TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

THESIS NO: 073/MSCS/651

Integrating Message Queuing Telemetry Transport (MQTT)with Kafka Connect for Processing IOT data

by

Anila Kansakar

A THESIS

SUBMITTED TO

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SYSTEM AND KNOWLEDGE ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

LALITPUR, NEPAL

November, 2019

Integrating Message Queuing Telemetry Transport (MQTT)with Kafka Connect for Processing IOT data

by

Anila Kansakar

073/MSCS/651

Thesis Supervisor:

Prof. Dr. Subarna Shakya

A thesis submitted in partial fulfillment of the requirement for

the degree of Master of Science in Computer System and Knowledge Engineering.

Department of Electronics and Computer Engineering

Institute of Engineering, Pulchowk Campus

Tribhuvan University

Lalitpur, Nepal

November, 2019

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a project report entitled "**Integrating Message Queuing Telemetry Transport (MQTT)with Kafka Connect for Processing IOT**" submitted by **Anila Kansakar** in partial fulfillment of the requirements for the degree of "**Master of Science in Computer System and Knowledge Engineering"**.

Supervisor: Prof. Dr. Subarna Shakya
Professor
Department of Electronics and Computer Engineering

External Examiner: Prof. Manish Pokharel
Professor
Kathmandu University

Committee Chairperson: Dr. Aman Shakya
Assistant Professor
Department of Electronics and Computer Engineering

Date: November, 2019

त्रिभुवन विश्वविद्यालय
Tribhuvan University
इन्जिनियरिङ्ग अध्ययन संस्थान
Institute of Engineering
पुल्चोक क्याम्पस
Pulchowk Campus
इलेक्ट्रोनिक्स तथा कम्प्युटर इन्जिनियरिङ्ग विभाग
Department of Electronics and computer Engineering

553 4070
552 1260
Extn: 315
Fax: 977 1 555 3946
Email: ece@ioe.edu.np
http://doece.pcampus.edu.np
Pulchowk, Lalitpur
P.O Box: 1175

DEPARTMENTAL ACCEPTANCE

The thesis entitled "**Integrating Message Queuing Telemetry Transport (MQTT)with Kafka Connect for Processing IOT**", submitted by **Anila Kansakar** in partial fulfillment of the requirement for the award of the degree of "**Master of Science in Computer System and Knowledge Engineering**" has been accepted as a bona fide record of work independently carried out by her in the department.

Assoc. Prof. Dr. Surendra Shrestha
Head of the Department
Department of Electronics and Computer Engineering,
Pulchowk Campus,
Institute of Engineering,
Tribhuvan University,
Nepal.

# ACKNOWLEDGEMENT

I would like to express the deepest appreciation to my supervisor Prof. Dr. Subarna Shakya for his guidance throughout the period of this work. His invaluable support, understanding and expertise have been very important in completing this work. It was a great honor for me to pursue my thesis under his supervision.

I am grateful to Dr. Surendra Shrestha our Head of Department of Electronics and Computer Engineering, Pulchowk Campus for his support.

I pay my sincere gratitude Dr. Aman Shakya, to MSCSKE Coordinator for his supervision and help during this research work.

I am highly grateful to Prof. Dr. Shashidhar Ram Joshi, Dr. Dibakar Raj Pant , Dr. Sanjeeb Prasad Pandey, and Dr. Basanta Joshi for their encouragement and guidance.

I would like to express my heartily gratitude towards the Institute of Engineering, Pulchowk Campus along with all my respected teachers, my friends, my family for giving me continuous support for their invaluable help.

Anila Kansakar

073/MSCS/651

ABSTRACT

The Internet of Things (IoT) architecture is defined as a layered structure in which each layer represents a coherent set of services. For supporting the communication among the different IoT entities many different communication protocols are now available in practice. For practitioners, it is often not clear which communication protocol is suitable for the various conditions in which the IoT systems need to be operated. The backbone of Internet of Things (IoT) is the communication protocols which seamlessly integrate thousands of nodes and enable a light weight data transfer process. This research is to analyze the efficiency and applicability of different Machine to Machine (M2M) protocols that are available for IoT communication. This thesis aims at exploring the capabilities of such middleware and how they can be integrated in real world application need to aggregate data on a large scale. MQTT and Kafka are two complementary technologies. Together they allow to build IoT end to end integration from the edge to the data center. Kafka Connect is a part of Apache Kafka and provides a scalable and reliable way to move data between Kafka and other datastores.

Keywords: IOT, REST API, MQTT, Kafka Connect, Source Connector, Sink Connector

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Abbreviations

ETL	Extract, Transform and Load
ICT	Information and Communication Technology
IOT	Internet of Thing
HTTP	Hypertext Transport Protocol
M2M	Machine to Machine
MQTT	Message Queuing Telemetry Transport

CHAPTER 1: Introduction

1.1 Background

Over the past decades, an effort has been made by the information and communications technology industries to continuously increase the number of Internet enabled devices. These devices, besides the traditional computers and mobile devices, are devices that ranges from home or domestic appliances, industrial machinery and automation, healthcare, transport, energy, buildings, cities and people are been connected to the Internet. Adding more devices, which were traditionally offline to the Internet, has become possible or feasible due to the technological advancement with the hardware, software developments and the idea of network convergence known as the Internet Protocol (IP) convergence. This avalanche of many new devices and other things being connected to the Internet was known as the evolution of the Internet, which is nowadays termed as the Internet of Things (IoT).

The main idea of Next generation internet devices is to connect things that are not yet connected to the Internet and to provide interconnectivity between other devices and the things to the global information and communications infrastructure. This interconnectivity of things will allow not only communication between devices and things, but it will offer intelligence to the things being connected and makes their data available to other network systems to utilize.

However, different devices from different manufacturers having different hardware platforms and networking protocols exist within the IoT, which makes it heterogeneous network of things. The interaction or interoperability with diverse devices from different manufacturers with different service platforms and networks need to be adapted to realize IoT applications. Moreover, the IoT networks could be complex due to the dynamic state of some devices and the things within the IoT. This means that some connected devices can change their states from, for example, sleeping to waking up, connected to disconnected as well as in the context of a device location and speed. The number of connected devices can change dynamically at any particular time which means that the number of devices that need to be managed will be of enormously high scale. Data collection and management from different sources is also critical to IoT applications [1].

1.2 Problem Statement

There are different number of protocols that could be used to communicate between a internet devices. Fundamental challenges is to choose the appropriate protocol for their specific IoT system requirements to address real-time processing, fast data response, and latency issues.

On technical perspective, how to connect the edge i.e. IOT devices and there may be gateway in the middle. On another hand, choosing the streaming platform i.e. Apache Kafka deployment. One of the most important factors is how to integrate end to end IOT data integration and processing in scalable manner.

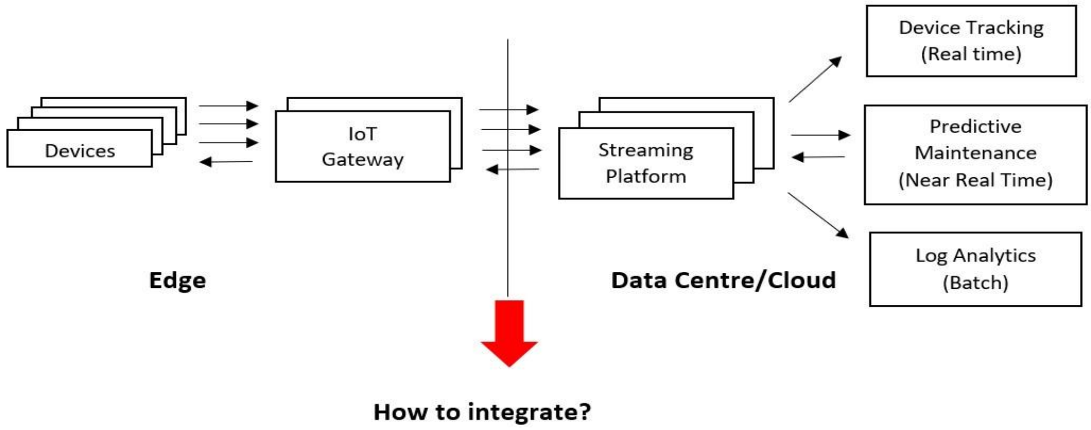

Figure 1. 2: Edge to Edge integration

IOT standard protocol MQTT architecture doesn’t support scale. It is not distributed system It is just a queuing system. it doesn’t handle streaming processing. It is not built for high scalability and reliability system for 24*7 transaction system. From IOT perspective, we need stream processing platform which give high throughput, large scale and high availability.

1.3 Objectives

The main objectives of this thesis are

- To designed Kafka Connect system to handle IOT data from MQTT broker Kafka broker.
- To implement Source and Sink Connector for Kafka Connect.
- To compare latency between traditional method i.e. using MQTT broker and new approach method i.e. using MQTT broker with Kafka Connect.

CHAPTER 2: Literature Review

Many researches have been proposed on Internet of Things.

There are many studies on using publish-subscribe messaging as means of communication for resource constrained devices [1] discusses on the use of publish subscribe as communication protocol for Wireless Sensor Networks (WSN). In WSN, sensors and actuators may change network address at any time, network links are likely to fail, and failed WSN nodes are replaced by new nodes. As Publish-subscribe is asynchronous and it does not need to know about the existence of other endpoints in the network, it is best suited for WSNs. It is common and widely used variant of data-centric communication

C. Rodríguez-Domínguez [2] analyzes both request-response and publish-subscribe as communication model in ubiquitous systems. The integration of request-response and publish subscribe communication model is discussed to fulfill the need for the system that requires features of both. The simultaneous use of request-response and publish subscribe is proposed for easy development of software solutions that required both synchronous and asynchronous communications. This paper brings the concept of using both request response and publish-subscribe as a solution to implement benefits of both approaches, with higher abstraction level which is technology independent.

