Example

When a broker receives a CONNECT message, it is obligated to respond with a CONNACK message. CONNACK message include status code. Here are the return codes at a glance:

Return Code	Return Code Response
0	Connection accepted
1	Connection refused, unacceptable protocol version
2	Connection refused, identifier rejected
3	Connection refused, server unavailable
4	Connection refused, bad user name or password
5	Connection refused, not authorized

2.1.2.3 PUBLISH

A PUBLISH message in MQTT has several attributes that help MQTT client publish messages. PUBLISH contain a topic that the broker can use to forward the message to interested clients. Typically, each message has a payload which contains the data to transmit in byte format.

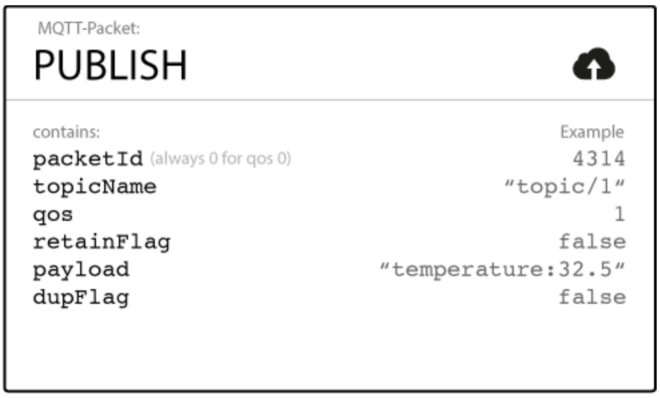


Figure 2.4: MQTT PUBLISH example

2.1.2.4 PUBACK (Publish acknownledged)

PUBACK is a message that the receiver send back to publisher to acknowledges receipt of the message when a receiver gets a message with QoS 1.

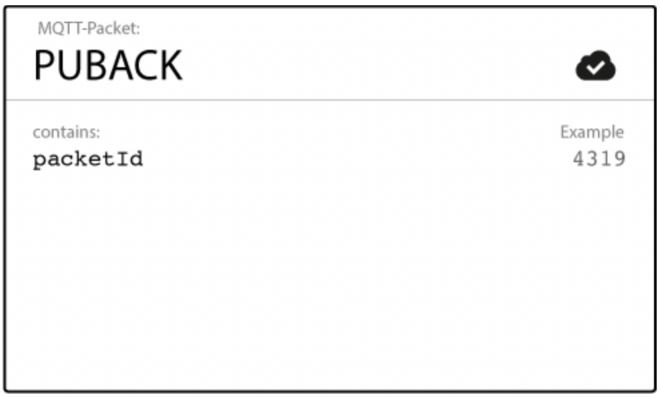


Figure 2.5: MQTT PUBACK example

2.1.3 Quality of Service (QoS)

2.1.3.1 QoS 0 - at most once

The minimal QoS level is zero. This service level guarantees a best-effort delivery. There is no guarantee of delivery. The recipient does not acknowledge receipt of the message and the message is not stored and re-transmitted by the sender. QoS level 0 is often called "fire and forget" and provides the same guarantee as the underlying TCP protocol.

2.1.3.2 QoS 1 - at least once

QoS level 1 guarantees that a message is delivered at least one time to the receiver. The sender stores the message until it gets a PUBACK packet from the receiver that acknowledges receipt of the message. It is possible for a message to be sent or delivered multiple times.

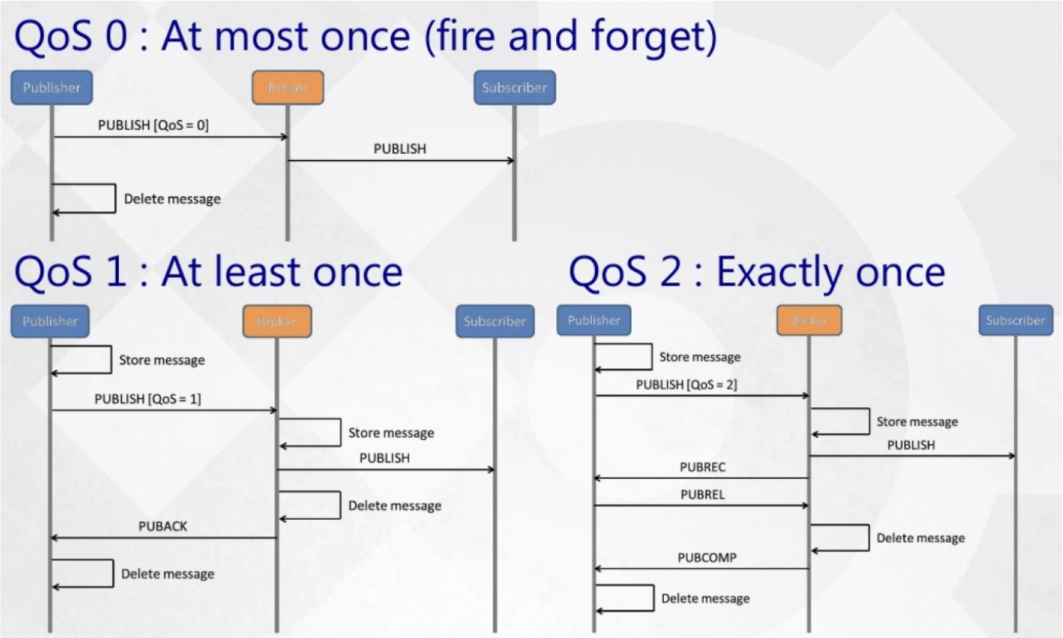


Figure 2.6: QOS

2.1.3.3 QoS 2 - exactly once

QoS 2 is the highest level of service in MQTT. This level guarantees that each message is received only once by the intended recipients. QoS 2 is the safest and slowest quality of service level. The guarantee is provided by at least two request/response flows (a four-part handshake) between the sender and the receiver. The sender and receiver use the packet identifier of the original PUBLISH message to coordinate delivery of the message.

2.1.4 Persistent Session

To receive messages from an MQTT broker, a client connects to the broker and creates subscriptions to the topics in which it is interested. If the connection between the client and broker is interrupted during a non-persistent session, these topics are lost and the client needs to subscribe again on reconnect. Re-subscribing every time the connection is interrupted is a burden for constrained clients with limited resources. To avoid this problem, the client can request a persistent session when it connects to the broker. Persistent sessions save all information that is relevant for the client on the broker. The clientId that the client provides when it establishes connection to the broker identifies the session.

In a persistent session, the broker stores the following information (even if the client is offline). When the client reconnects the information is available immediately.

- Existence of a session
- All subscriptions
- All messages with QoS 1 and 2 that client not yet confirmed
- All messages with QoS 1 and 2 that client missed
- All messages with QoS 2 that not yet acknownledged

2.1.5 MQTT vs HTTP

2.1.5.1 HTTP protocol

HTTP (Hypertext Transfer Protocol) is responsible for the action that a server has to take while sending information over the network. When a URL is being entered into the browser, this protocol sends an HTTP request to the server and then an HTTP response is sent back to the browser. This protocol is also responsible for the controlling of webpages on the World Wide Web for their formatting and representation.

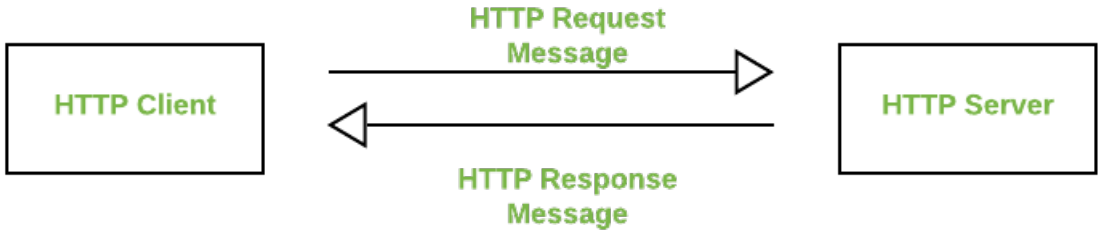


Figure 2.7: HTTP Architecture

2.1.5.2 Difference between MQTT and HTTP protocols

According to Wang, HTTP has an advantage of smaller overhead time when making the initial connection setup. However, the real advantage of MQTT when comparing with HTTP is that we can reuse the single connection for multiple messages [1]. As a result, MQTT protocol should be used when sending messages from IoT sensors, while HTTP should be used for less-intense message transmission (smartphones, PCs, ...)

2.2 Thingsboard

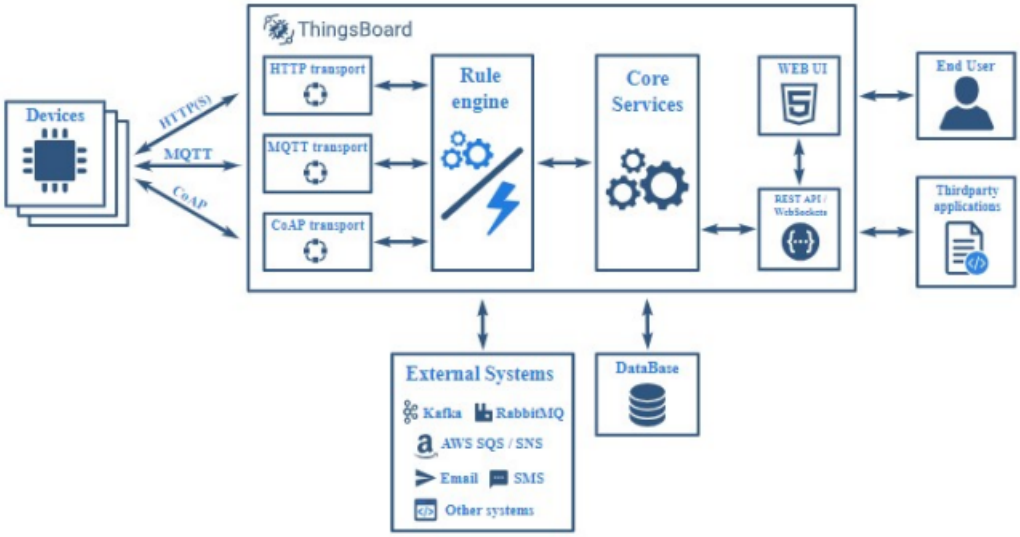


Figure 2.8: Thingsboard model
Source: [2]

2.2.1 Overview

Thingsboard is an open IoT platform. It enables rapid development, management and scaling of IoT projects. With the Thingsboard platform you can collect, process, visualize and manage devices.

Thingsboard allows to connect devices via industry standard IoT protocols - MQTT, CoAP and HTTP, supporting both cloud and on-premises deployments.

ThingsBoard also allows integrating of connected devices with legacy and third-party systems using existing protocols. Connect to OPC-UA server, MQTT broker, Sigfox Backend or Modbus slaves in minutes by connecting via IoT Gateway.

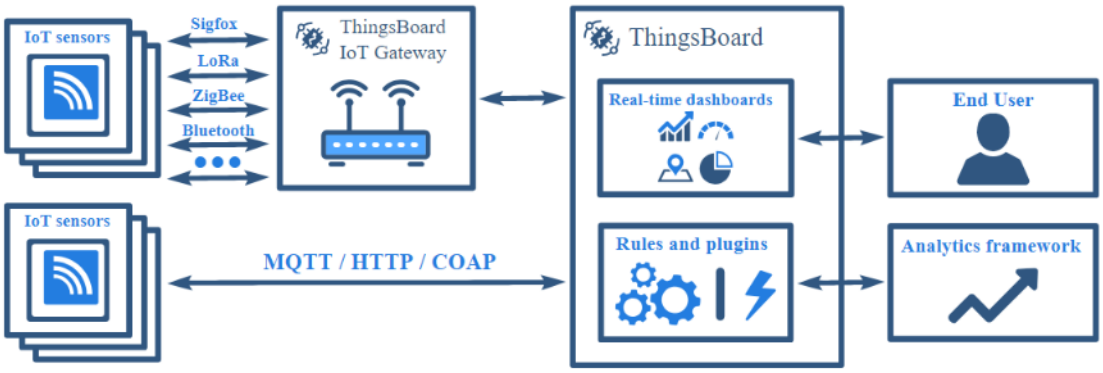


Figure 2.9: ThingsBoard IoT Gateway
Source: [3]

ThingsBoard enables user to create rich IoT Dashboards to display data and control devices remotely in real time.

ThingsBoard also allows you to create complex Rule Chains to process data from your device and tailored to

Figure 2.10: ThingsBoard Dashboard
Source: [4]

your specific application use cases.

2.2.2 ThingsBoard's Features and Architecture

ThingsBoard provides many convenient features. There are:

- Provision devices, assets and customers, and define relations between them.
- Collect and visualize data from devices and assets.
- Analyze incoming telemetry and trigger alarms with complex event processing.
- Control your devices using remote procedure calls (RPC).