The author comparing two different protocols, but comparing two different message passing mechanisms, one is MQTT publish-subscribe protocol and other is the REST architectural style. This paper discusses how to use MQTT as the protocol for IoT application deployment and remote management of those applications, such that it can work in all network conditions [3].

Different protocols may be appropriate for different situations regarding the necessities of the IoT system. D. Thangavel [4] discusses on the comparison of different lightweight protocols regarding the data transmission time from endpoints and the bandwidth consumption in the IoT system.

Omer Koksal,Bedir Tekinerdogam [5] focus on the session layer which is responsible for setting up and taking down of the association between the IoT connection points. The session layer provides services related issues of the session such as initiation, maintenance, and disconnection. As such, frequency and duration of various types of sessions are related with the session layer. Also, session information might enforce encryption and other security measures. They adopt a feature-driven domain analysis whereby they have identified the important knowledge sources and extracted and modeled the important features of the session layer communication protocols. The result of the domain analysis process, as such, is a feature model that defines the common and variant properties of the session layer communication protocols.

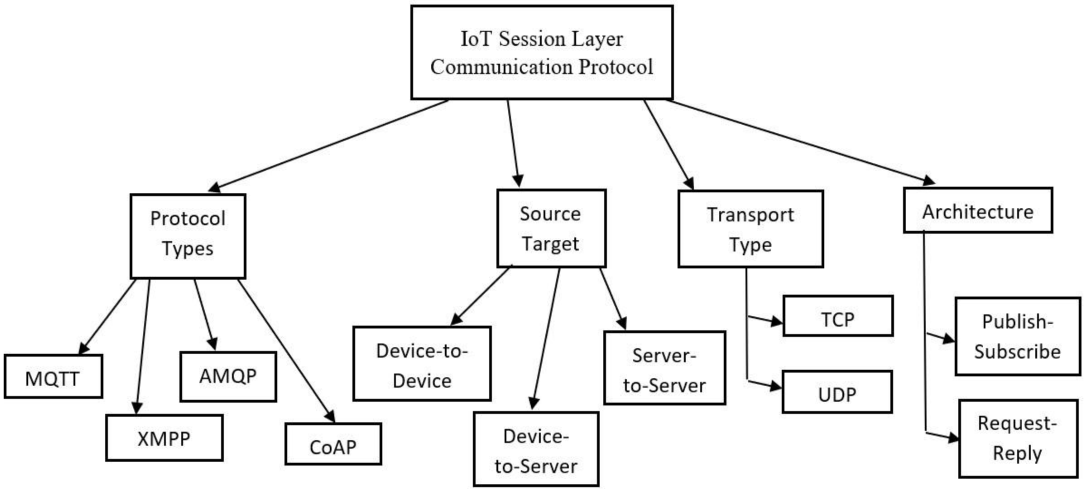

Figure 2.3: Feature Diagram of Session Level Communication Protocols of IoT

Rishika Shree [6], comparison between Apache Flume and Apache Kafka. Kafka can be used when you particularly need a highly reliable and expandable enterprise messaging system to connect many multiple systems. Kafka is capable to make pipeline for activities like a set or group for publish/subscribe which is actually real time which consequently means that whatever activities going on the site is published to topics which is central that includes for each activity there is one topic. Kafka is capable to serve and can be used for external commit-log for a system which is distributed.

The writer develops Dual Streaming Model. The idea of this paper is to specify the result of a stream processing operator as successive updates to a table so that latency of streaming processing is not compromised. To handle of out -of-order data directly in the stream processing model has further advantages. This paper discussed how model makes explicit the trades-off between result completeness, processing cost and latency in data stream processing environment. Finally, they presented an implementation of the Dual Streaming Model in Apache Kafka, a widely adopted open source stream processing platform [7].

Kafka's Origin- Kafka was created to address the data pipeline problem at LinkedIn. It was designed to provide a high-performance messaging system that can handle many types of data and provide clean, structured data about user activity and system metrics in real time[8].

Anindya Dey [9], explored the impact of alternative real time streaming topologies within the edge server of IoT analytical systems. The author evaluated these topologies in terms of the time to insight from our machine learning models as well as the quality of predictions. There results show that topology impacts stream processing in multiple ways and real-world parameters like missing values, out of order arrivals, varying sparsity have a significant impact as we scale up the density of sensor deployments

CHAPTER 3: Related theory

3.1 Message Queuing Telemetry Transport (MQTT)

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight messaging protocol that provides resource constrained network clients with a simple way to distribute telemetry information. The protocol, which uses a publish/subscribe communication pattern, is used for machine-to machine (M2M) communication and plays an important role in the internet of things (IoT). MQTT enables resource-constrained IoT devices to send, or publish, information about a given topic to a server that functions as an MQTT message broker. The broker then pushes the information out to those clients that have previously subscribed to the client's topic. To a human, a topic looks like a hierarchical file path. Clients can subscribe to a specific level of a topic's hierarchy.

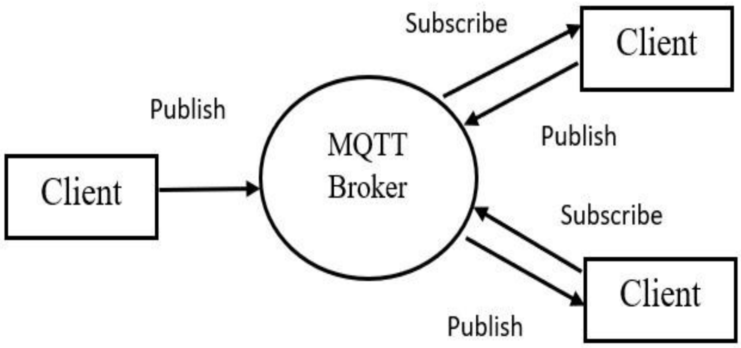

Figure 3.4 :MQTT Broker process

How MQTT works

An MQTT session is divided into four stages: connection, authentication, communication and termination. A client starts by creating a TCP/IP connection to the broker by using either a standard port or a custom port defined by the broker's operators. When creating the connection, it is important to recognize that the server might continue an old session if it is provided with a reused client identity.

Because the MQTT protocol aims to be a protocol for resource-constrained and IoT devices, SSL/TLS might not always be an option and, in some cases, might not be desired. In such cases, authentication is presented as a clear-text username and password that is sent by the client to the server as part of the CONNECT/CONNACK packet sequence. Some brokers, especially open brokers published on the internet, will accept anonymous clients.

In such cases, the username and password are simply left blank.

MQTT is called a lightweight protocol because all its messages have a small code footprint. Each message consists of a fixed header 2 bytes an optional variable header, a message payload that is limited to 256 MB of information and a quality of service (QoS) level.

The three different quality of service levels determine how the content is managed by the MQTT protocol. Although higher levels of QoS are more reliable, they have more latency and bandwidth requirements, so subscribing clients can specify the highest QoS level they would like to receive.

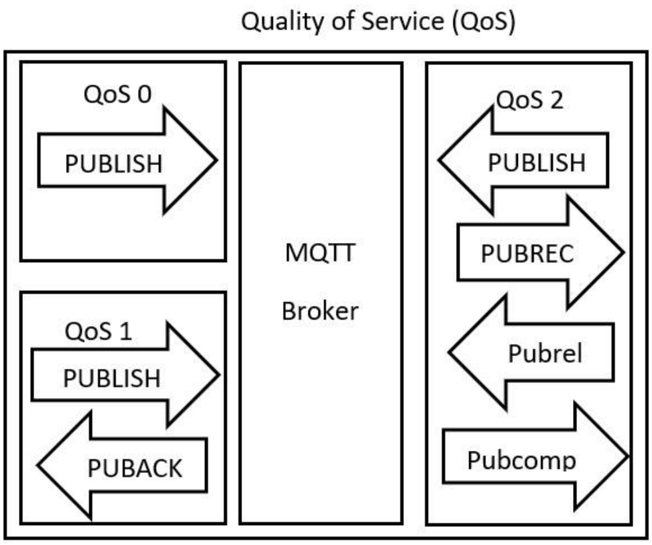

Figure 3. 2: Three different level of QoS of MQTT

The simplest QoS level is unacknowledged service. This QoS level uses a PUBLISH packet sequence; the publisher sends a message to the broker one time and the broker passes the message to subscribers one time. There is no mechanism in place to make sure the message has been received correctly, and the broker does not save the message. This QoS level may also be referred to as at most once, QoS0, or fire and forget.

The second QoS level is acknowledged service. This QoS level uses a PUBLISH/PUBACK packet sequence between the publisher and its broker, as well as between the broker and subscribers. An acknowledgement packet verifies that content has been received and a retry mechanism will send the original content again if an acknowledgement is not received in a timely manner. This may result in the subscriber receiving multiple copies of the same message. This QoS level may also be referred to as at least once or QoS1

The third QoS level is assured service. This QoS level delivers the message with two pairs of packets. The first pair is called PUBLISH/PUBREC, and the second pair is called PUBREL/PUBCOMP. The two pairs ensure that, regardless of the number of retries, the message will only be delivered once. This QoS level may also be referred to as exactly once or QoS2[10].

Table 3.1: Description of MQTT Message

MQTT Message	Description
CONNECT	Client request to connect to server
CONNACK	Connect Acknowledgement
PUBLISH	Publish message
PUBACK	Publish Acknowledgement
PUBREL	Publish release
PUBCOMP	Publish complete
PUBREC	Publish received
SUBCRIBE	Client subscribe request
UNSUBSCRIBE	Unsubscribe request
UNSUBACK	Unsubscribe Acknowledgement
DISCONNECT	Client is disconnecting

3.2 Apache Kafka

Apache Kafka is a publish/subscribe messaging system. It is often described as a “distributed commit log” or more recently as a “distributing streaming platform.” A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Messages in Kafka are categorized into topics. Topics are additionally broken down into a number of partitions. Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server.

A single Kafka server is called a broker. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.