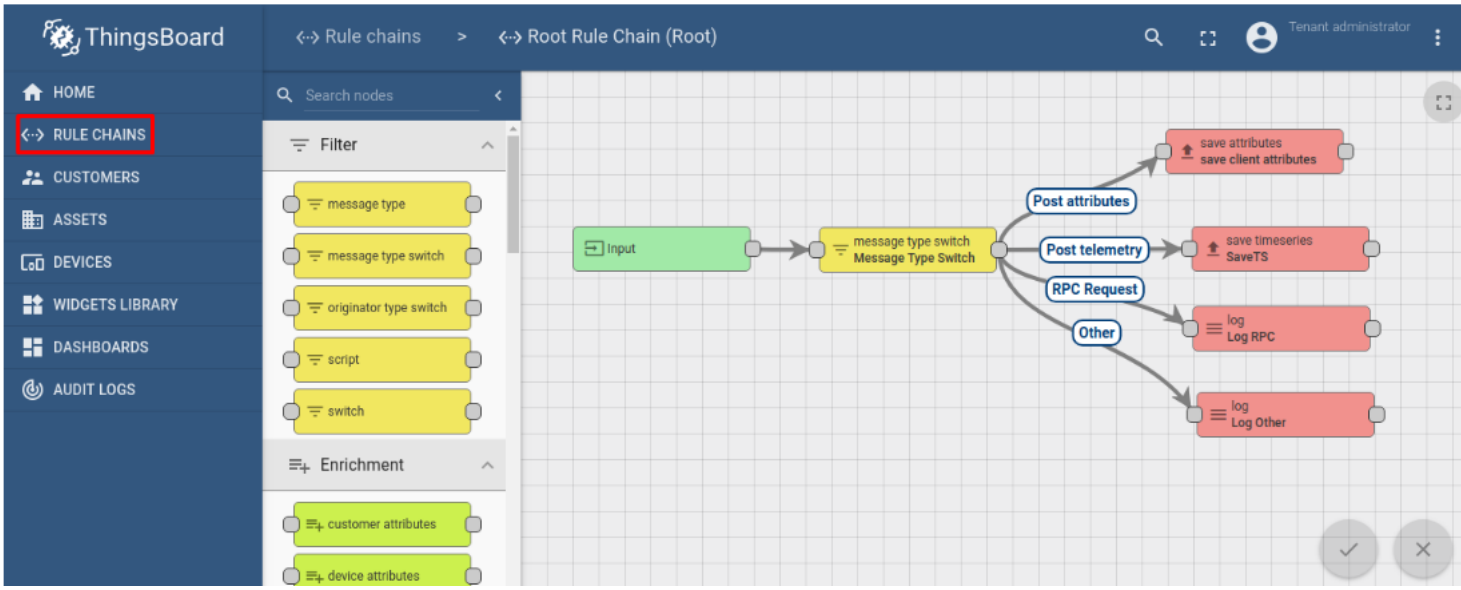


Figure 2.11: ThingsBoard Rule Chain
Source: [5]

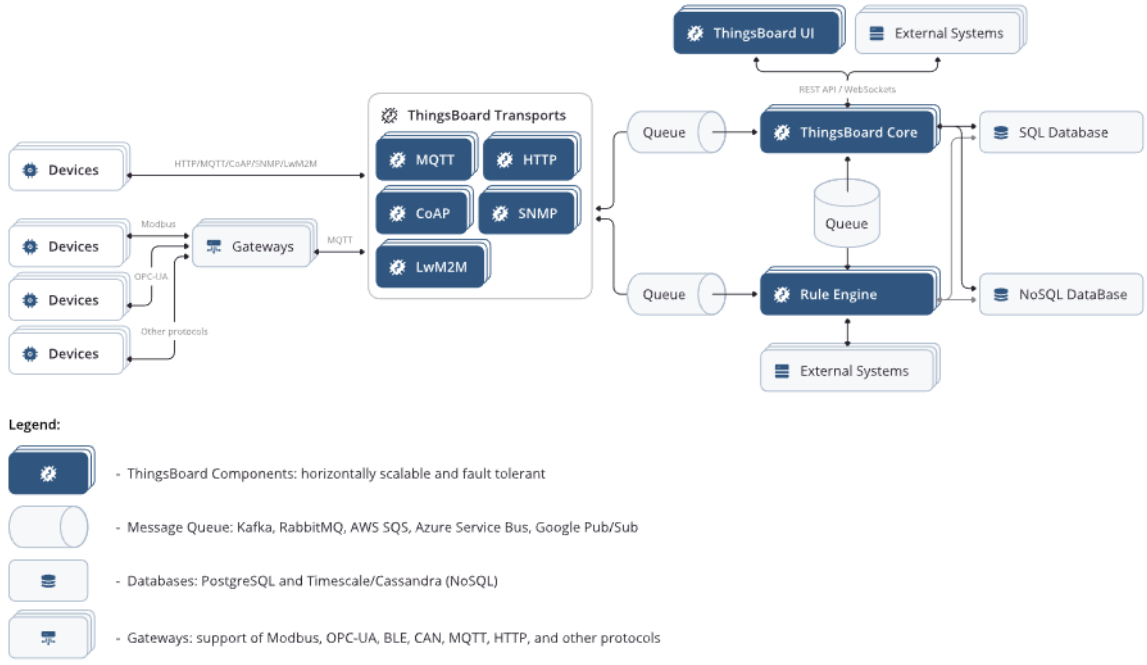


Figure 2.12: Overview ThingsBoard's features
Source: [6]

- Build work-flows based on a device life-cycle event, REST API event, RPC request, etc.
- Design dynamic and responsive dashboards and present device or asset telemetry and insights to your customers.
- Enable use-case specific features using customizable rule chains.
- Push device data to other systems.

About the architecture, ThingsBoard is designed to be:

- **Scalable:** the horizontally scalable platform, built using leading open-source technologies.

- **Fault-tolerant:** no single-point-of-failure, every node in the cluster is identical.
- **Robust and efficient:** a single server node can handle tens or even hundreds of thousands of devices, depending on the use-case. ThingsBoard cluster can handle millions of devices.
- **Customizable:** adding new functionality is easy with customizable widgets and rule engine nodes.
- **Durable:** never lose your data.

2.2.3 Key concepts in ThingsBoard

ThingsBoard provides the user interface and REST APIs to provision and manage multiple entity types and their relations in your IoT application. In ThingsBoard, you will work with supported entities:

- **Tenants:** is an individual or an organization who owns or produce devices and assets; Tenant may have multiple tenant administrator users and millions of customers, devices and assets.
- **Customer:** is also a separate business-entity: individual or organization who purchase or uses tenant devices and/or assets; Customer may have multiple users and millions of devices and/or assets.
- **Users** are able to browse dashboards and manage entities.
- **Devices:** basic IoT entities that may produce telemetry data and handle RPC commands. For example, sensors, actuators, switches;
- **Assets:** abstract IoT entities that may be related to other devices and assets. For example factory, field, vehicle.
- **Entity views:** are useful if you like to share only part of device or asset data to the customers.
- **Alarms:** are events that identify issues with your assets, devices, or other entities.
- **Dashboards:** visualization of your IoT data and ability to control particular devices through the user interface.
- **Rule Nodes:** processing units for incoming messages, entity lifecycle events, etc.
- **Rule Chain:** defines the flow of the processing in the Rule Engine. May contain many rule nodes and links to other rule chains.

Each entity supports:

- **Attributes** - static and semi-static key-value pairs associated with entities.
- **Time-series data** - time-series data points available for storage, querying and visualization.
- **Relations** - directed connections to other entities.

Some entities support profiles:

- **Tenant Profiles** - contains common settings for multiple tenants: entity, API and rate limits, etc. Each Tenant has the one and only profile at a single point in time.
- **Device Profiles** - contains common settings for multiple devices: processing and transport configuration, etc. Each Device has the one and only profile at a single point in time.

2.2.4 The role of ThingsBoard in our system

In our system, ThingsBoard rule engine supports basic analysis of incoming telemetry data. The idea behind rule engine is to provide functionality to route data from IoT Devices to different plugins, based on device attributes or the data itself.

There are two main role of ThingsBoard in this system:

- is a bridge between protocols (MQTT, CoAP, HTTP) to Kafka.
- makes it easier to manage devices (sensors).

2.3 Apache Kafka

2.3.1 Overview

Apache Kafka is an event streaming platform, used to build data pipelines between different systems.

2.3.2 Publish/subscribe messaging

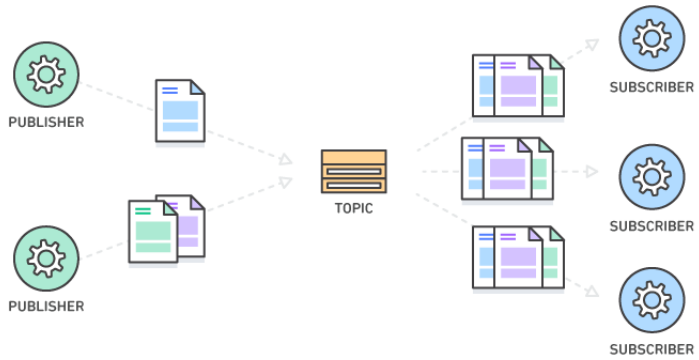


Figure 2.13: Publish/Subscribe messaging
Source: [7]

Publish/subscribe is a system which:

- 1 system acts as a Publisher (Producer) and publishing messages
- another system acts as a Subscriber (Consumer) and subscribes to receive new messages.

Using the above mechanism helps Kafka reduce the complexity when integrating many different types of system together: These system only have to implement the protocol used by the Pub/Sub messaging platform. For instance, Kafka Connect is tool for provides connectors for multiple types of Publishers and Consumers.

2.3.3 Data partitioning

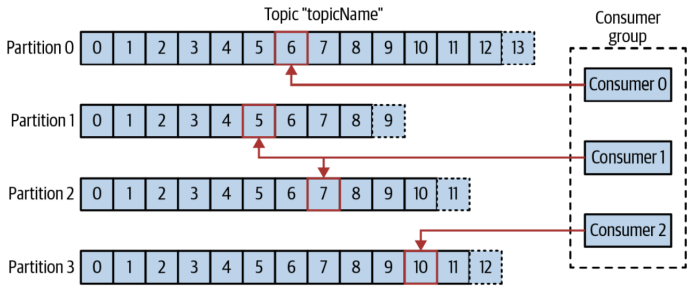


Figure 2.14: Topic partitioning in Kafka
Source: [8]

Data in Kafka are categorized into Topics. Topics are typically split into many different Partitions. The partitions reside on many different servers not only allow load balancing between a Kafka cluster, but also helps Consumers share the processing data.

If messages written to Kafka don't have *key* field, they will be divided randomly between the partitions. Otherwise, they will be separated based on their hashed key.

2.3.4 Data retention

By storing data inside topic as a queue, Kafka makes sure disconnected Consumer can still re-subscribe and receive data in the correct order. In addition, Kafka cluster also has a replicate mechanism among the brokers, allowing it

to operate even when many brokers are down. This helps make Apache Kafka a **High-Availability** service.

2.3.5 ZooKeeper and Kafka

2.3.5.1 What is ZooKeeper

ZooKeeper is an open-source, distributed coordination service. It is used by many other distributed applications to help manage synchronization, configuration maintenance and grouping - naming.

To achieve both high-availability and consistency, data within ZooKeeper is divided between its nodes. As one node fails, ZooKeeper then perform instant failover migration.

2.3.5.2 Why is ZooKeeper necessary for Kafka?

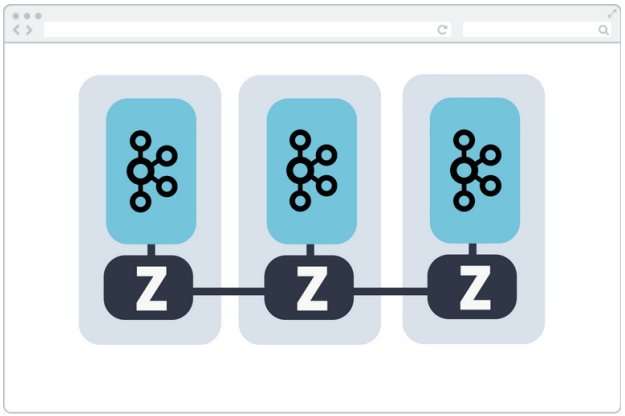


Figure 2.15: ZooKeeper and Kafka co-ordination
Source: [9]

Electing leader: Even though Kafka doesn't follow the Master-Worker architect, it still need a way to managing the partitions and replications. ZooKeeper provides a way for brokers to elect a leader that handle those tasks. ZooKeeper also helps the brokers re-elect the leader when nodes go down.

Members of cluster: What brokers and members of a cluster is still alive? ZooKeeper handles it.

ACL: Using ZooKeeper, Kafka can maintain a list of the Producer/Consumer that have access to a topic.

2.4 Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing. The main feature of Apache Spark is its **in-memory cluster computing** that increases the processing speed of an application. Spark provides an interface for programming entire clusters with implicit **data parallelism and fault tolerance**. It is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries, and streaming.

2.4.1 Spark ecosystem

2.4.1.1 Spark Core

Spark Core is the base engine for large-scale parallel and distributed data processing. Further, additional libraries that are built on the top of the core allows diverse workloads for streaming, SQL, and machine learning. It is responsible for memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster & interacting with storage systems.

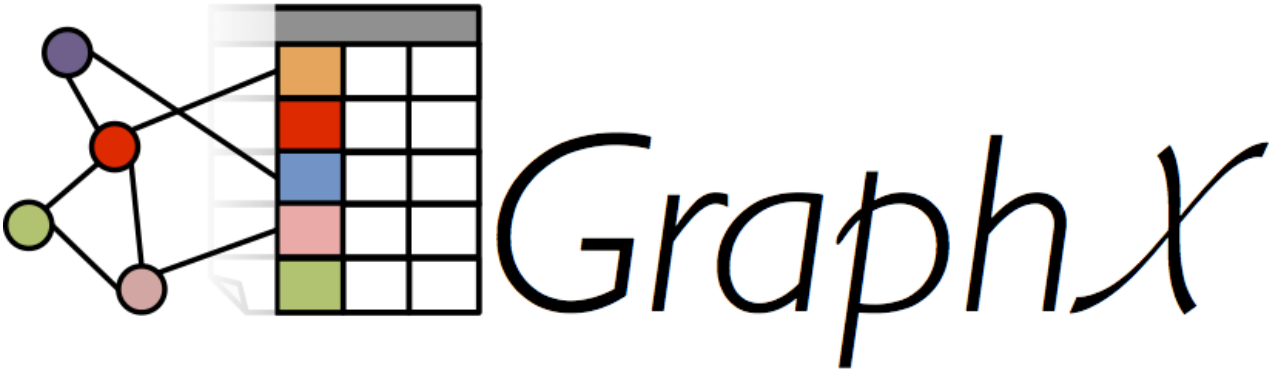
2.4.1.2 Spark Streaming

Spark Streaming is the component of Spark which is used to process real-time streaming data. Spark Streaming enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards.

Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs

Figure 2.16

2.4.1.3 GraphX



GraphX is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph (a directed multigraph with properties attached to each vertex and edge).

2.4.1.4 MLlib (Machine Learning)

MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

At a high level, it provides tools such as:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

2.4.2 Spark Architecture

Apache Spark has a well-defined layered architecture where all the spark components and layers are loosely coupled. This architecture is further integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions:

- Resilient Distributed Dataset (RDD)
- Directed Acyclic Graph (DAG)

2.4.2.1 Resilient Distributed Dataset (RDD)

RDDs are the building blocks of any Spark application. RDDs Stands for:

- **Resilient**: Fault-tolerant and is capable of rebuilding data on failure.
- **Distributed**: Distributed data among the multiple nodes in a cluster
- **Dataset**: Collection of partitioned data with values

At a high level, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

2.4.2.2 Directed Acyclic Graph (DAG)

DAG is a finite direct graph with no directed cycles. There are finitely many vertices and edges, where each edge directed from one vertex to another. It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence. It is a strict generalization of MapReduce model. DAG operations can do better global optimization than other systems like MapReduce.

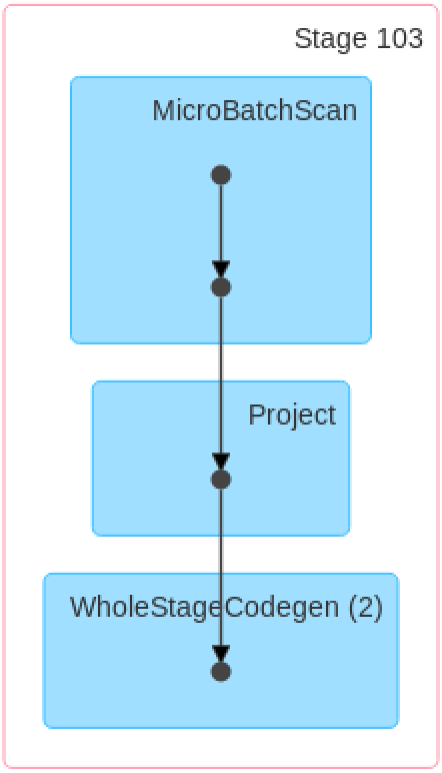


Figure 2.17: Sample DAG of our own Spark project

Apache Spark DAG allows the user to dive into the stage and expand on detail on any stage. In the stage view, the details of all RDDs belonging to that stage are expanded. The Scheduler splits the Spark RDD into stages based on various transformation applied.