Kafka Connect is a part of Apache Kafka and provides a scalable and reliable way to move data between Kafka and other datastores. It provides APIs and a runtime to develop and run connector plugins libraries that Kafka Connect executes and which are responsible for moving the data. Connectors start additional tasks to move large amounts of data in parallel and use the available resources on the worker nodes more efficiently. Source connector tasks just need to read data from the source system and provide Connect data objects to the worker processes. Sink connector tasks get connector data objects from the workers and are responsible for writing them to the target data system [8].

CHAPTER 4: Research Methodology

4.1 System Design

In this system design, MQTT and Kafka are two complementary technologies. Together they allow to build IoT end to end integration from the edge to the data center. Therefore, MQTT and Kafka are a perfect combination for end to end IoT integration from edge to data center. As shown in Figure 4.1, different sensor data like temperature, pressure, CO2, humidity and location are taken. these IoT data are passed through MQTT protocol. MQTT protocol used different types of broker. In this system Mosquitto broker is used. A MQTT connector to read the data from MQTT and push them to Kafka. In Kafka connect, connector implementation for data sources and sinks to move data into and out of Kafka. A source connector ingests entire databases and stream tables update to Kafka topics. It can also collect data from our servers into Kafka topics, making the data available for stream processing with low latency. A sink Connector deliver data from Kafka topic into Kafka consumer. Kafka connect is focused on streaming data to and from Kafka, making it simpler for high quality, reliable and high-performance connector plugin. Kafka Connect is integral component of an ETL pipeline when combined with Kafka and streaming framework.

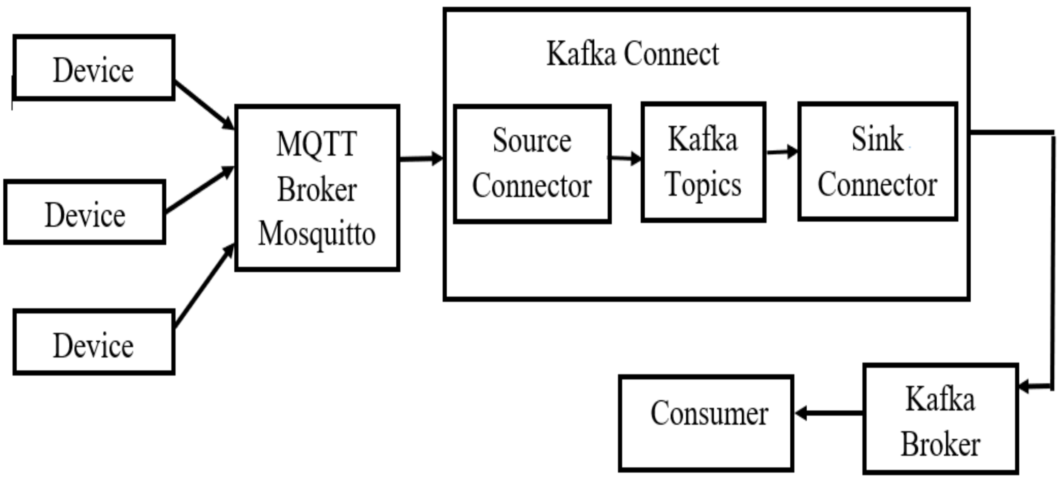

Figure 4. 3: System Architecture of MQTT with Kafka connect

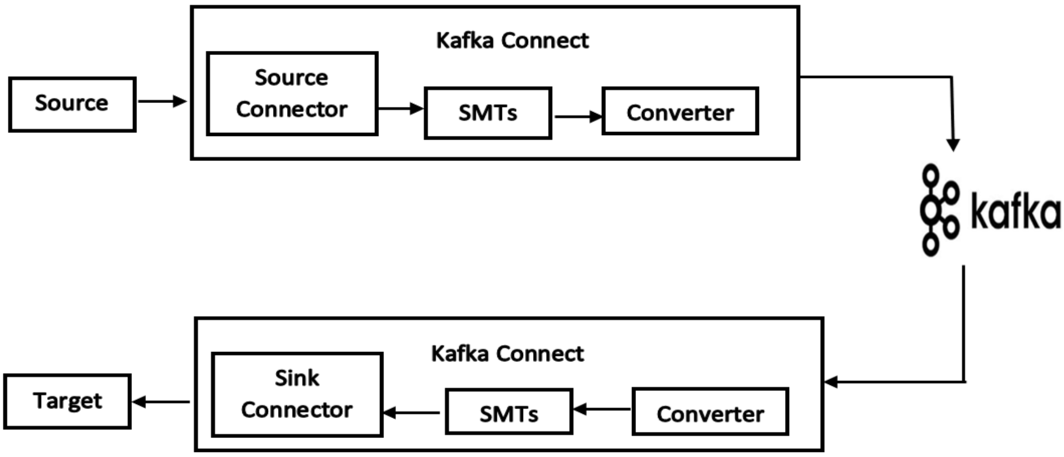

Figure 4. 2: System Architecture of Kafka connect

Sensors Requirements:

Detail Technical Specification

- WIND SPEED SENSOR:
    - 3 Levels – l sensor (primary and redundant at each)
    - 30m and 50 m level and primary sensor at 20 m height)
    - Sensor: 3 cup rotor polycarbonate
    - Range: up to 75 m/sec
    - Accuracy: ±0.3m/sec (= 10m/sec)
    - Resolution: 0.8 m/s or better
    - Distance constant: ~0.3 m/sec
    - Cup diameter (approx.): 60 mm or less
    - Power supply: 1.5-5V DC
    - Sensor Type: Hall Effect sensor(A3141) with 3 cup rotor

- AIR TEMPERATURE SENSOR:
    - Range: -20 C to + 60 C
    - Accuracy: ±0.2 C
    - Radiation Shield: Non-Aspirated Radiation
    - Resolution in degree: 0.1 C
    - Power supply: 1.5-5V DC
    - Material: Conducting epoxy casing

- Sensor Type: DHT 11 Humidity & Temperature sensor

- AIR PRESSURE SENSOR:
    - Sensor: Absolute Pressure Sensor
    - Range: 15 kPa – 115 kPa
    - Output: Analog (or Digital with SCM)
    - Resolution: Absolute Pressure in kPa = (Voltage x 21.79) + 10.55 typical
    - Accuracy: 1.5 kPa (15 mb) max.
    - Uncorrected offset (+/- 0.443 inches Hg)
    - Power Supply: 3 V to 35 V
    - Enclosure: Weather Proof
    - Sensor type: absolute pressure sensor BP-20
- RELATIVE HUMIDITY SENSOR:
    - Relative/Absolute Humidity Range: 0 to 100 %
    - Accuracy: ±2 % (0 – 90%)
    - Resolution: 0.7% Radiation Shield: Non-Aspirated Radiation Shield
    - Output: Analog (or Digital with SCM)
    - Power supply: 3 – 35 5 V DC
    - Sensor Type: DHT 11Humidity & Temperature sensor
- SOLARRADIATION:
    - Sensor: Solar Radiation Spectral response: 0.3 - 3 microns
    - Operating temperature: - 10 - 50o C
    - Shield: Weatherproof
    - Sensitivity/output: ~0.1 m/mw/cm2
    - Range: 0 - 2 kW/m2
    - Wave Length: 0.3–2.9µm
    - Resolution:0.1W/m2
    - Sensor type: High-stability silicon photovoltaic detector (blue enhanced).

4.2 Source connector and Sink connector mechanism

In this system, the data from MQTT can't communicated with Kafka server. So, a Kafka connected with all the modification to build a connector which can communicated with both the system. A Kafka connected come in two flavors. One for input and another for output. So, source connectors are built for handling input data, and sink connectors for output. For example, and “Mqtt91.sourceconnector” would import a data into Kafka server and “Mqtt91.sinkconnector” would export the data of Kafka topic to consumer.

A connector is responsible for breaking the job into a set of Tasks that can be distributed to Kafka connect works. Tasks also come into two type Source task and Sink task. A task must copy its subset of the data to or from Kafka. The data that a connector copies must be represented as a partitioned stream, and to each Kafka topic.

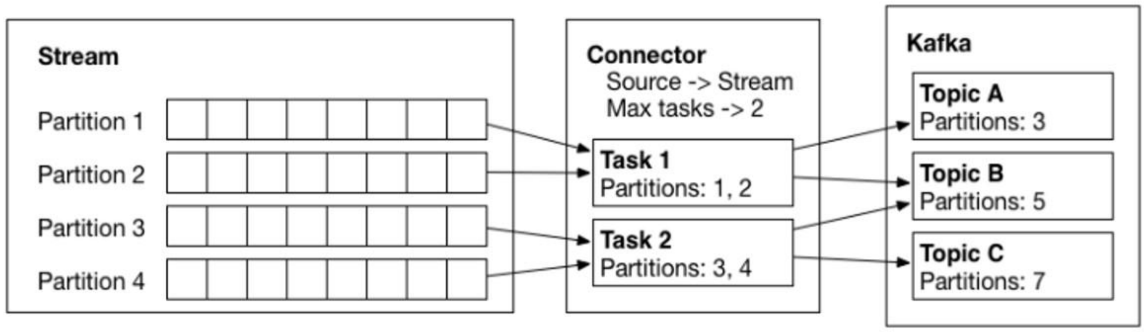

Figure 4. 3: Source Connector which has created two tasks which copy data from input partition and write record to Kafka

Task

It’s is the main component for our connector. Each connector instance coordinates a set of tasks that actually copy the data. This breaking of jobs allows the Kafka to support for parallelism and scalable data copying with little configuration. The tasks state is stored in specify topics i.e. “config.storage.topic”.”

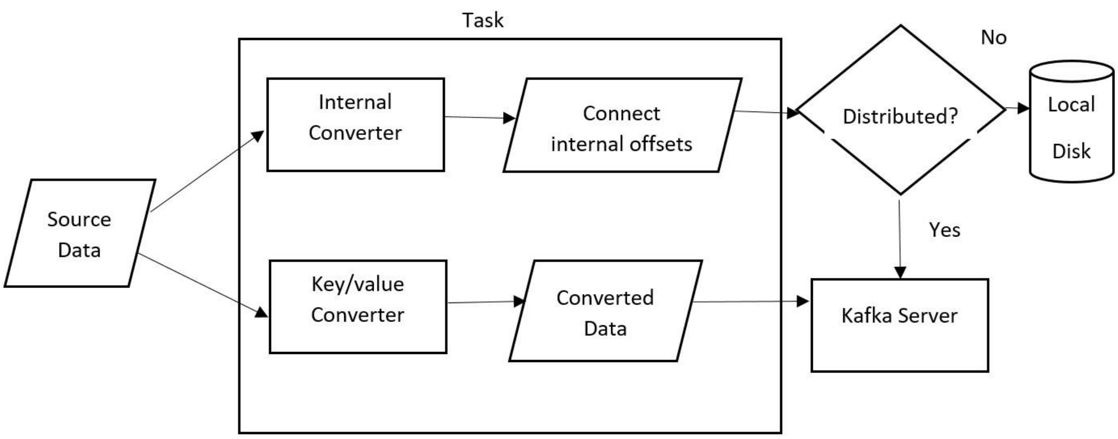

Figure 4. 4: Representation of data passing through a connector source task into Kafka

Converter

- Converters are necessary to have a Kafka Connect deployment support a data format when writing to or reading from Kafka.
- Tasks use converters to change the format of data from bytes to Connect internal data format and vice versa.

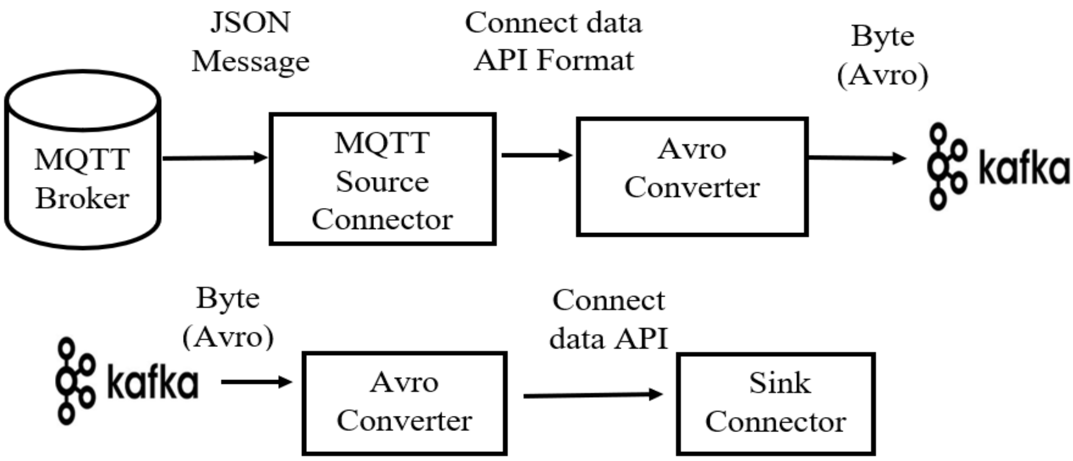

Figure 4. 4: Process of converting JSON data type to Avro data type

Kafka Connect- Connector and Tasks lifecycles

- Validate configuration
- Completely configure driven
- Deploy the connector & run code start (…)
- Poll(..) function read the data

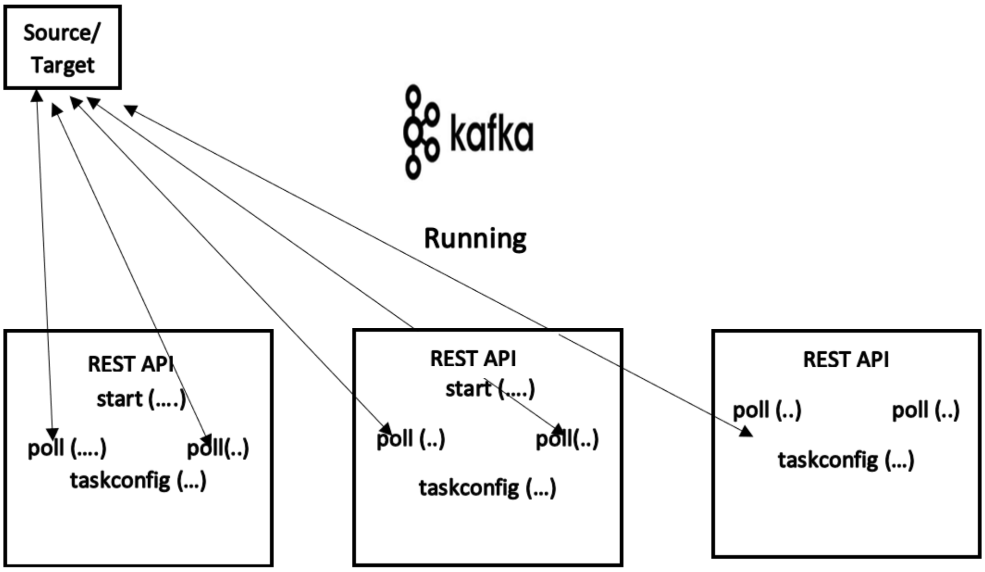

Figure 4. 4: Process of Kafka Connect- Connector and Tasks lifecycles

4.3 Code Architecture

Taking advantage of Java's feature like interfaces, the implementation has been developed in modular way to generic cod processing from broker-specific operations.

- BusConfig

Singleton class used to load and provide other class setting read from the configuration file.

- MessagePusher

Interface that provides a line to the configuration singleton and defines the two primary method that all message pusher should implement: pushmessage(Message) and shutdown().Wherever there is need to open a connection toward a message broker, it should have to reference to MessagePusher object ie KakfaPusher or MQTTPusher.

- KafkaPusher

Class that has all the logic to push message to a Kafka broker. Each instantiation results in the creation of a new TCP connection to the broker that will push messages to the specified topic name and using the KafkaMessagePartitioner partition chooser.

- KafkaMessagePartitioner

Class used by KafkaPusher to decide to which partition of a topic should a message be sent.

- MQTTPusher

Class that has all the logic to push message to Mosquitto MQTT broker. Each instantiation results in the creation of a new TCP connection to the broker, dedicated to the queue whose name is given as argument when the object is constructed. • Message

Model class that defines the structure of message's content.