2.4.2.3 Distributed Computing

To achieve distributed computing requires resource and task management over a cluster of machines. Resource management involves acquiring the available machines for the current task, while task management involves coordinating the code and data across the cluster.

A Spark application consists of a driver program and executes parallel computing on a cluster. To start a Spark application, the driver program that runs on a host machine will first initiate a SparkContext object. This SparkContext object will communicate with a cluster manager, which can be either Spark's own standalone cluster manager, Mesos, YARN, or Kubernetes, to acquire resources for this application. Then, the SparkContext object will send the application code and tasks to the worker nodes.

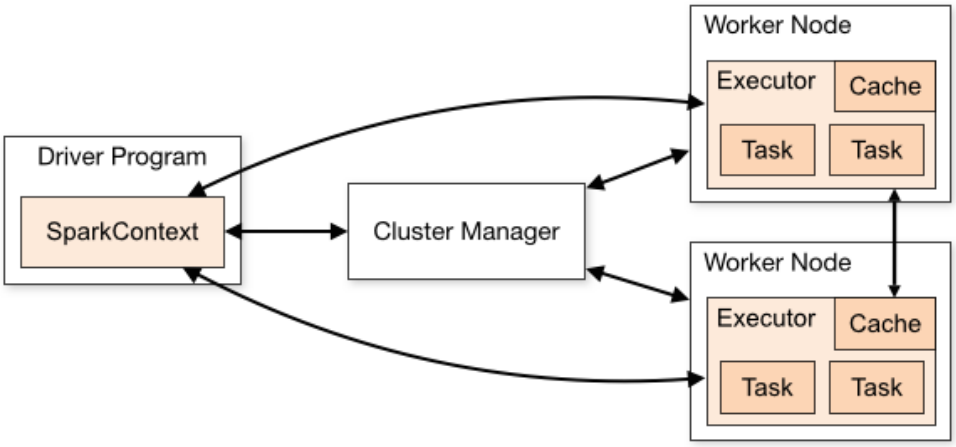


Figure 2.18: Spark Architecture in Distributed Computing

For one application, a worker node can have multiple executors, depending on the number of CPUs available on this worker node. During the computation for an application, each executor keeps data in memory or disk storage and runs tasks. In this way, executors are isolated from each other, and the tasks for the same application run in parallel.

2.5 HDFS

2.5.1 Overview

When the data file size exceeds the storage capacity of a computer, it will inevitably lead to the need to divide the data onto many computers. File systems that manage the storage of data over a network of many computers are called distributed file systems. Because they operate over the internet, distributed file systems are much more complex than a local file system. For example, a distributed file system must manage status of the servers participating in the file system.

Apache brings us the HDFS (Hadoop Distributed File System) in an attempt to create a responsive data storage platform for large volumes of data at a low cost.

2.5.2 HDFS Architecture

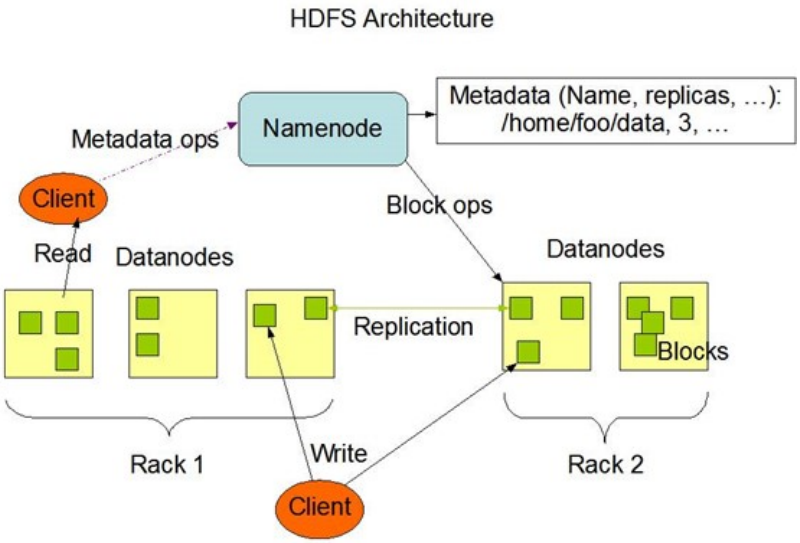


Figure 2.19: The HDFS Architecture
Source: [10]

HDFS is designed based on master/slave model, where master is NameNode and slave is DataNode.

NameNode is a place to store metadata information of the cluster, including information such as location, number of blocks of a file, access permission to that file, etc. NameNode is a place to receive read and write requests from the client and coordinate cluster's activities.

The metadata in the NameNode is stored as two files format:

- FSImage: Filesystem image that stores NameNode's metadata since initialization.
- EditLogs: Contains information about the most recent changes of the cluster not yet in FSImage.

DataNode is a physical data storage place, this is where the block of the file is stored. The DataNode is responsible for reporting the status to the NameNode, informing about the list of blocks it manages. NameNode will replicate blocks of this DataNode to other DataNode to ensure data is always available, in case the DataNode dies. Based on the coordination of the NameNode, the DataNode is the place to perform the user's file read and write requests.

Besides NameNode, there is also a process called SecondaryNamenode that works as a process supporting NameNode and is not used for replacing NameNode.

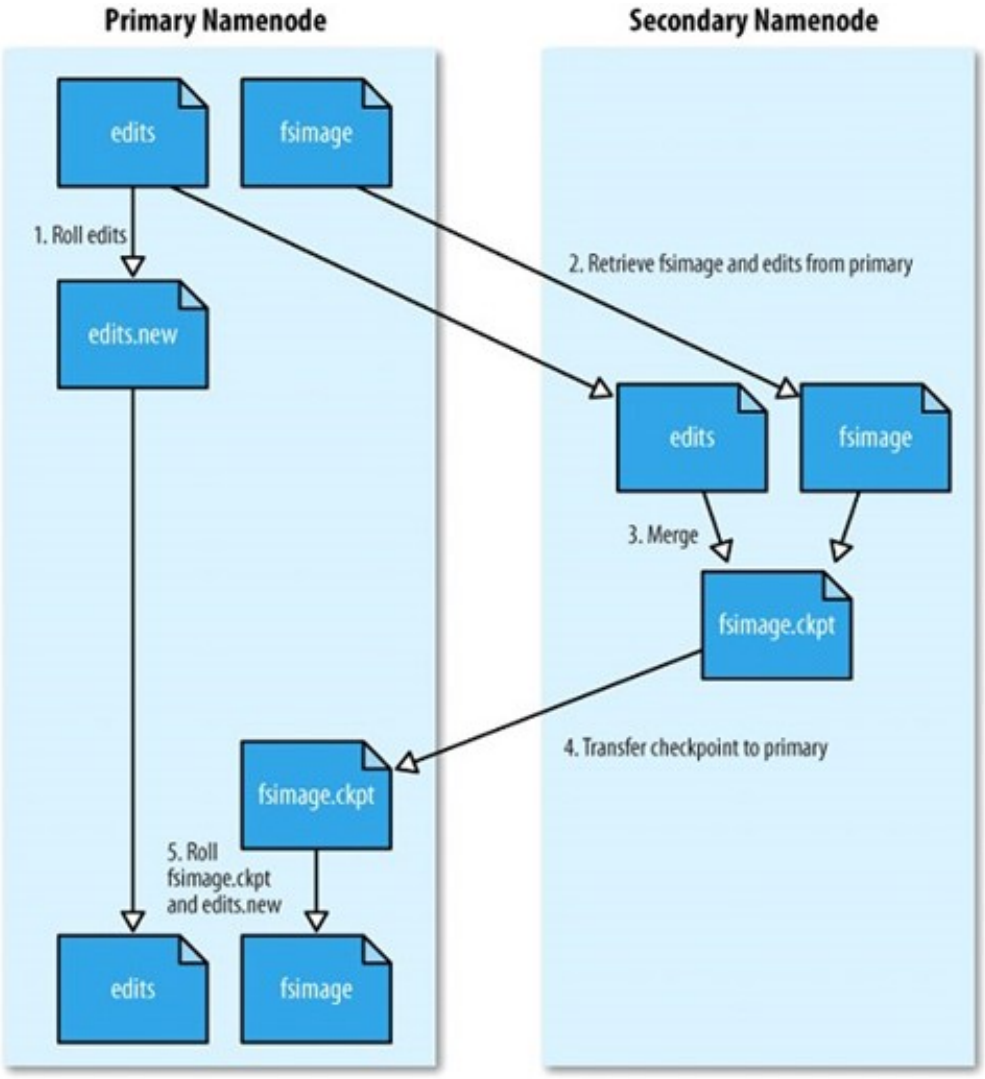


Figure 2.20: SecondaryNamenode
Source: [10]

2.5.3 Blocks

Blocks is a storage unit of HDFS, the data put into HDFS will be divided into blocks of fixed size (if not configured, it defaults to 128MB).

What happens if you store small files on HDFS?

Answer: HDFS will not be good when dealing with a large number of small files. Each data stored on HDFS is represented by 1 block with a size of 128MB, so if we store a large number of small files, we will need a large number of blocks to store them while each block we only need to use a little and there is a lot of excess space that causes waste. We can also see that the block size of the file system in other typical operating systems like Linux is 4KB, which is very small compared to 128MB.

2.5.4 Data Writing Process

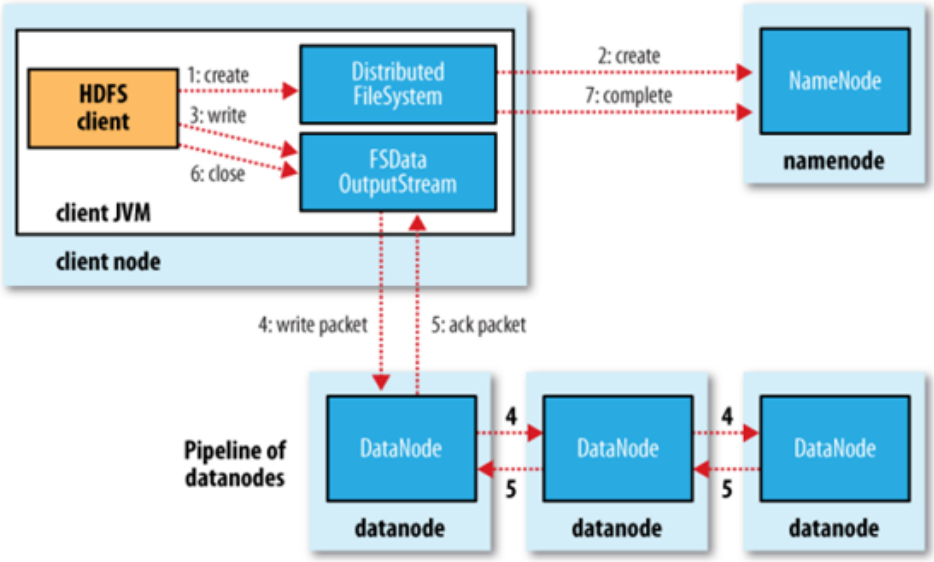


Figure 2.21: Data Writing Process
Source: [11]

From the sequence in the figure, we have the following steps to write data:

Step 1: Client sends request to create file in DistributedFileSystem APIs.

Step 2: DistributedFileSystem requests creating a file at NameNode. NameNode checks the client's permissions and checks if the new file exists or not...

Step 3: DistributedFileSystem returns FSDataOutPutStream to the client to write data.

FSDataOutputStream contains DFSOutputStream, it is used for handling interaction with NameNode and DataNode. When the client writes data, the DFSOutputStream splits the data into packets and pushes it into the DataQueue queue. The DataStreamer will tell the NameNode to allocate blocks to datanodes to store duplicates.

Step 4: The DataNodes form the pipeline, the number of datanodes is equal to the number of copies of the file. The DataStream sends the packet to the first DataNode in the pipeline, which forwards the packet to the Datanodes in the pipeline respectively.

Step 5: DFSOutputStream has an Ack Queue to maintain packets that have not been acknowledged by DataNodes. Packet out of Ack Queue upon receipt of acknowledgments from all DataNodes.

Step 6: The client calls close() to finish writing data, the remaining packets are pushed into the pipeline.

Step 7: After all packets are written to the DataNode, annouce that the data writing process is completed.

That's all that goes on behind the scenes, and here's an example of one of the hdfs write commands:

```
hdfs dfs -put <path_on_your_computer> <path_on_hadoop>
```

In the actual process, almost will not work directly with hadoop's file system (HDFS) with commands, but we often read and write via spark, for example:

```
dataFrame.write.save("<path_on_hadoop>")
```

2.5.5 Data Reading Process

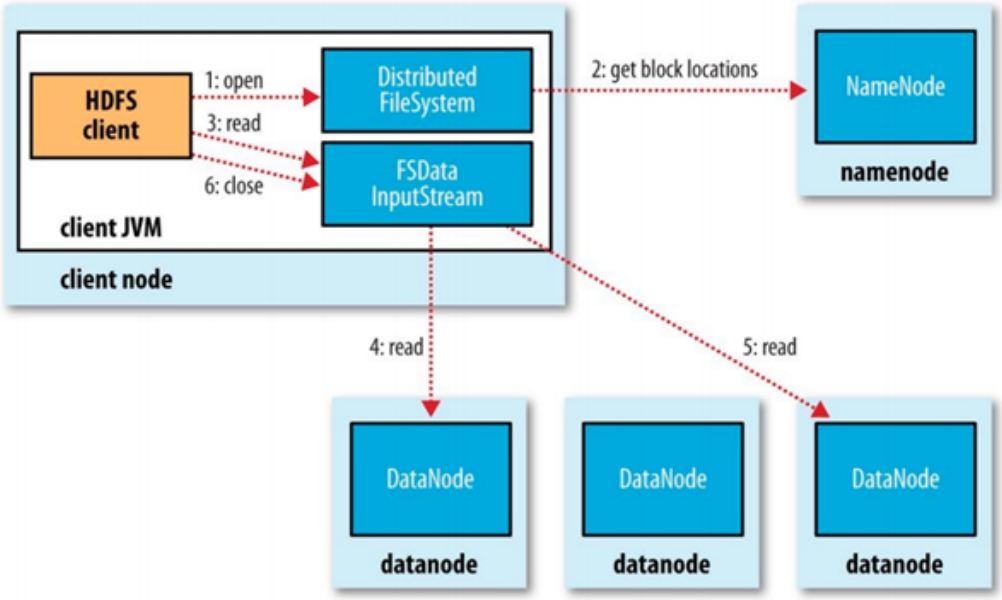


Figure 2.22: Data Reading Process
Source: [11]

Step 1: To open the file, the client calls the open method on FileSystemObject.