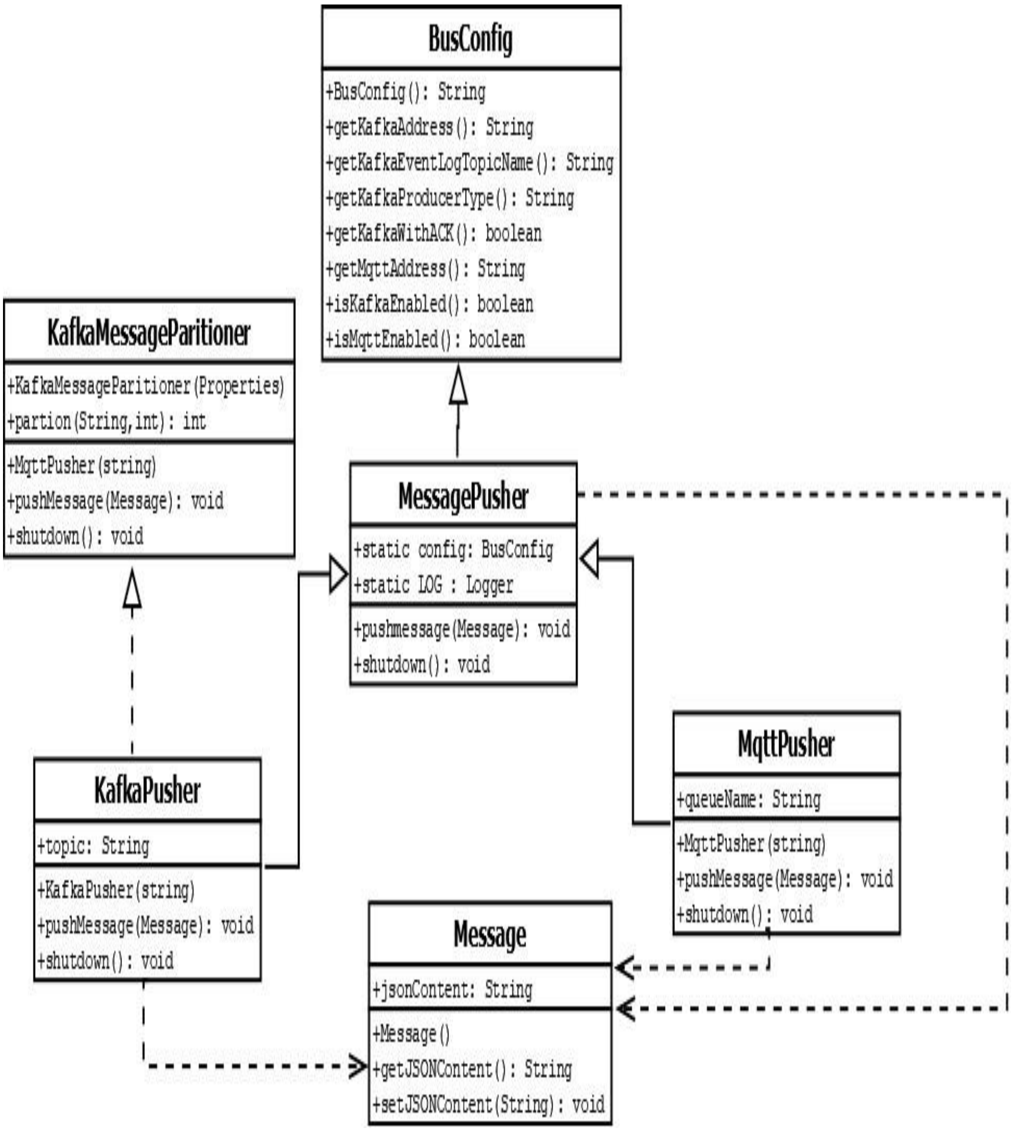

Figure 4. 5: Class diagram of Kafka Connect

4.4 Flowchart of Source and Sink Connector Mechanism

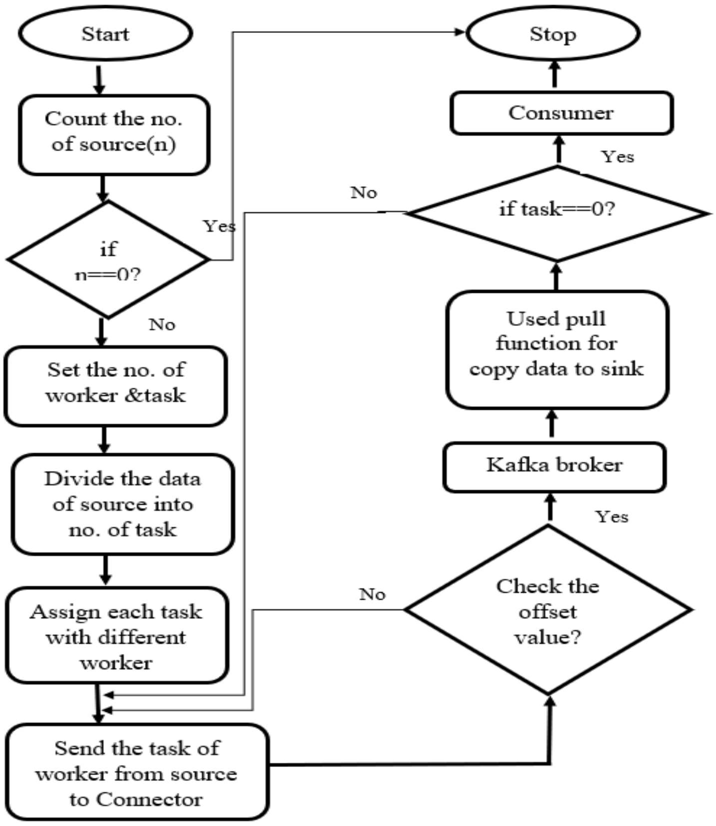

Figure 4. 6: Flowchart of Source and Sink Connector Mechanism

4.5 Request Processing

Most of what a Kafka broker does is process requests sent to the partition leaders from clients, partition replicas, and the controller. Kafka has a binary protocol that specifies the format of the requests and how brokers respond to them both when the request is processed successfully or when the broker encounters errors while processing the request. Clients always initiate connections and send requests, and the broker processes the requests and responds to them. All requests sent to the broker from a specific client will be processed in the order in which they were received this guarantee is what allows Kafka to behave as a message queue and provide ordering guarantees on the messages it stores.

All requests have a standard header that includes:

• Request type (also called API key)

• Request version (so the brokers can handle clients of different versions and respond accordingly)

• Correlation ID: a number that uniquely identifies the request and also appears in the response and in the error logs (the ID is used for troubleshooting)

• Client ID: used to identify the application that sent the request

The network threads are responsible for taking requests from client connections, placing them in a request queue, and picking up responses from a response queue and sending them back to clients. Figure 4.5 for a visual of this process [8].

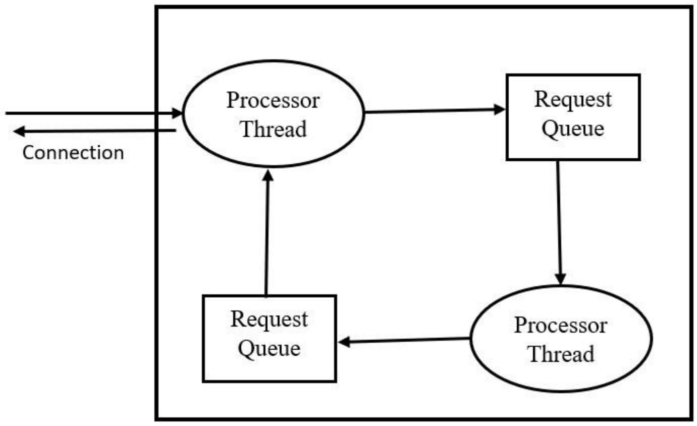

Figure 4. 7: Request Processing inside Apache Kafka

Produce requests

Sent by producers and contain messages the clients write to Kafka brokers.

Fetch requests

Sent by consumers and follower replicas when they read messages from Kafka brokers.

Both produce requests and fetch requests must be sent to the leader replica of a partition. If a broker receives a produce request for a specific partition and the leader for this partition is on a different broker, the client that sent the produce request will get an error response of "Not a Leader for Partition." The same error will occur if a fetch request for a specific partition arrives at a broker that does not have the leader for that partition.

Kafka's clients are responsible for sending produce and fetch requests to the broker that contains the leader for the relevant partition for the request.

4.5 Tools

Different tools and languages to be used for this thesis are discussed in this section.

4.4.1 Mosquitto Broker

Eclipse Mosquitto is an open source message broker that implements the MQTT protocol versions 5.0, 3.1.1 and 3.1. Mosquitto is lightweight and is suitable for use on all devices from low power single board computers to full servers. The MQTT protocol provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for Internet of Things messaging such as with low power sensors or mobile devices such as phones, embedded computers or microcontrollers [16].

4.4.2 Apache Kafka

Apache Kafka is an open-source stream-processing software platform developed by LinkedIn and donated to the Apache Software Foundation, written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Its storage layer is essentially a "massively scalable pub/sub message queue designed as a distributed transaction log, making it highly valuable for enterprise infrastructures to process streaming data. Kafka can also connect to external systems (for data import/export) via Kafka Connect [17].

4.4.3 Wireshark

Wireshark is a Free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education. Wireshark lets the user put network interface controllers that support promiscuous mode into that mode, so they can see all traffic visible on that interface, not just traffic addressed to one of the interface's configured addresses and broadcast/multicast traffic. However, when capturing with a packet analyzer in promiscuous mode on a port on a network switch, not all traffic through the switch is necessarily sent to the port where the capture is done, so capturing in promiscuous mode is not necessarily enough to see all network traffic [18].

4.4.4 Java and Scala Programming Language

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture [19].

Scala combines object-oriented and functional programming in one concise, high-level language. Scala's static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes and build high-performance systems with easy access to huge ecosystems of libraries [20].

4.4.5 Oracle Virtual Box

VirtualBox is a cross-platform virtualization application. It extends the capabilities of existing computer so that it can run multiple operating systems (inside multiple virtual machines) at the same time. It allows to run more than one operating system at a time. Virtual machine (VM) is the special environment that VirtualBox creates for guest operating system while it is running. The key features of oracle virtual box are portability, no hardware virtualization required and guest additions.

4.4.6 Confluent Control Box

Confluent Control Center is a web-based tool for managing and monitoring Apache Kafka®. Control Center facilitates building and monitoring production data pipelines and streaming applications. The use Control Center to manage and monitor Kafka Connect, the toolkit for connecting external systems to Kafka. We can easily add new sources to load data from external data systems and new sinks to write data into external data systems. Additionally, we can manage, monitor, and configure connectors with Control Center. And view the status of each connector and its tasks.

CHAPTER 5: Experimental Outputs

5.1 MQTT Broker and Kafka Connect Setup

Firstly, I setup MQTT Broker by installing Mosquitto broker. Now setting the Mosquito internet protocol 192.168.10.26 is the host internet protocol of operating system and setting the client internet protocol 127.0.0.1 of mosquito broker. The Figure 5.1 shows MQTT Broker creation.

Start the MQTT Broker and test publish / subscribe with 'dummy' topic:

```
brew services start mosquitto
mosquitto_sub -h 127.0.0.1 -t dummy
mosquitto_pub -h 127.0.0.1 -t dummy -m "Hello world"
```

Figure 5. 7: Starting Mosquitto MQTT Broker

5.2 Starting a Kafka

After Connecting MQTT broker, starting Kafka for further processing.

```
confluent local start connect
```

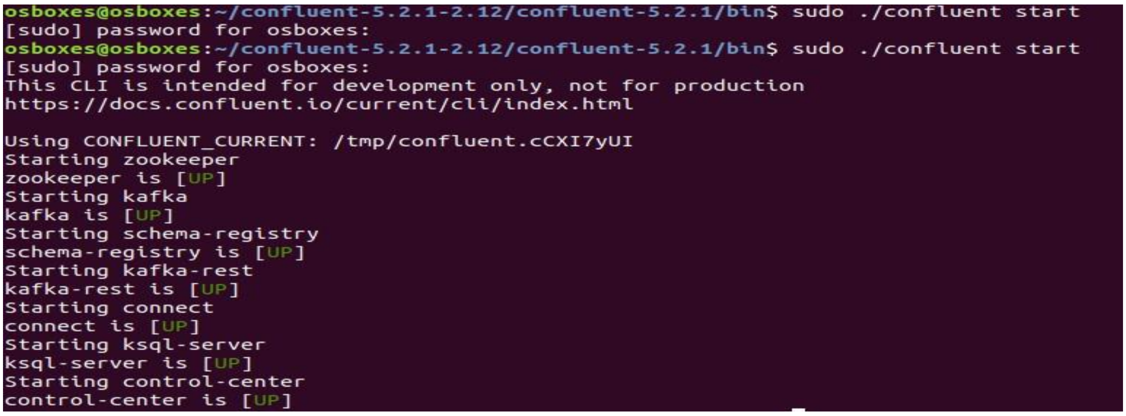

Figure 5. 8: Starting Kafka

5.3 Kafka Connect setup

Building Source connector

To create a custom connector, we need to implement two classes provided by the Kafka connector API connector and task. Our implementation of connector will provide some configuration that describes the data to be ingested. The connector itself will divide the job of ingesting data into a set of tasks and sending those tasks to Kafka Connect workers.

5.4 Topics and Consumer setup

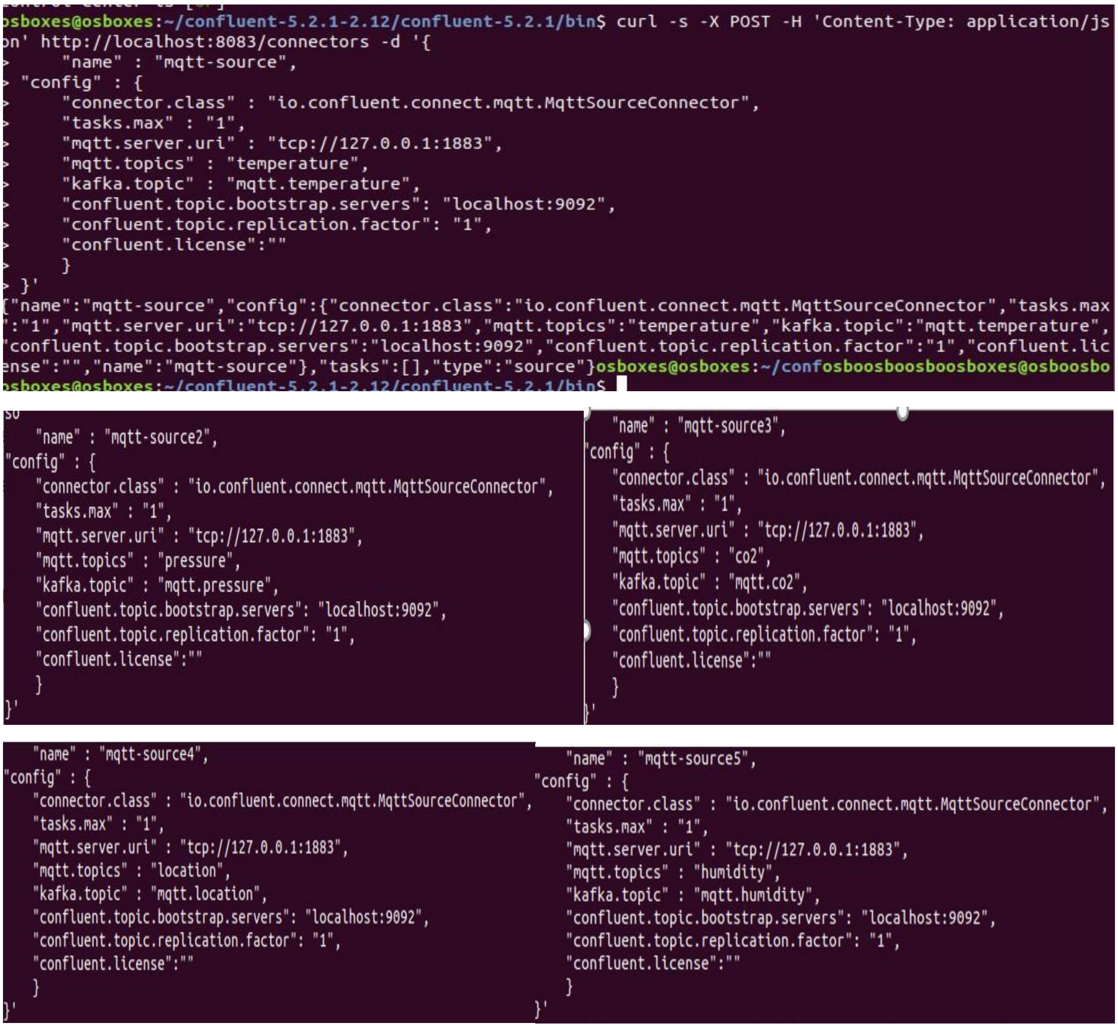

Figure 5. 9: Creating five different sensor value topics

Messages in Kafka are categorized into topics. For experiment, five different sensors are taken so five topics are created as shown in Figure 5.3. Producers create new messages. In other publish/subscribe systems, these may be called publishers or writers. In general, a message will be produced to a specific topic. Consumers read messages. In other publish/subscribe systems, these clients may be called subscribers or readers. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. As shown in Figure 5.4 consumers are creating as per topics. This is different command line and setup process for creating different topics and consumer for each source.

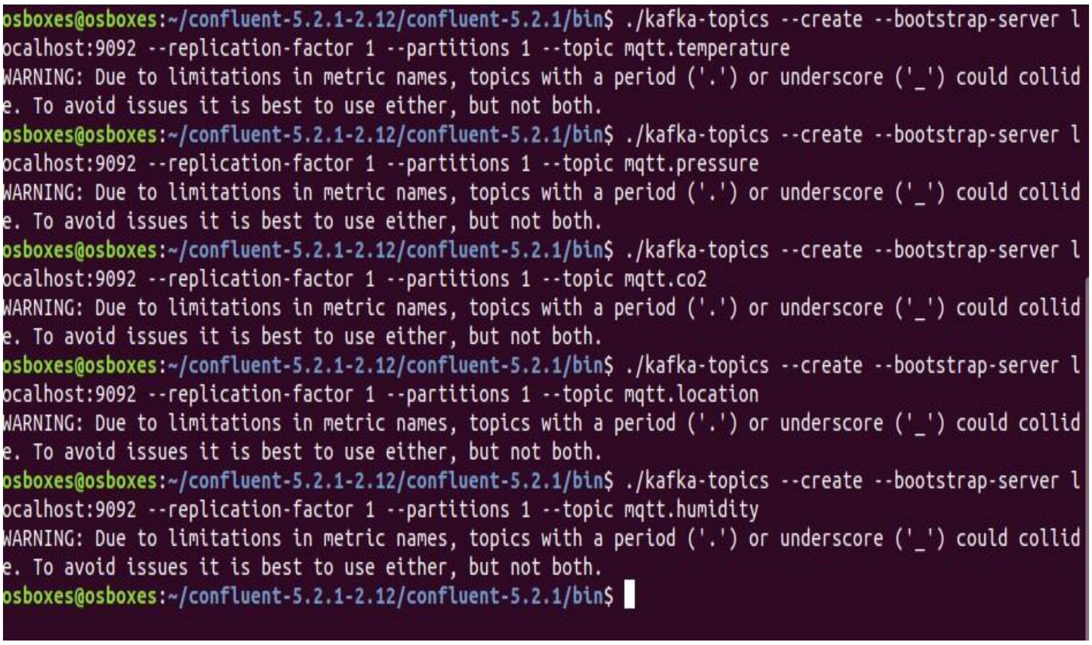

Figure 5. 10: Creating five different sensor value topics

With the help of Confluent Control Box latency can be measured. As shown in Figure 5.5, there is message flow graph with respect to latency graph.

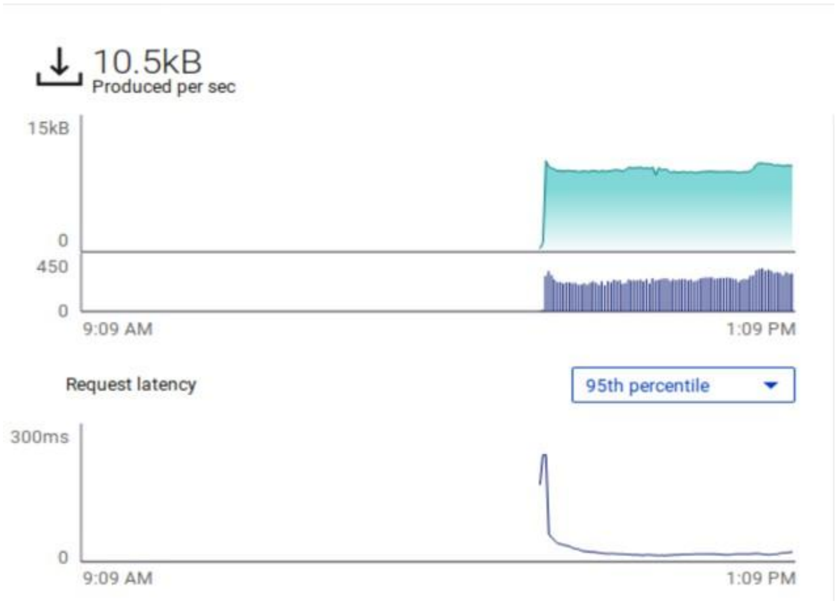

Figure 5. 11: Measuring latency using confluent control hub

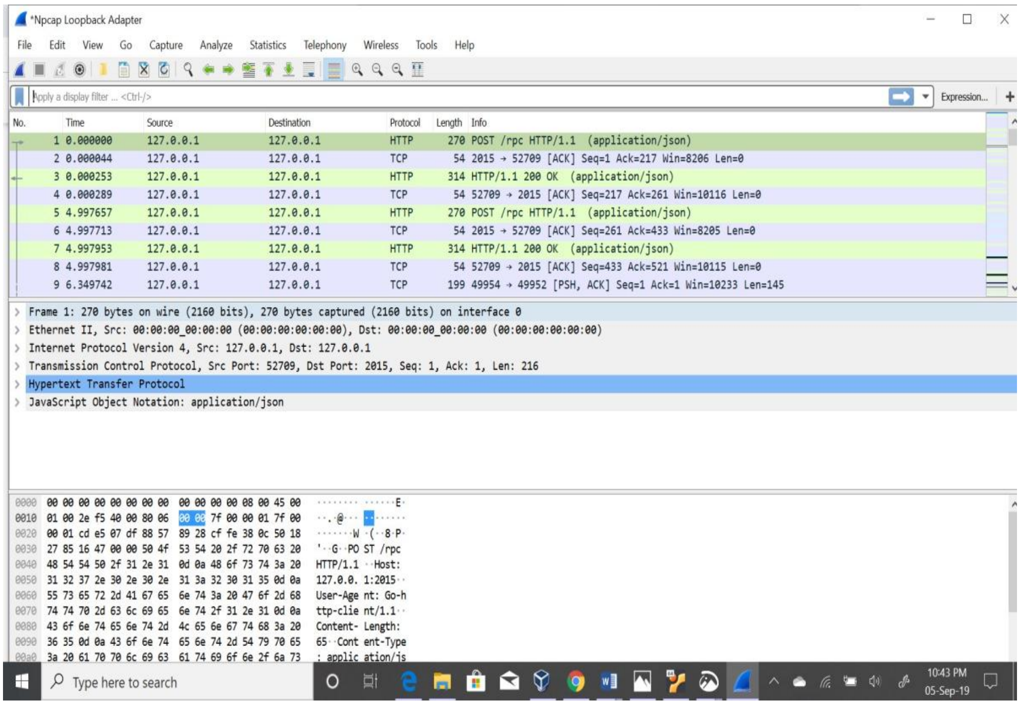

Figure 5. 12: Wire-shark capture in network

CHAPTER 6: Results, Analysis and Comparison

The deployment of Kafka connect- Source and Sink Connector results in latency of data to reach from source to destination. It is due to proper partition of task for data to travel from source to destination. The latency and throughput measurement of the path is done in traditional method i.e. using MQTT broker and new approach method i.e. using MQTT broker with Kafka Connect. Along with that, test is done traditional method and new approach method to find latency improvement in our custom network.