Step 2: DistributedFileSystem calls Name to get the location of the file's blocks. NameNode returns the addresses of the DataNodes that contain copies of that block.

Step 3: After receiving the address of the NameNode, an FSDataInputStream object is returned to the client. FSDataInputStream contains DFSInputStream. DFSInputStream manages the I/O of the DataNode and the NameNode.

Step 4: Client calls read() method at FSDataInputStream, DFSInputStream connects to the nearest DataNode to read the first block of the file. The read() method is repeated many times until the end of the block

Step 5: After reading, DFSInputStream disconnects and determines the DataNode for the next block. When DFSInputStream reads the file, if there is an error it will switch to the nearest other DataNode containing that block.

Step 6: When the client finishes reading the file, call close().

2.5.6 Identify the topology of the network

In the context of large-scale data processing across a network environment, recognizing the bandwidth limit between nodes is an important factor for Hadoop to make decisions in data distribution and distributed calculating. The idea of measuring the bandwidth between 2 nodes seems reasonable, but doing this is difficult (because the measurement of the network bandwidth needs to be done in a quiet environment, i.e. at that time it must not happen data exchange over the network). Therefore, Hadoop used the network topology of the cluster to quantify the data transmission capacity between any two nodes.

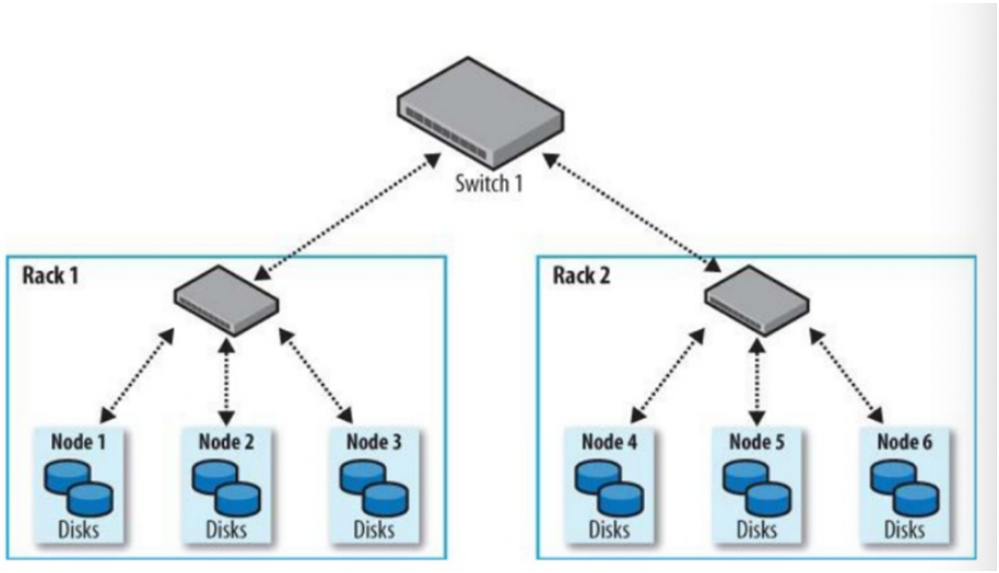


Figure 2.23: Hierachical tree structure of the network
Source: [11]

Hadoop perceives the cluster network topology through a hierarchical tree structure. This structure will help Hadoop to recognize the distance between 2 nodes on the cluster. On this hierarchical structure, bridges will act as internal nodes to divide the network into subnets, 2 nodes with the same parent node or on the same subnet are considered to be on the same rack.

Hadoop introduces a concept of network addressing to determine the relative location of nodes. The network address of any node will be the path from the root node to the specified node. For example, the network address of Node 1 in the image above will be Switch 1 – Rack 1 – Node 1.

Hadoop will calculate the distance between any 2 nodes simply by the sum of the distances of the 2 nodes to the nearest common parent node. For example, as shown in the image above, the distance between Node 1 and Node 2 is 2, the distance between Node 1 and Node 4 is 4.

Hadoop makes the following assumptions about racks:

- Bandwidth decreases in the following order:

  - Processes on the same node.
  - Different nodes on the same rack.
  - Nodes are not on the same rack.

- The closer two nodes are to each other, the larger the bandwidth between those two nodes. This assumption reaffirms the first assumption.

- The possibility that two nodes located on the same rack will fail is higher than that of two nodes located on two different racks. This will be applied when the NameNode makes copies of a block down to the DataNodes.

2.5.7 Arrange copies of blocks onto DataNodes

When choosing a DataNode to store, the NameNode will choose based on the following 3 factors:

- Reliability

- Read bandwidth

- Data logging bandwidth

The DataNode will be selected according to the following strategy, the first copy of a data block will be placed on the same node as the client – if the client program writing data is also in the cluster, otherwise, the NameNode will randomly choose a DataNode. The second copy will be placed on a random DataNode on a different rack than

the node that stored the first. The third one, will be placed on a DataNode located on the same rack as the node storing the second copy. Further copies are placed on randomly selected DataNodes.

2.5.8 Cluster balancing

Over time the distribution of data blocks on the DataNode can become unbalanced, with some nodes storing too many data blocks while others have fewer. An unbalanced cluster can affect MapReduce optimization and will put pressure on DataNodes that store too many data blocks.

To avoid this situation, Hadoop has a program called balancer which will run as a daemon on the NameNode that will do the rebalancing of the cluster. Starting or opening. this program will be independent of HDFS (i.e. when HDFS runs, we can freely turn on or off this program), but it is still a component on HDFS. The balancer will periodically redistribute copies of the data block by moving it from overloaded DataNodes to empty DataNodes, while still ensuring strategies for arranging copies of blocks onto the DataNodes.

2.5.9 HDFS Advatages – Why do we use HDFS for storage?

- Big data storage cost savings: can store data megabytes to petabytes, in structured or unstructured form.

- Data is highly reliable and has the ability to fix errors: Data stored in HDFS is replicated into many versions and stored at different DataNodes. When one machine fails, the data is still stored in another DataNode.

- High accuracy: Data stored in HDFS is regularly checked by a checksum code calculated during file writing, if there is an error, it will be restored by copies.

- Scalability: can increase hundreds of nodes in a cluster.

- High throughput: high data access processing speed.

- Data Locality: moving the computation close to where the actual data resides on the node, instead of moving large data to computation. This minimizes network congestion and increases the overall throughput of the system.

Chapter 3

Our project

Our team designs this system with a target of easy install & upgrade, simple scaling up / down and high-availability as good as possible.

3.1 Explanation

3.1.1 System Context

Figure 3.1: The System Context diagram of our DataPipeline system

In general, our system helps move data collected from a source to a Data Lake. The Data Source this could be physical/virtual sensors (research topic #1, #2), or PCs and smartphones. Stored data can be later process using Apache Spark (research topic #4)

3.1.2 Container

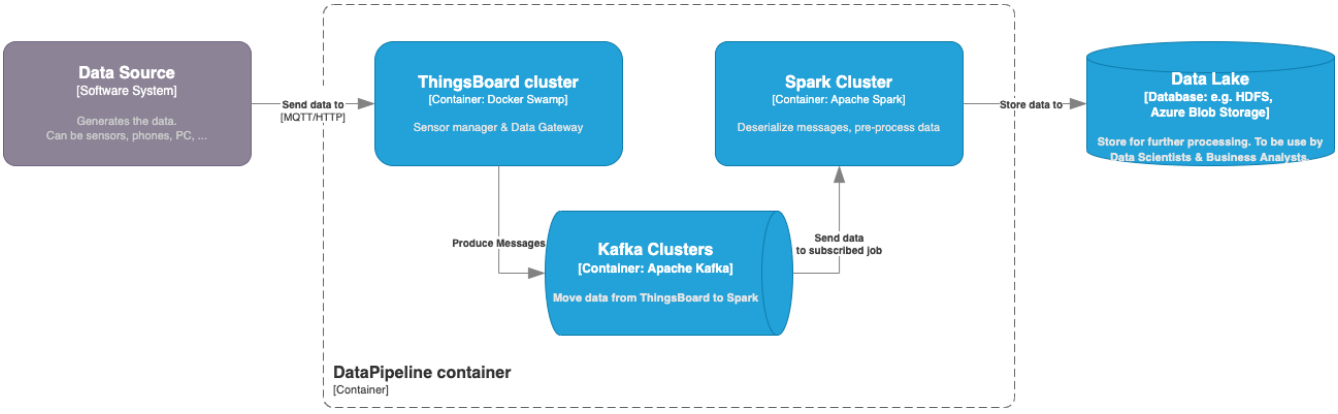


Figure 3.2: The Container diagram of our DataPipeline system

More specifically, DataPipeline will consist of 3 major components: a ThingsBoard, a Kafka and a Spark cluster. ThingsBoard will work as both a device manager and a gateway for incoming messages. Upon receiving a new message, ThingsBoard will then forward it to the Spark cluster using a Kafka broker. A Spark job will then deserialize the message as well as perform simple transformation. Finally, new data will be written into a Data Lake (HDFS cluster)

3.1.3 Components

3.1.3.1 Sensors

The sensor start working when we run file ./start-sensor.sh. This file first checks if there are any processes working by checking file sensor.id. If there's no process working, the system will create a process and write the process' PID to file sensor.pid.

```
if [[ -f "/tmp/sensor.id" ]]; then
    echo "Process is already running"
    exit 1
fi

source ".venv/bin/activate"

python3 "sensor_mqtt.py" &
exit
```

Figure 3.3: Code of start-sensor.sh

Then the process will run file sensor_mqtt.py to receive data continuously from sensor and write it to the cleaner_weather_data.csv file with the delay time and the working sensor's token as setting in file sensor.cfg. We use both HTTP and MQTT to send data from sensor to the data cluster: port 80 for HTTP and port 1883 for MQTT.

```
[root@hpcc-29 bin]# ./kafka-console-consumer.sh --bootstrap-server 10.1.8.29:9092 --topic weather-data
{"StationCode":101,"SO2":0.004,"NO2":"0.059000000000000004","O3":0.002,"CO":1.2,"PM10":73.0,"PM2.5":57.0,"SendTime":"2022-06-03T10:05:51.129073"}
{"StationCode":101,"SO2":0.004,"NO2":"0.059000000000000004","O3":0.002,"CO":1.2,"PM10":73.0,"PM2.5":57.0,"SendTime":"2022-06-03T10:06:41.448128"}
{"StationCode":101,"SO2":0.004,"NO2":"0.057999999999999996","O3":0.002,"CO":1.2,"PM10":71.0,"PM2.5":59.0,"SendTime":"2022-06-03T10:06:56.461542"}
{"StationCode":101,"SO2":0.004,"NO2":"0.055999999999999994","O3":0.002,"CO":1.2,"PM10":70.0,"PM2.5":59.0,"SendTime":"2022-06-03T10:07:11.477373"}
{"StationCode":101,"SO2":0.004,"NO2":"0.055999999999999994","O3":0.002,"CO":1.2,"PM10":70.0,"PM2.5":58.0,"SendTime":"2022-06-03T10:07:26.494822"}
```

Figure 3.4: Data received while running the sensor

```
1  [sensor]
2  # Delay between sendings (seconds)
3  delay.time=20
4  sensor.number=101
5
6  # The Python logging's debug level
7  debug.level=DEBUG
8
9  # [mqtt]
10 # broker=localhost
11 # port=1883
12 # token=VXS85aKXpED850XcCFon
13 # # topic=weather/telemetry
14 # topic=v1/devices/me/telemetry
15
16 [http]
17 hostname=localhost:8080
18 token=W7jpUW9EZMIFa1inbRqt
19
```

Figure 3.5: Setting at file sensor.cfg

If we want to turn off the sensor, kill the current process by running file ./stop−sensor.sh.

3.1.3.2 ThingsBoard

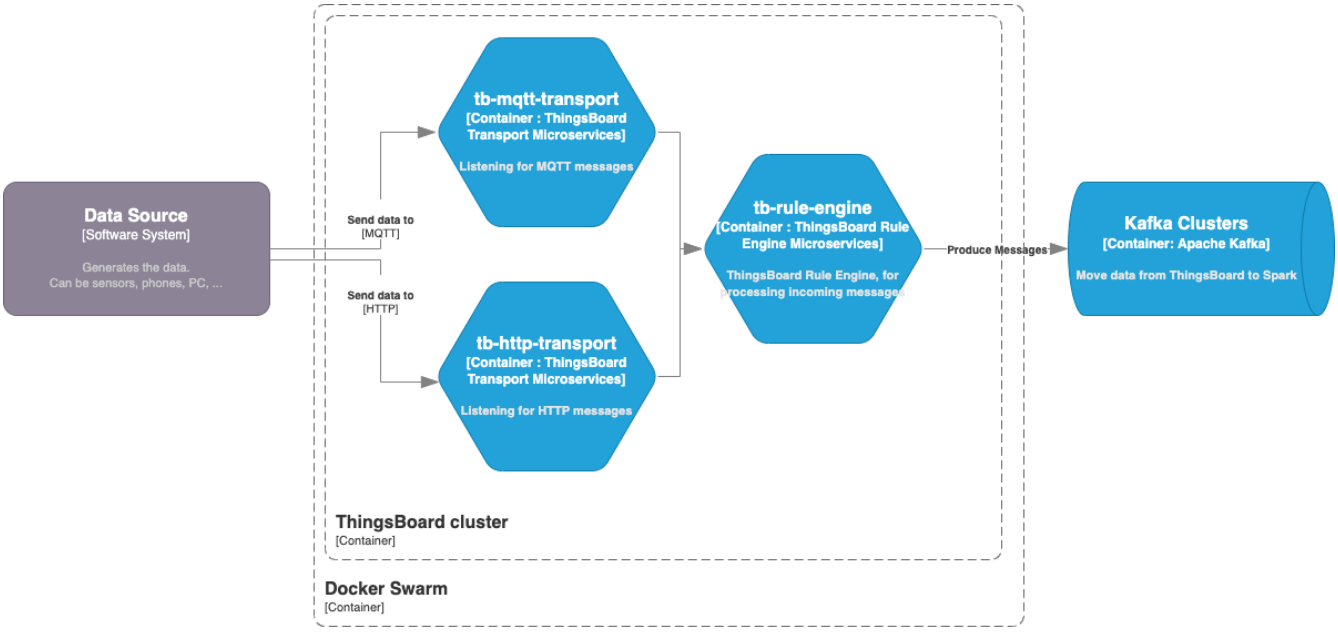


Figure 3.6: Component view of ThingsBoard cluster

Fig.3.6 describe component of ThingsBoard cluster. In this model, thingsboard run on docker that help model become high availability and scalability.

1. Install thingsboard
   Installing thingsboard isn't complicated, but it too long to instruct in this report. So we can reference this link to installing thingsboard.

   After installing ThingsBoard, you can open the online dashboard at 10.1.8.28:8080. You should see Things-

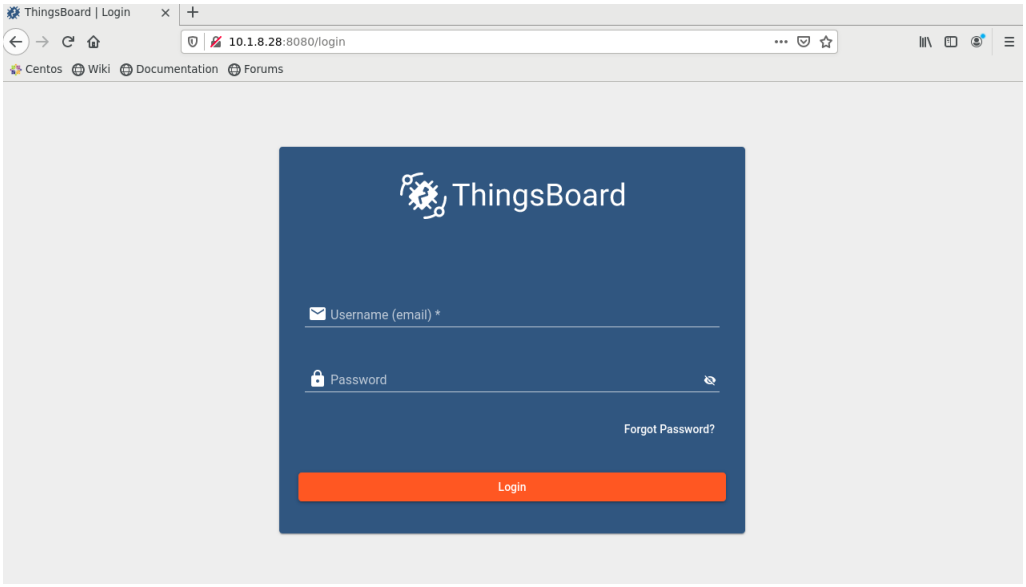


Figure 3.7: ThingsBoard login page

Board login page (see fig.3.7). Use the following default credentials:

- System Administrator: sysadmin@thingsboard.org / sysadmin
- Tenant Administrator: tenant@thingsboard.org / tenant

- Customer User: customer@thingsboard.org / customer

2. Create rule chains
   - Firstly, click on rule chain symbol on home page.

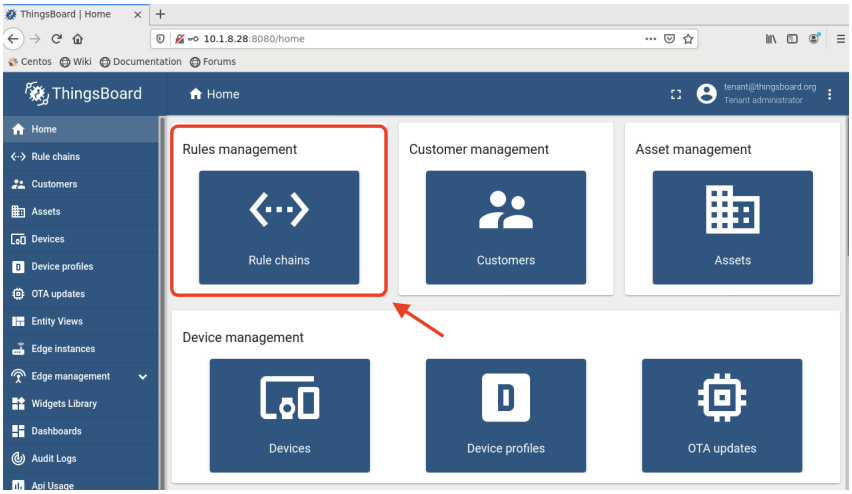


Figure 3.8: Rule chain in home page

- Secondly, click plus symbol in right-top of page to create new rule chains.

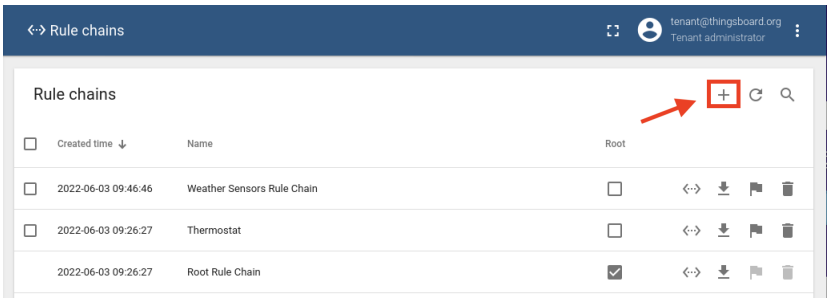


Figure 3.9: Add rule chain

- Thirdly, name rule chain.

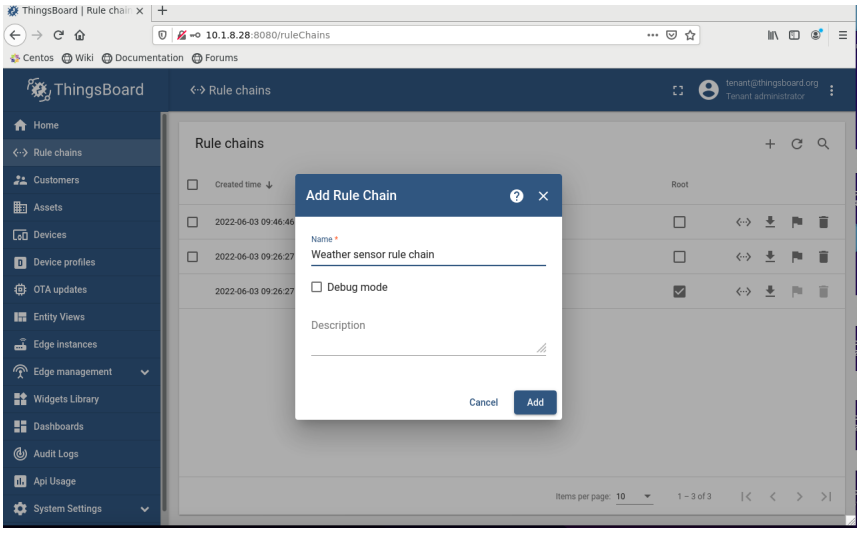


Figure 3.10: Name rule chain

- Finally, configure rule chain to compatible to our project.

  In our project, rule chain handle message and post it to kakfa if message type is telemetry.

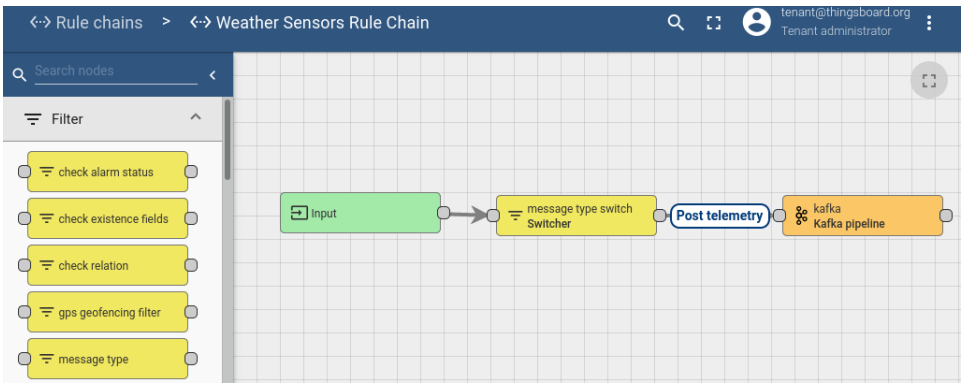


Figure 3.11: Configure rule chain

3. Create device profile
   - Firstly, Click on **Device profile in menu.**
   - Secondly,Name Device profile.

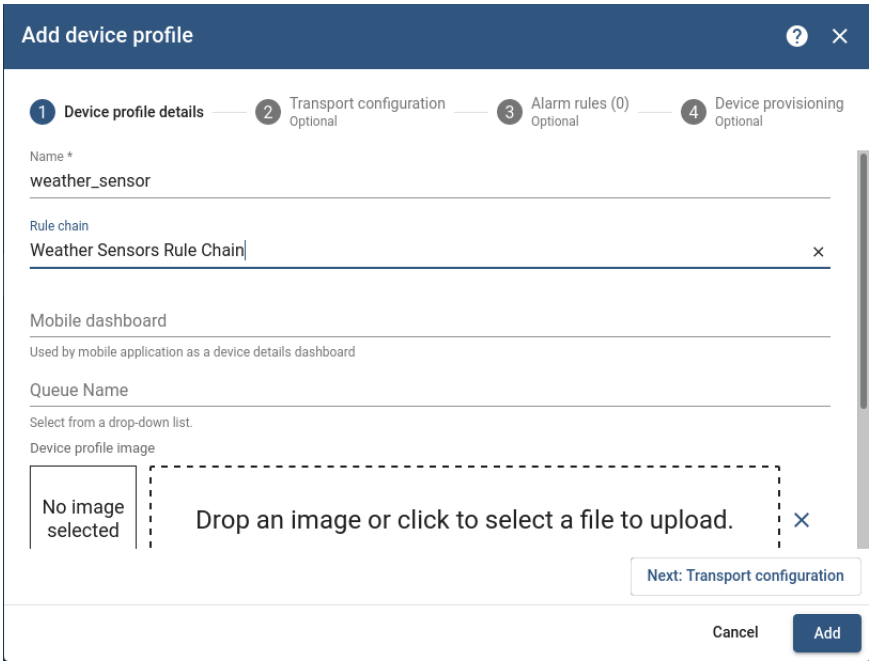


Figure 3.12: Named device profile

   - Finally, Configure protocol to Device.
4. Create device

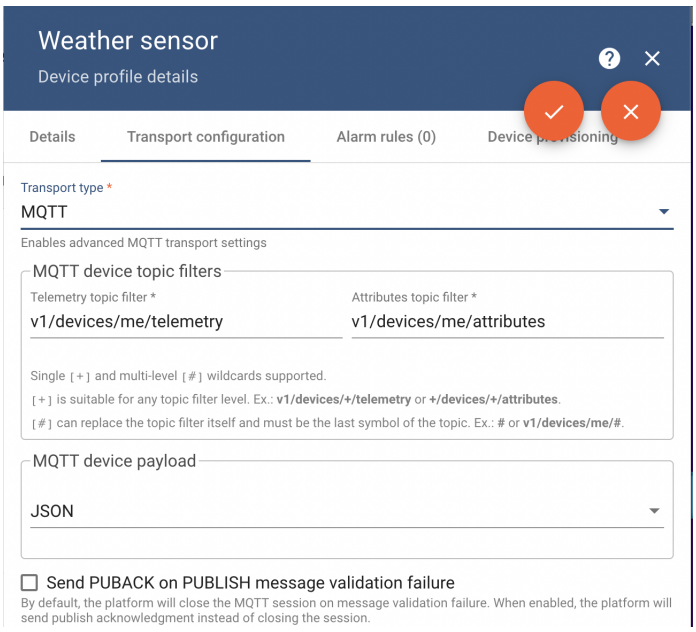


Figure 3.13: Configure protocol to device

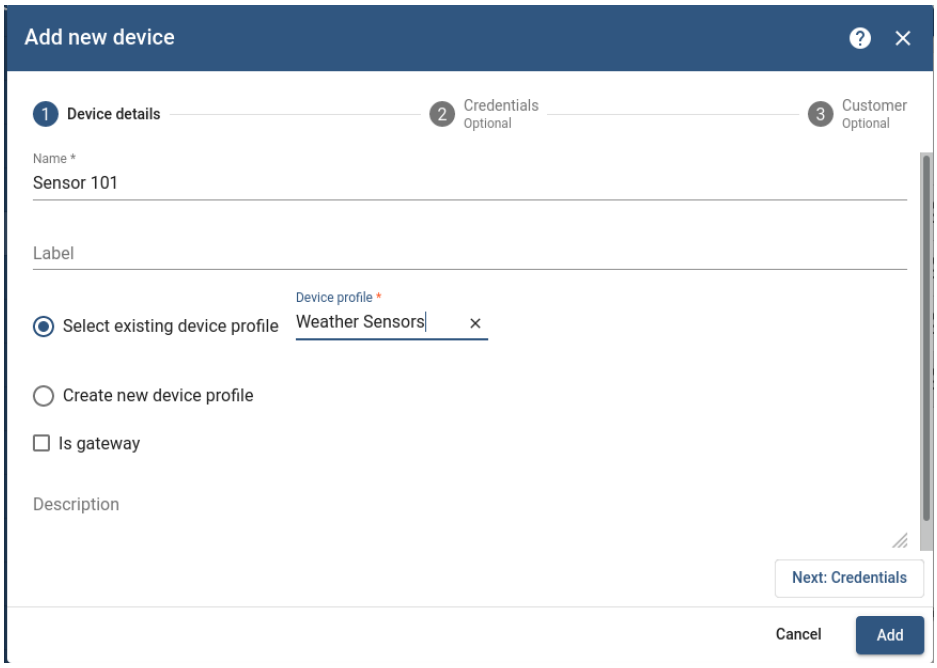


Figure 3.14: Create new Device

3.1.3.3 Kafka

Our Kafka cluster is comprised of 3 brokers running on 3 different machines. Each machine also runs a small ZooKeeper cluster. Installation starts with downloading both ZooKeeper and Kafka binaries from the official website.