6.1 Latency Test Analysis

Latency is the amount of time it takes for the data that enters the channel or links at one end to exit at the other. If the link is short and not so congested, then the packets exits the bottom of the link almost as quickly. All latency measurements necessarily include the network latency between the application and the messaging system. Assuming all tests are performed in the same network configuration and that network provides consistent latency, then the network latency is a constant that affects all tests equally. When comparing latency measurements, then, it is important the network is held constant when making comparisons. Publishing latency is the amount of time that passes from when the message is sent until the time an acknowledgment is received from the messaging system. The acknowledgment indicates that the messaging system has persisted the message and will guarantee its delivery. Fetch latency is simply the time from when the message is sent by the producer to when it received by the consumer.

Table 6.1: Latency Measurement using one topic with one partition and four partition

1 topic 1 partition

Data Size (Bytes)	Produce latency(ms) 1 partition	Fetch latency(ms) 1 partition	CPU Usage (%)
100	9.876	11.437	13
500	12.065	13.765	13
1000	20.453	24.765	15
2000	27.987	29.876	18
3000	32.765	37.659	19
4000	42.982	50.468	21
5000	58.508	63.769	23

1 topic 4 partition

Data Size (Bytes)	Produce latency(ms) 4 partition	Fetch latency(ms) 4 partition	CPU Usage (%)
100	5.672	9.579	22
500	7.897	12.568	23
1000	9.614	15.472	23
2000	13.472	15.467	26
3000	15.285	17.546	27
4000	17.567	26.428	27
5000	23.784	27.829	30

In our first observation, we create one topic with one partition and one topic with four partitions. From observed data we collected Produce latency and fetch latency on different data size as shown table 6.1. In the case of one topic and one partition , latency increase as increase the byte size of data and same data is pass to 4 partition system result see that our Average produce latency decrease from 40.927ms to 18.6562ms and Average fetch latency decrease from 46.3478ms to 17.8412ms.CPU Usage percentage is also calculate from same data size and result see that CPU Usage percentage increase from 17.42% to 25.475% by one partition to 4 partition.

Table 6.2 Latency Measurement using one topic with one partition and four partition

2 topic 1 partition

Data Size (Bytes)	Produce latency(ms) 1 partition	Fetch latency(ms) 1 partition	CPU Usage (%)
100	22.465	27.834	17
500	36.246	44.145	17
1000	48.486	57.328	18
2000	52.691	64.241	20
3000	68.432	72.593	20
4000	76.842	86.836	22
5000	97.936	108.492	24

2 topic 4 partition

Data Size (Bytes)	Produce latency(ms) 4 partition	Fetch latency(ms) 4 partition	CPU Usage (%)
100	14.348	17.582	27
500	16.047	18.567	31
1000	21.381	24.278	32
2000	24.192	26.267	32
3000	33.873	36.532	35
4000	39.692	47.509	37
5000	42.863	47.824	37

Now, creating three topics with one partition and three topics with four partitions. From observed data we collected Produce latency and fetch latency on different data size as shown table 6.2. In the case of three topics and one partition, latency increase as increase the byte size of data and same data is pass to four partitions system result see that our Average produce latency decrease from 74.758ms to 27.485ms and Average fetch latency decrease from 84.495ms to 31.227ms. CPU Usage percentage is also calculated from same data size and result see that CPU Usage percentage increase from 19.71% to 31.227% by one partition to 4 partition

Table 6.3: Latency Measurement using three topics with one partition and four partition

3 topics 1 partitions

Data Size (Bytes)	Produce latency(ms) 1 partition	Fetch latency(ms) 1 partition	CPU Usage (%)
100	38.278	48.623	22
500	54.380	67.490	22
1000	72.891	89.201	22
2000	92.721	104.568	23
3000	106.724	127.854	23
4000	127.863	138.391	25
5000	143.292	157.306	25

3 topics 4 partition

Data Size (Bytes)	Produce latency(ms) 4 partition	Fetch latency(ms) 4 partition	CPU Usage (%)
100	24.367	28.782	34
500	28.092	32.901	35
1000	39.362	43.091	37
2000	44.098	46.679	37
3000	59.587	63.973	40
4000	64.083	69.145	40
5000	74.087	76.109	41

Now, creating three topics with one partition and three topics with four partitions. From observed data we collected Produce latency and fetch latency on different data size as shown table 6.3. In the case of three topics and one partition, latency increase as increase the byte size of data and same data is pass to four partitions system result see that our Average produce latency decrease from 90.878ms to 47.668ms and Average fetch latency decrease from 104.77ms to 51.485ms. CPU Usage percentage is also calculated from same data size and result see that CPU Usage percentage increase from 23.143% to 37.714% by one partition to 4 partition.

Table 6.4: Latency Measurement using four topics with one partition and four partition

4 topic 1 partition

Data Size (Bytes)	Produce latency(ms) 1 partition	Fetch latency(ms) 1 partition	CPU Usage (%)
100	49.103	55.753	28
500	72.431	88.234	28
1000	84.987	96.436	30
2000	124.039	139.076	30
3000	131.953	148.096	31
4000	143.048	155.612	32
5000	165.402	176.087	34

4 topics 4 partition

Data Size (Bytes)	Produce latency(ms) 4 partition	Fetch latency(ms) 4 partition	CPU Usage (%)
100	34.051	37.091	44
500	56.015	58.820	46
1000	74.098	79.098	47
2000	94.152	102.081	49
3000	128.140	130.201	51
4000	135.088	137.091	52
5000	152.146	163.097	54

Now, creating four topics with one partition and four topics with four partitions. From observed data we collected Produce latency and fetch latency on different data size as shown table 6.4. In the case of four topics and one partition, latency increase as increase the byte size of data and same data is pass to four partitions system result see that our Average produce latency decrease from 110.137ms to 96.241ms and Average fetch latency decrease from 122.75ms to 17.8412ms. CPU Usage percentage is also calculated from same data size and result see that CPU Usage percentage increase from 30.42% to 49.76% by one partition to 4 partition.

Table 6.5: Latency Measurement using five topics with one partition and four partition

5 topic 1 partition

Data Size (Bytes)	Produce latency(ms) 1 partition	Fetch latency(ms) 1 partition	CPU Usage (%)
100	51.103	57.753	32
500	74.871	89.002	32
1000	89.321	101.777	32
2000	131.939	142.996	35
3000	139.953	160.912	35
4000	151.848	168.692	39
5000	167.402	185.986	39

5 topics 4 partition

Data Size (Bytes)	Produce latency(ms) 4 partition	Fetch latency(ms) 4 partition	CPU Usage (%)
100	39.581	43.001	47
500	56.015	77.237	47
1000	79.876	90.430	50
2000	108.569	127.901	50
3000	120.90	147.812	52
4000	139.088	155.815	54
5000	159.619	172.185	55

Now, creating five topics with one partition and five topics with four partitions. From observed data we collected Produce latency and fetch latency on different data size as shown table 6.5. In the case of five topics and one partition, latency increase as increase the byte size of data and same data is pass to four partitions system result see that our Average produce latency decrease from 115.205ms to 90.567ms and Average fetch latency decrease from 135.96ms to 104.87ms. CPU Usage percentage is also calculated from same data size and result see that CPU Usage percentage increase from 34.85% to 50.714% by one partition to 4 partition.

Table 6.6: Throughputs Measurement using different topics with different partition

No. of topics	No. of partition	Throughputs (bps)
1	1	102500
1	4	407861
2	1	78600
2	4	325446
3	1	52968
3	4	247801
4	1	30813
4	4	150724

From the observation, throughput is inversely proportion to number of topics. As increase the number of topics throughput decrease. And throughput is directly proportion to number of partitions as increase the number of partition throughput increase shown in table 6.6.

To check the system performance same set of data are tested to traditional method i.e. using MQTT Broker from transfer data from source to destination. In this case, produce latency, fetch latency and CPU usage are calculated as show in table 6.5.