To start ZooKeeper, use the following command:

```
./bin/zkServer.sh
```

And to check for ZooKeeper status, run:

```
./bin/zkServer.sh status
```

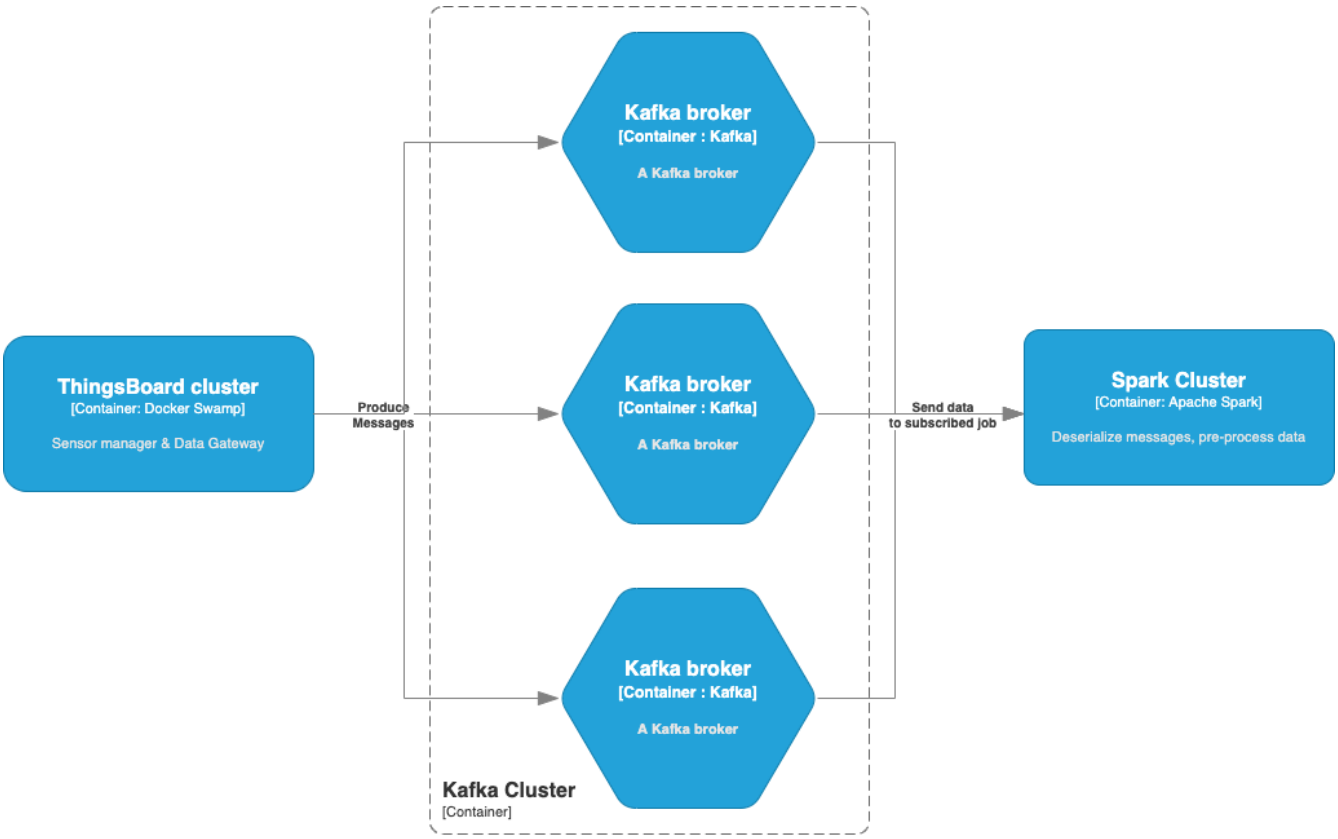


Figure 3.15: Component view of Kafka cluster

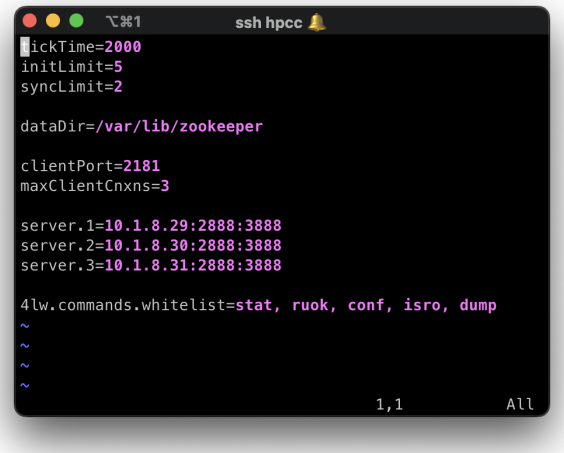


Figure 3.16: Sample ZooKeeper configuration file

After we have finished setting up ZooKeeper, we can move on to modify the Kafka configurations. One important settings of Kafka is *zookeeper.connect* which must be set to the corresponding ZooKeeper hostnames.

Start Kafka by running:

```
./bin/kafka-server-start.sh ./config/server.properties
```

Using ZooKeeper CLI, we can check if our Kafka has setup successfully.

```
./zkCli.sh ls /brokers/ids
```

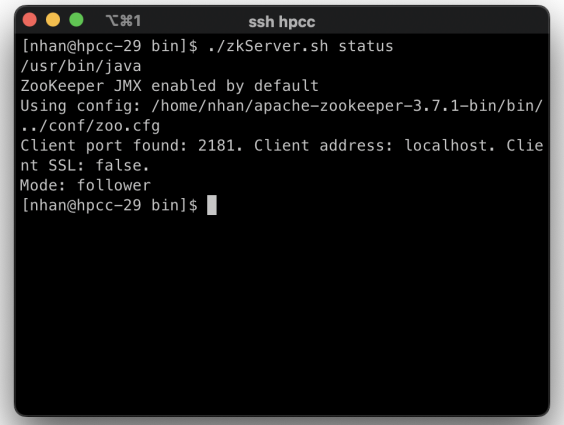


Figure 3.17: Sample ZooKeeper running status

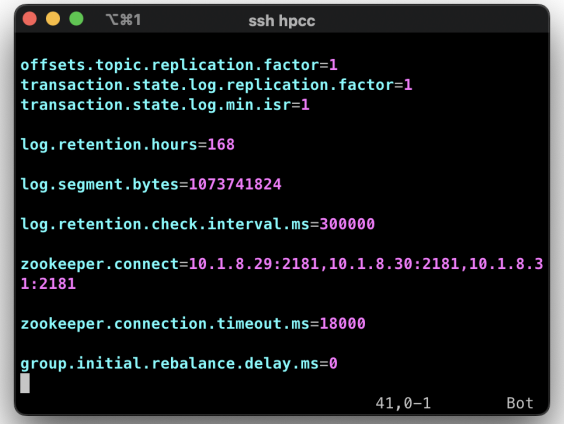


Figure 3.18: Sample Kafka configuration file

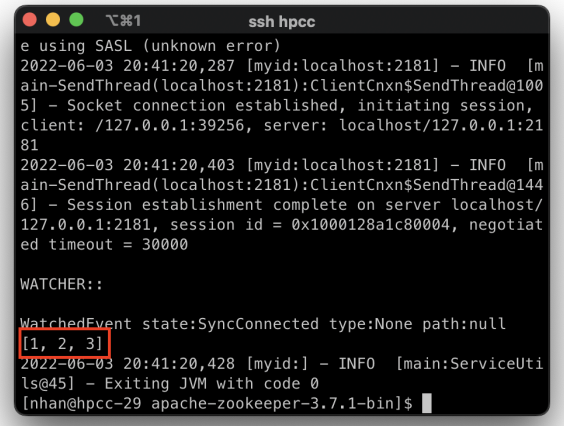


Figure 3.19: Display all Kafka broker's ids

3.1.3.4 Spark

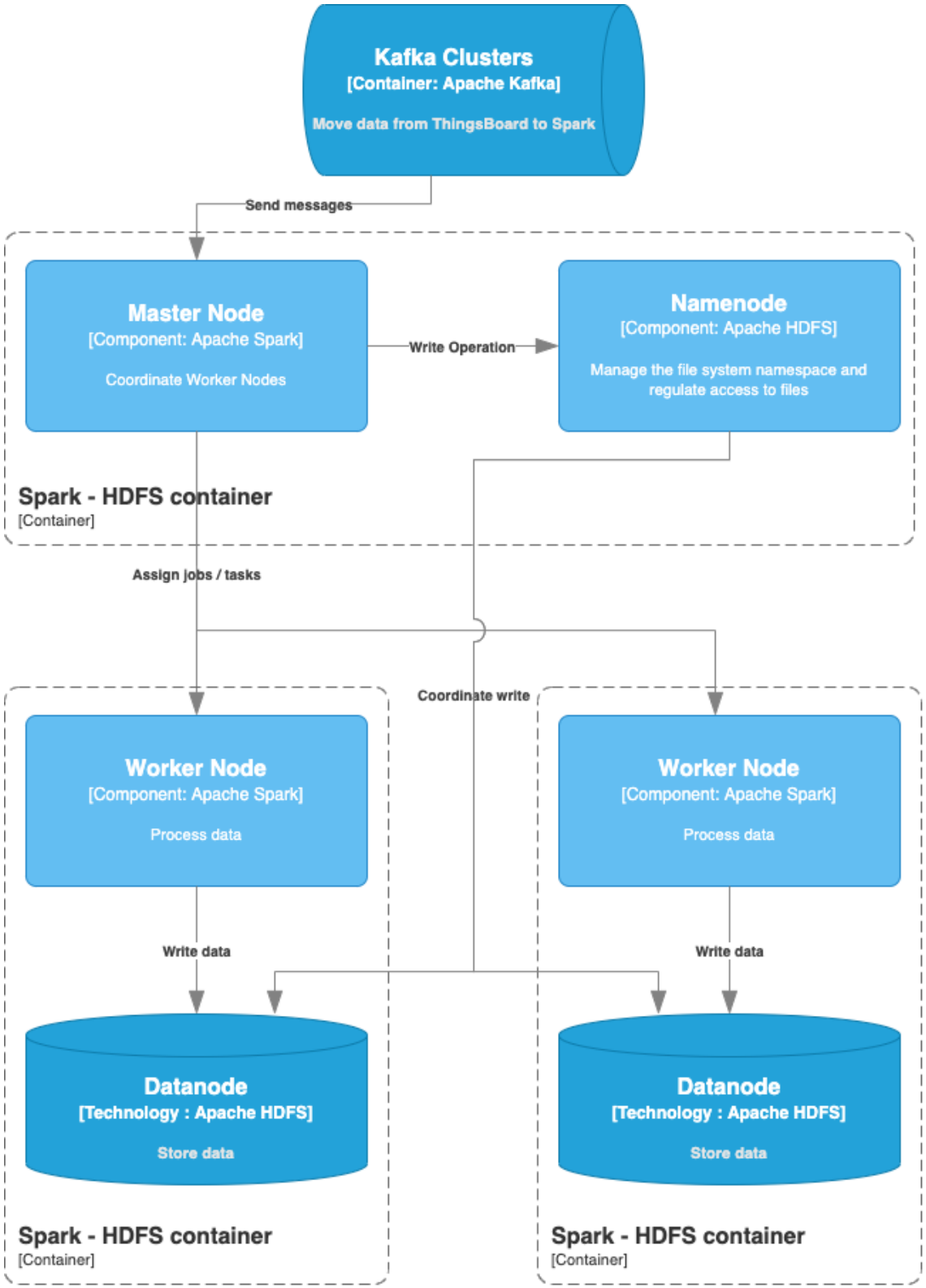


Figure 3.20: Component view of Spark - HDFS cluster

3.1.3.4.a Set up Spark

In provided machine, we have 4 nodes and we chose 3 of them to run Spark. On these machines, we installed Spark 3.2.1, Hadoop 3.2 and Scala 2.13 and set up that node `10.1.8.29` is the Spark master and others are workers.

3.1.3.4.b Spark Deploying

In this assignment, we run Spark in Standalone Mode. To install Spark Standalone mode in clusters, we did some following steps:

- We deploy Spark Standalone by cluster launch scripts. So, firstly, in ready phase, we create a file called `conf/workers` in Spark directory, which must contain the hostnames of all the machines where we intend to start Spark workers, one per line. In this case, we added 3 node of this machine as `10.1.8.29`, `10.1.8.30` and `10.1.8.31`. Where `10.1.8.29` is master and others are workers nodes.
- The master machine accesses each of the worker machines via SSH. By default, SSH is run in parallel and requires password-less (using a private key) access to be setup. So, we generated a SSH key on each worker node and added it to authorize key, which make it run password-less through the machine.
- After we complete the ready phase, we can launch or stop your cluster with the following shell scripts, based on Hadoop's deploy scripts, and available in `SPARK_HOME/sbin`. It has many option that can run each node separately, but we chose to run : `sbin/start-all.sh` and `sbin/stop-all.sh` to start/stop both the master and all workers nodes that had been config before. Note that these scripts must be executed on the machine you want to run the Spark master on (`10.1.8.29` for example in this case).
- Furthermore, we can optionally configure the cluster by setting environment variables in `conf/spark-env.sh`. Create this file by starting with the `conf/spark-env.sh.template`, and copy it to all worker machines for the settings to take effect.

3.1.3.4.c Data Processing with Spark

Spark has many programming languages support as Java, Python,... In this assignment, we chose Scala as it is easier to implement than Java and it has better runtime than Python. To work with Spark, first of all, we must has a Spark Session.

```
  val spark = SparkSession
    .builder()
    .appName("SparkScala")
    .getOrCreate()
```

After create a Spark Session, we can do stuffs with Spark. Next we read data from Kafka as streaming messages:

```
val rawDF = spark.readStream
  .format("kafka")
  .options(options)
  .load()
```

With this `readStream` command, we can read data from kafka with some options that we have designed, which include kafka bootstrap server, with topic we subscribe and the stating offset on this topic. By running that statement, we got a data frame which stored in `rawDF`. After that, we use the command `rawDF.printSchema()` to return the schema of streaming data from Kafka. The returned DataFrame contains all the familiar fields of a Kafka record and its associated metadata. The result will be:

```
> rawDF.printSchema()
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

The variable we consider in this dataframe is `value`, and it was saved as a binary type, which type that we are not expect it to use then. So, now we have to convert it to a Data Frame that we can easily handle it in the future, which called **Deserialization** job in Spark. To do this, first of all, we create a schema with some column that we

need to analyse with this data by a variable `weatherDataSchema`, then we use following command to complete this job:

```
val weatherDataDF =
  rawDF
    .select(
      from_json(col("value").cast(StringType), weatherDataSchema)
        .as("data")
    )
    .select("data.*")
```

Our data is comprised of 2 major information that air quality of a station and its coordinates. To more convenience to process data, we split it into 2 part and save it in 2 separated files. After working and return the result with air quality data file, we decide to combine it into one Data Frame before write it to stream. To do this, we save station information in to variable `stationDF` and join it with weather data that we have processed before.

```
val processedDF = weatherDataDF
  .join(
    stationDF.hint("broadcast"),
    weatherDataDF("StationCode") === stationDF("infoStationCode"),
    "left"
  )
  .drop("infoStationCode")
```

Finally, after processed data, we write it on stream with a trigger that write every 10 seconds:

```
val query = processedDF.writeStream
  .format("parquet")
  .option("checkpointLocation", "./checkpoint")
  .option("path", "~/scala/SparkScala/weatherData")
  .trigger(Trigger.ProcessingTime("10 seconds"))
  .start()
```

After implemented all stuff, we built and packaged it into .jar file and submit to Spark. Note that to read and write streaming, we need a running Kafka server that our Spark can connect to.

3.1.3.5 HDFS

To maximize the utilization between Spark and HDFS clusters, HDFS namenodes are installed and started on the same machines that used to run the Spark cluster. Which in our project is: PC-29, 30 and 31.

3.2 Run results

3.2.1 Kafka

If sensors and ThingsBoard are configured properly, our simple kafka consumer should capture some messages.

```
./kafka-console-consumer.sh --bootstrap-server 10.1.8.29:9092,10.1.8.30:9092,10.1.8.31:9092
↪  --topic weather-data
```

The result is as followed:

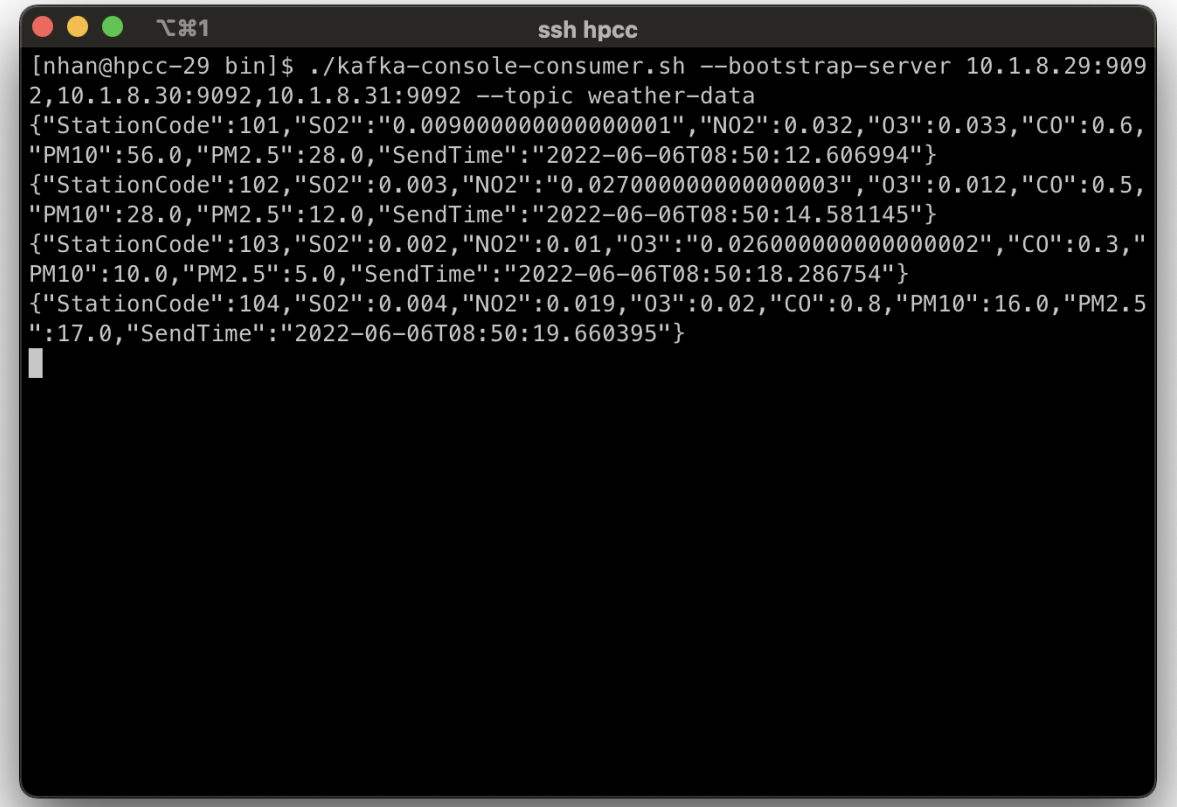


Figure 3.21: Sample Kafka messages

3.2.2 Spark

In addition to the result that popped in the terminal an saved as files, Spark also has a Website UI that has some information of jobs, workers, status, ...

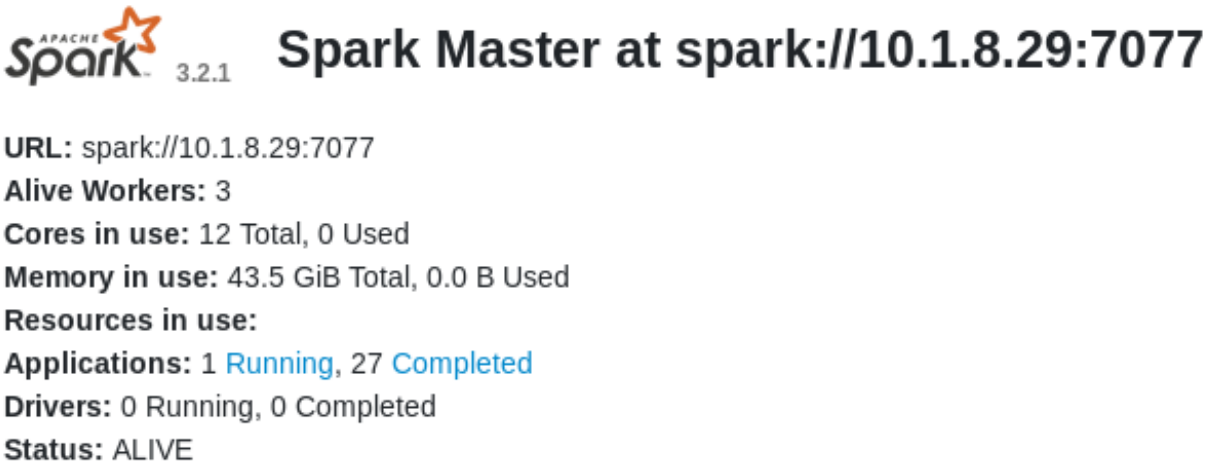


Figure 3.22: Spark detail information on machines

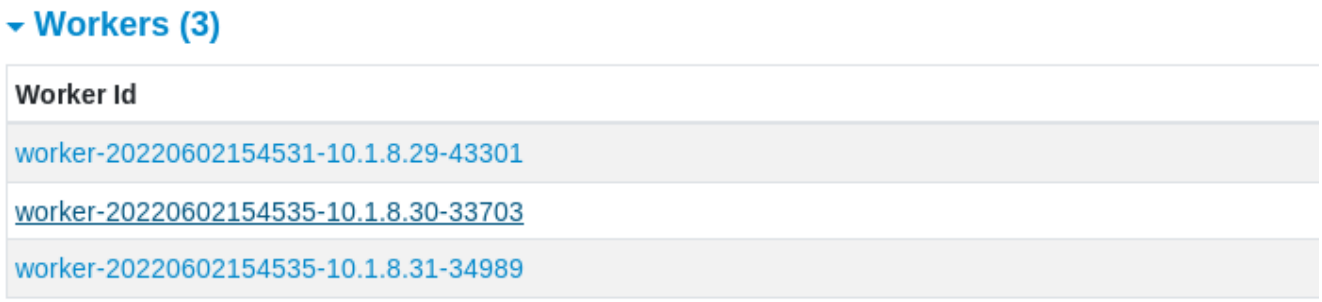
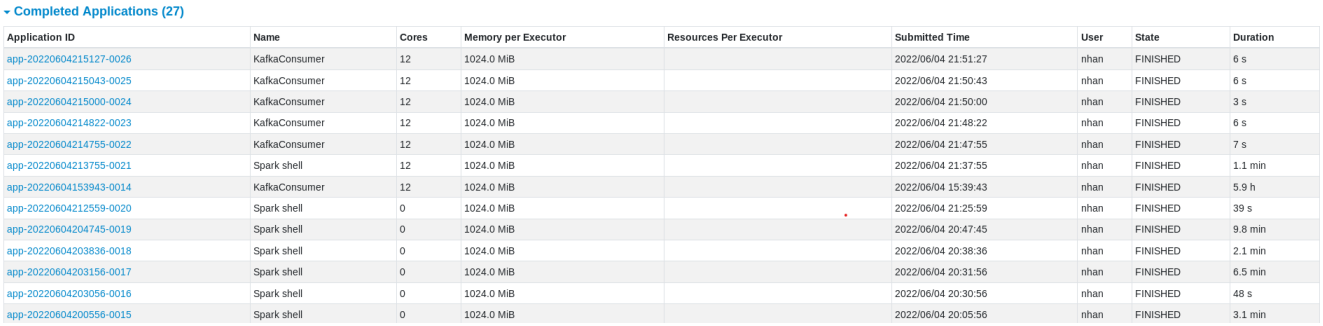


Figure 3.23: Spark Workers detail



Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220604215127-0026	KafkaConsumer	12	1024.0 MiB		2022/06/04 21:51:27	nhan	FINISHED	6 s
app-20220604215043-0025	KafkaConsumer	12	1024.0 MiB		2022/06/04 21:50:43	nhan	FINISHED	6 s
app-20220604215000-0024	KafkaConsumer	12	1024.0 MiB		2022/06/04 21:50:00	nhan	FINISHED	3 s
app-20220604214822-0023	KafkaConsumer	12	1024.0 MiB		2022/06/04 21:48:22	nhan	FINISHED	6 s
app-20220604214755-0022	KafkaConsumer	12	1024.0 MiB		2022/06/04 21:47:55	nhan	FINISHED	7 s
app-20220604213755-0021	Spark shell	12	1024.0 MiB		2022/06/04 21:37:55	nhan	FINISHED	1.1 min
app-20220604153943-0014	KafkaConsumer	12	1024.0 MiB		2022/06/04 15:39:43	nhan	FINISHED	5.9 h
app-20220604212559-0020	Spark shell	0	1024.0 MiB		2022/06/04 21:25:59	nhan	FINISHED	39 s
app-20220604204745-0019	Spark shell	0	1024.0 MiB		2022/06/04 20:47:45	nhan	FINISHED	9.8 min
app-20220604203836-0018	Spark shell	0	1024.0 MiB		2022/06/04 20:38:36	nhan	FINISHED	2.1 min
app-20220604203156-0017	Spark shell	0	1024.0 MiB		2022/06/04 20:31:56	nhan	FINISHED	6.5 min
app-20220604203056-0016	Spark shell	0	1024.0 MiB		2022/06/04 20:30:56	nhan	FINISHED	48 s
app-20220604200556-0015	Spark shell	0	1024.0 MiB		2022/06/04 20:05:56	nhan	FINISHED	3.1 min

Figure 3.24: Spark Jobs detail

3.2.3 HDFS

As figure 3.25 shows, the final data is written into /WeatherData - a .parquet file format.

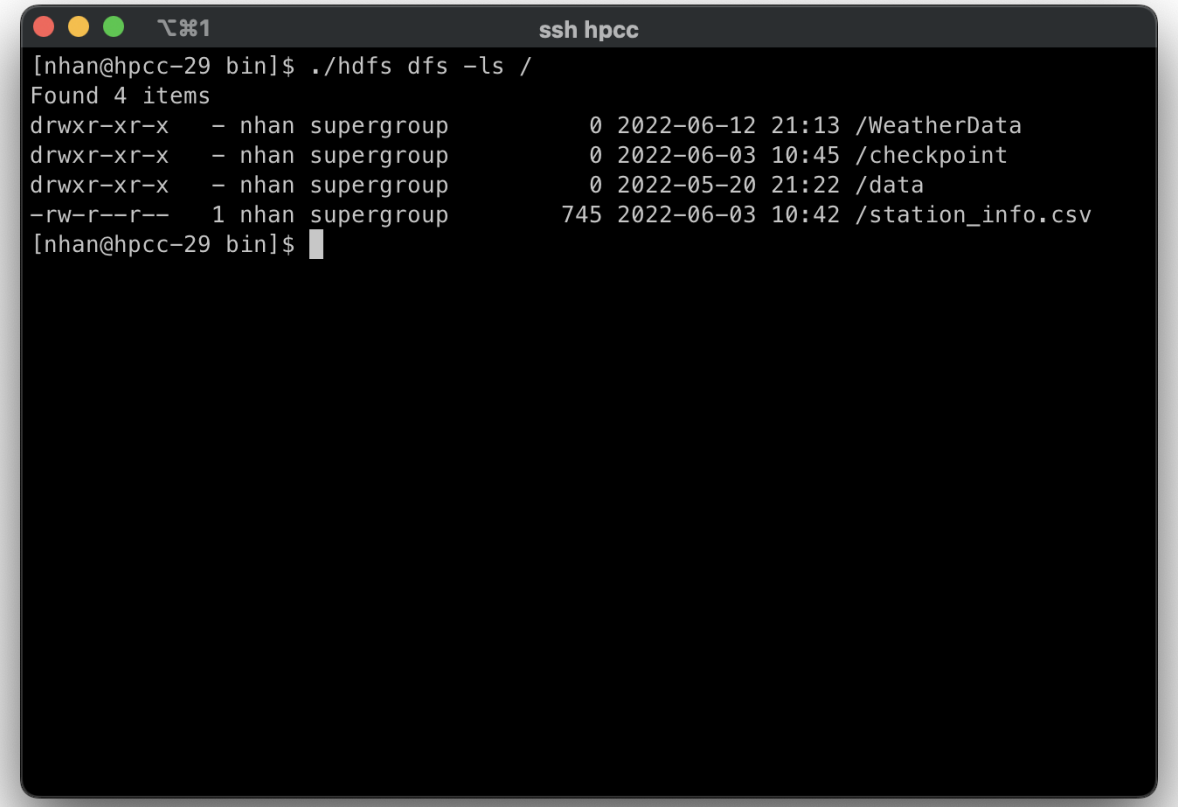


Figure 3.25: Some files on HDFS cluster

3.3 Challenges

Our system is designed to Data collection and integration software framework. But in further future, my system can be applied to some more complicated system. In that case, Our team have an example problem that my system may face up.

3.3.1 Data crawler integration

As our demo system only supports MQTT/HTTP-protocol sensors, one problem arise while implement it in a real world situation. That is: How do integrate data crawlers (Facebook comments, Twitter tweets, etc,.) into our system.

Solution depends mostly on how the bots behave. If they don't use typical IoT network protocols (MQTT, CoAP, ...), we can skip the ThingsBoard layer and sending messages directly to the Kafka pipeline.

On the other hand, if our bots act as a typical sensor, ThingsBoard can help supervising and forwarding messages from these crawler back to the storage.

3.3.2 Connect created sensor to Thingsboard

In our demo with Thingsboard, we only create new sensors to collect data for the system. To connect a created sensor to Thingsboard, we still create a new sensor from Thingsboard, then access the data the created sensor collected to the new sensor.

3.3.3 Alarming system

In the previous section, our team has applied our system to collect and processed data about air quality. And problem may be base on the data that has been collected, we have to analyst it and return an alarm to stations if the air quality is in bad status.

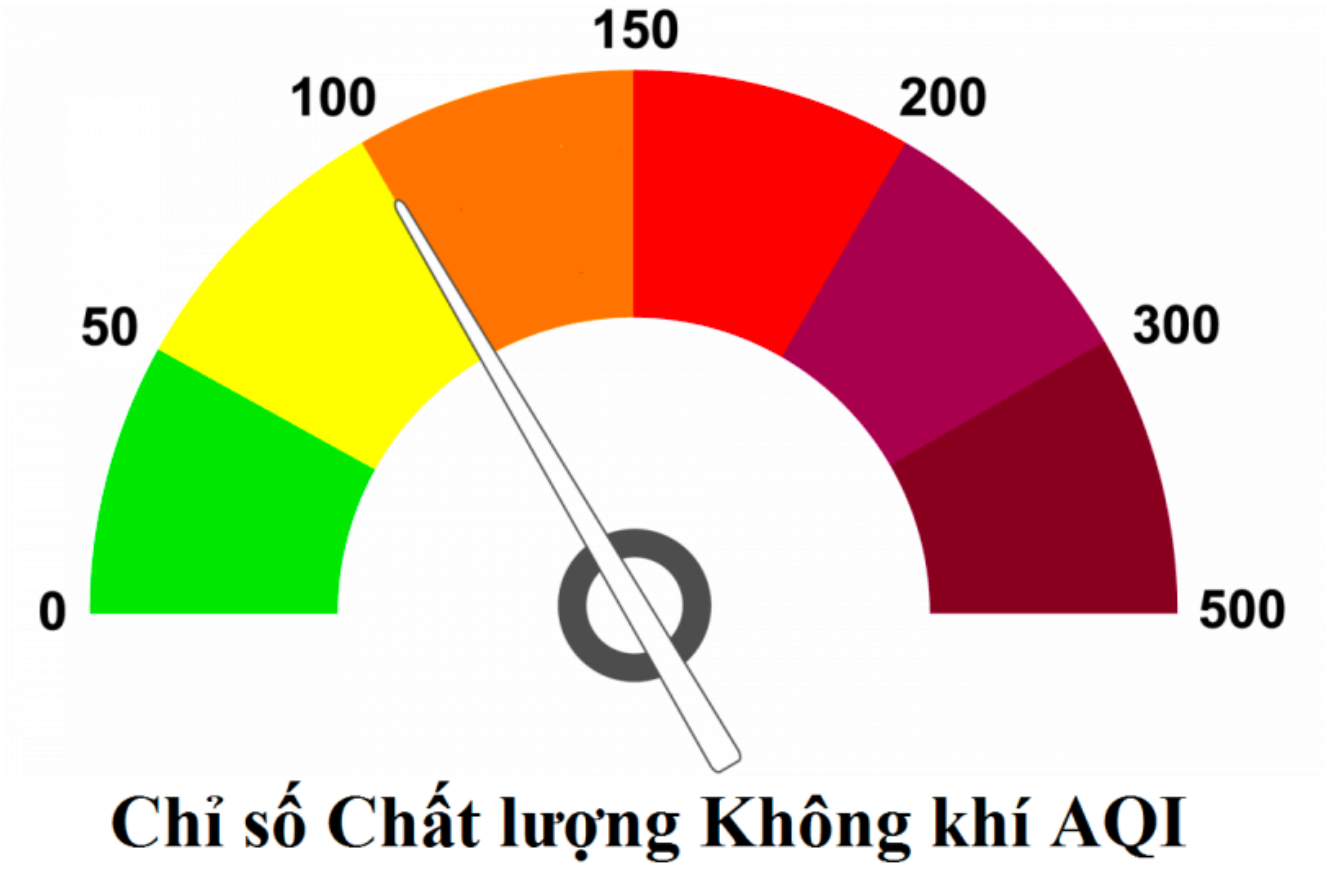
Normally, to solve this problem, we can simply use ThingsBoard to create alarm to devices. But our team have designed that our system should handle a specific mission. In that case, ThingsBoard now have two others jobs, and to solve this problem in our system, we suggest that we should solve it in Spark. Through our study, we found out a formula for the relationship of quantity parameter of pollutant I_p and the air quality, the formula will be:

$$I_p = \frac{I_{H_i} - I_{L_0}}{BP_{H_i} - BP_{L_0}}(C_p - BP(L_0)) + I_{L_0}[12]$$

Where:

- I_p is quantity parameter of pollutant p
- C_p Concentration of pollutant p
- BP_{H_i} upper parameter of C_p
- BP_{H_0} lower parameter of C_p

After calculated AQI, we have a chart that show the air quality base on AQI:



Apply this formula, so in the incoming data, we filter stations that exceed the limit value. After that, we save it to a file of stations and send it back to a station center, station center can contact each station that have bad air quality and trigger an alarm. Here is our implement in Spark for "PM2.5" and same on other parameter:

```
val alarm =
    subDF
      .filter(aqi(col("`PM2.5`")) > 150)
      .groupBy("StationCode")
```

3.4 Improvements

3.4.1 Nginx - HAProxy

One major security hole in this system design is that the connection between sensors and ThingsBoard cluster doesn't use an encryption (for instance: TLS/SSL, ...). Even though our sensors have token for authentication, others may use packet sniffer to capture these unencrypted messages.

Also, our ThingsBoard cluster only expose **1** port for external connection. As our system grows, we need a way to balance the load between our Docker-services.

Common load balancer such as Nginx or HaProxy can perform these tasks seamlessly.[13]

3.4.2 Prometheus - Grafana

In contrast to the default UI managing tools provided by ThingsBoard and Spark, these real-time monitoring tools provide I.T. manager with dashboards and reports that are simpler and more robust to use with.

Prometheus and Grafana unify all the logs and data from different sources and system. When implemented, system manager can monitor the entire system without switching between components

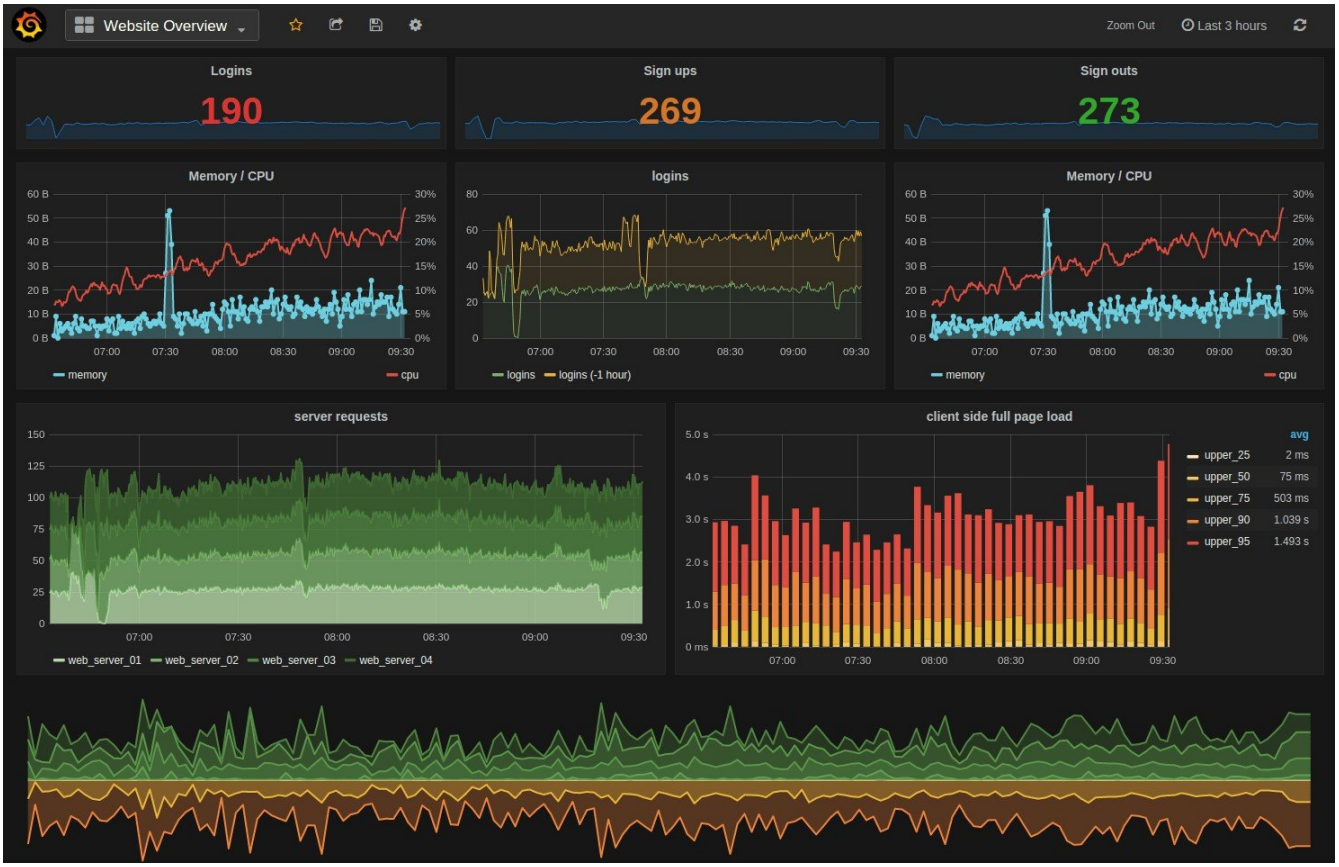


Figure 3.26: Sample Grafana windows

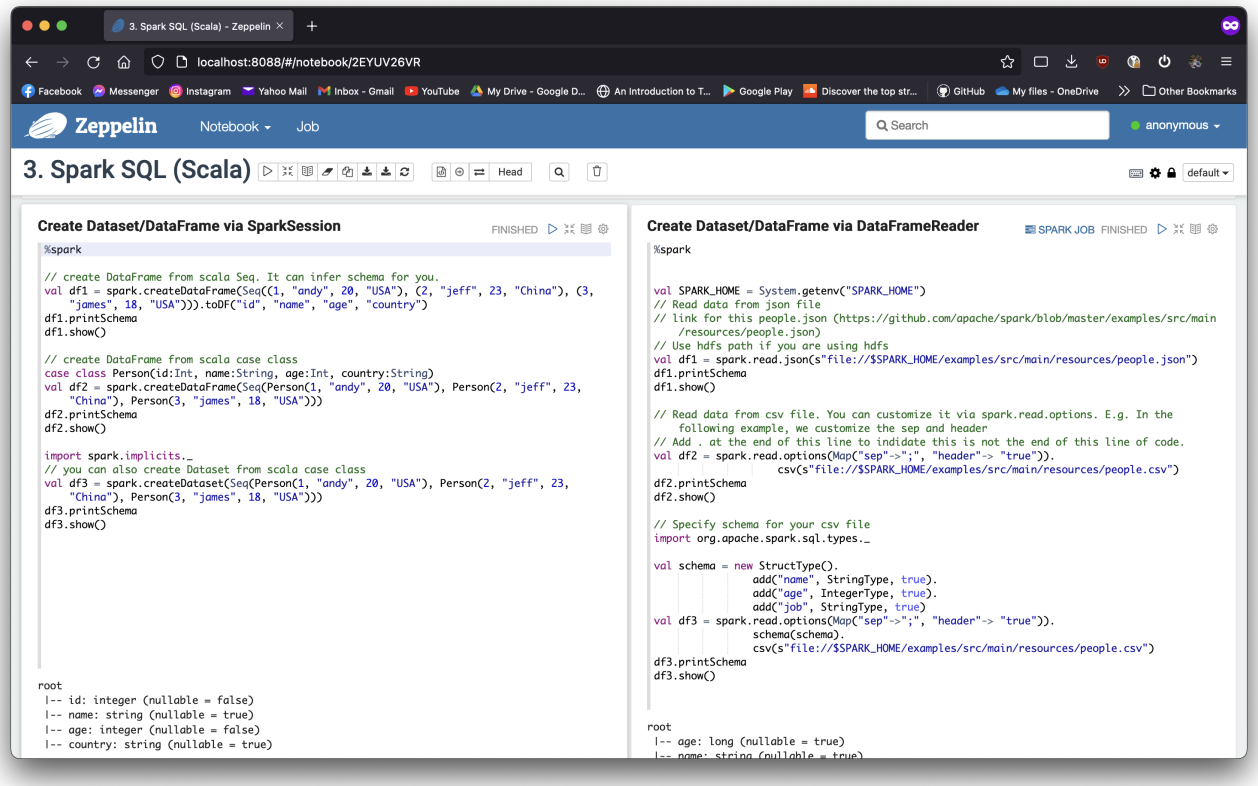


Figure 3.27: How our Zeppelin notebook will work

3.4.3 Zeppelin

As we only install Spark core on our HDFS-Spark cluster, Data Scientist / Business Analyst will have a hard time using it to perform further data aggregation tasks.

Zeppelin provides a cleaner and more user-friendly development environment.

Bibliography

[1] C. Wang. "HTTP vs. MQTT: A tale of two IoT protocols," Google. (Nov. 2018), [Online]. Available: https://cloud.google.com/blog/products/iot-devices/http-vs-mqtt-a-tale-of-two-iot-protocols.

[2] F. Colasante. "How to setup an iot system using thingsboard." (), [Online]. Available: https://colasante-francesco.medium.com/how-to-setup-an-iot-system-using-thingsboard-b705c9189e37.

[3] H. N. 97. "Bai 1: Gioi thieu ve thingsboard iot platform," lophocvui.com. (), [Online]. Available: https://lophocvui.com/iot-internet-of-things/gioi-thieu-ve-thingsboard-iot-platform/.

[4] S. A. Alavi. (), [Online]. Available: https://www.researchgate.net/figure/Developed-web-based-dashboard-using-the-Thingsboard-IoT-platform_fig2_333600103.

[5] "Getting stated with rule engine," Thingsboard authors. (), [Online]. Available: https://thingsboard.io/docs/pe/user-guide/rule-engine-2-0/re-getting-started/.

[6] "What is thingsboard," Thingsboard authors. (), [Online]. Available: https://thingsboard.io/docs/pe/getting-started-guides/what-is-thingsboard/.

[7] Amazon. "What is Pub/Sub messaging?" (), [Online]. Available: https://aws.amazon.com/pub-sub-messaging/.

[8] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide*. O'Reilly Media, 2017, ch. 1, ISBN: 9781491936160.

[9] E. Vinka. "What is Zookeeper and why is it needed for Apache Kafka?" CloudKarafka. (), [Online]. Available: https://www.cloudkarafka.com/blog/cloudkarafka-what-is-zookeeper.html.

[10] D. Borthakur. "Hdfs architecture guide," Apache. (), [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[11] N. Q. Huy. "HDFS," De Manejar Team. (), [Online]. Available: https://demanejar.github.io/posts/hdfs-introduction/.

[12] "Chi so chat luong khong khi aqi đc tinh nhu the nao?" VCAP. (), [Online]. Available: http://vietcleanair.vn/chi-so-chat-luong-khong-khi-aqi-duoc-tinh-nhu-the-nao/?fbclid=IwAR2tOry6c5ptxJKDr_HaLcQsolulDHA04dwp3EBa0QRQonz3YUdoTALenks.

[13] R. Nelson. "Docker swarm load balancing with nginx and nginx plus," F5, Incl. (), [Online]. Available: https://www.nginx.com/blog/docker-swarm-load-balancing-nginx-plus/.

[14] R. A. Atmoko and R. Riantini. "Iot real time data acquisition using mqtt protocol." (), [Online]. Available: https://www.researchgate.net/publication/317391853_IoT_real_time_data_acquisition_using_MQTT_protocol.

[15] "Spark overview," Apache Spark. (), [Online]. Available: https://spark.apache.org/docs/latest/index.html.

[16] "Structured streaming + kafka integration guide," Apache Spark. (), [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html.