Table 6. 7: Latency Measurement using MQTT Broker using different topics

1 topic

Data Size (Bytes)	Produce latency(ms)	Fetch latency(ms)	CPU Usage (%)
100	4.876	7.437	5
500	5.065	8.765	7
1000	8.453	11.765	11
2000	9.987	12.876	13
3000	14.765	15.659	16
4000	16.982	17.468	19
5000	18.508	19.769	21

2 topics

Data Size (Bytes)	Produce latency(ms)	Fetch latency(ms)	CPU Usage (%)
100	10.876	11.437	15
500	25.065	32.065	17
1000	48.453	59.436	21
2000	77.897	89.765	22
3000	102.658	111.659	22
4000	122.876	128.752	25
5000	148.508	153.875	27

3 topics

Data Size (Bytes)	Produce latency(ms)	Fetch latency(ms)	CPU Usage (%)
100	23.045	36.254	23
500	50.185	70.655	23
1000	82.581	94.372	26
2000	107.987	115.626	27
3000	135.168	147.169	29
4000	156.058	160.924	32
5000	168.146	178.106	34

4 topics

Data Size (Bytes)	Produce latency(ms)	Fetch latency(ms)	CPU Usage (%)
100	39.561	46.437	33
500	72.186	83.140	36
1000	108.173	114.065	37
2000	138.745	140.182	37
3000	162.067	179.154	39
4000	182.137	194.712	41
5000	208.508	219.924	43

5 topics

Data Size (Bytes)	Produce latency(ms)	Fetch latency(ms)	CPU Usage (%)
100	12.631	13.182	37
500	14.369	15.268	36
1000	23.155	24.264	37
2000	35.782	39.577	37
3000	48.369	53.254	39
4000	62.283	67.764	41
5000	58.508	63.769	43

By integrating Kafka connect between MQTT and Kafka broker we maintain performance and reduce latency in server. As we can see from the table, by increasing of number topics i.e. data of different sensor, Latency is doesn’t increase because of distributed processing of Kafka Connect. From the Figure 6.1, from our custom Kafka connect configuration we can maintain the latency in design system even if number of source and message increase in system.

The Figure 6.1, Figure 6.2, Figure 6.3, Figure 6.4 and Figure 6.5 show the graph plot between Data Size and Latency measure in different scenario. The Figure 6.1 show the plot between Data Size and Produce/Fetch latency using 1 topic with 1 partition and 4 partition. The Figure 6.2 show the plot between Data Size and Produce/Fetch latency using 2 topics with 1 partition and 4 partition. The Figure 6.3 show the plot between Data Size and Produce/Fetch latency using 3 topics with 1 partition and 4 partition. The Figure 6.4 show the plot between Data Size and Produce/Fetch latency using 4 topics with 1 partition and 4 partition. The Figure 6.5 show the plot between Data Size and Produce/Fetch latency using 5 topics with 1 partition and 4 partition. From the analysis of graph plot the produce latency is always slightly lesser than fetch latency because the latency measured is only between the client and server in produce latency, where as in fetch latency server send the signal to client for acknowledgment after produce latency, so latency is slightly higher.

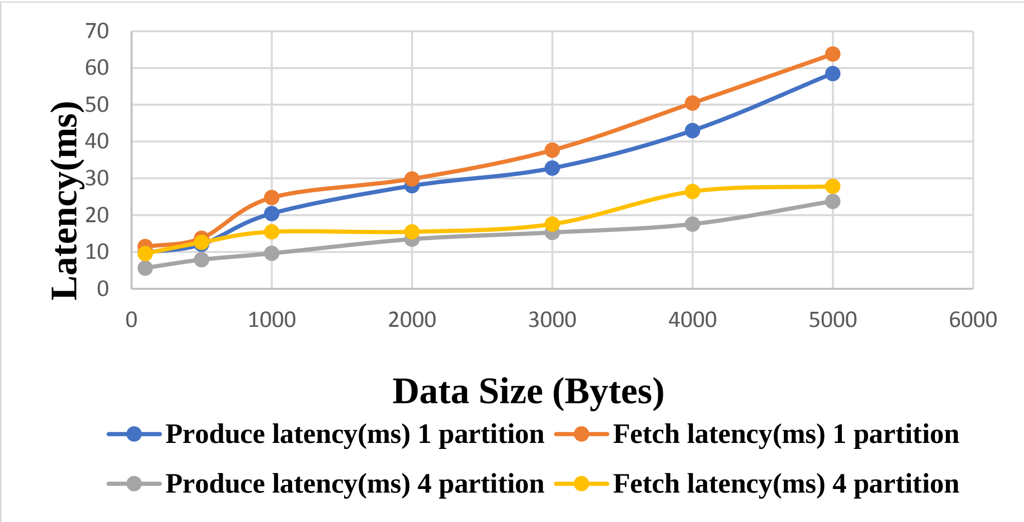

Figure 6. 11: Graph plot between Data Size and Produce/Fetch latency using 1 topic with 1 partition and 4 partition

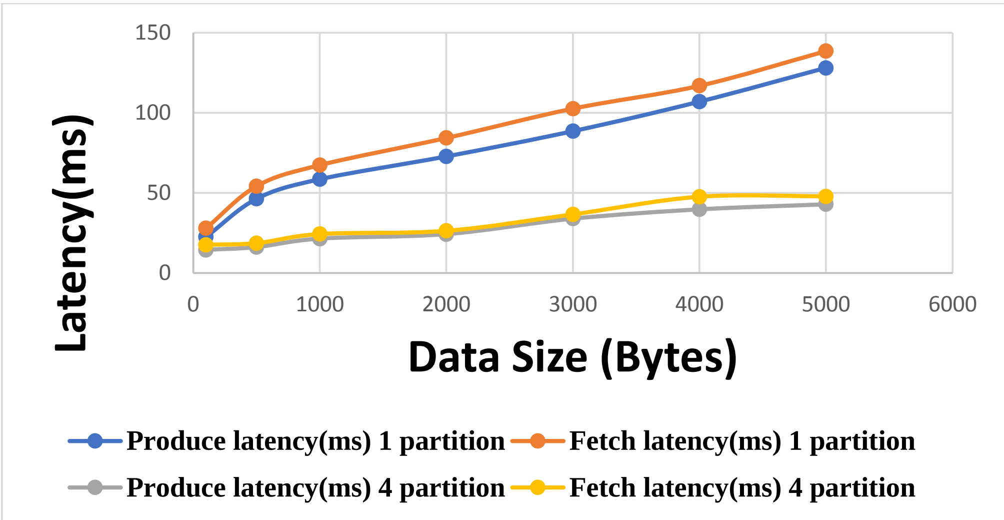

Figure 6. 12: Graph plot between Data Size and Produce/Fetch latency using 2 topics with 1 partition and 4 partition

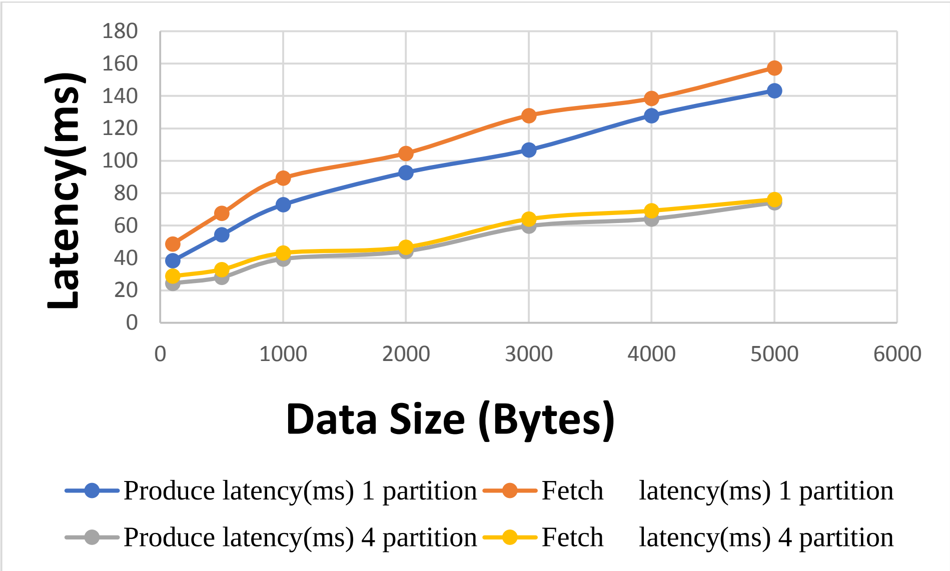

Figure 6. 13: Graph plot between Data Size and Produce/Fetch latency using 3 topics with 1 partition and 4 partition

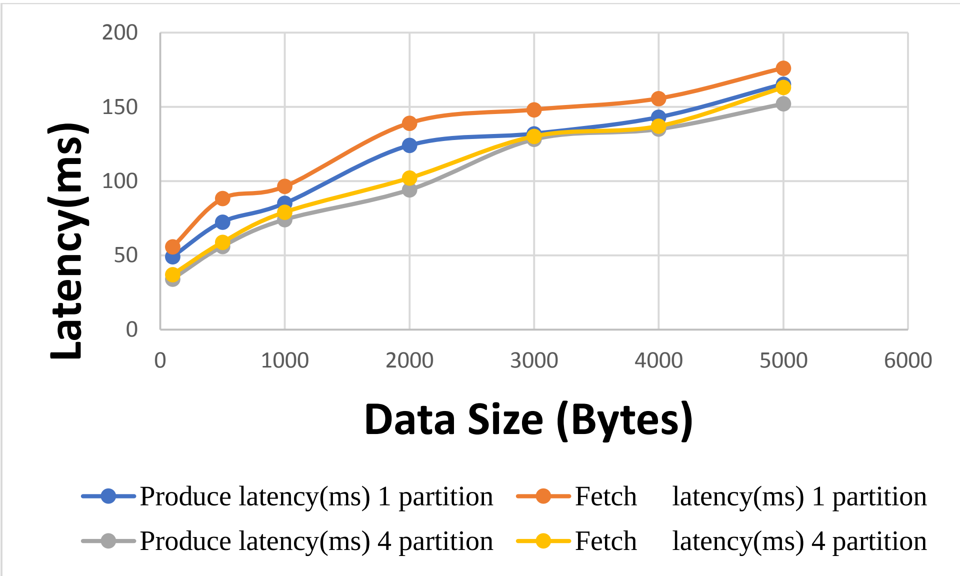

Figure 6. 14: Graph plot between Data Size and Produce/Fetch latency using 4 topics with 1 partition and 4 partition

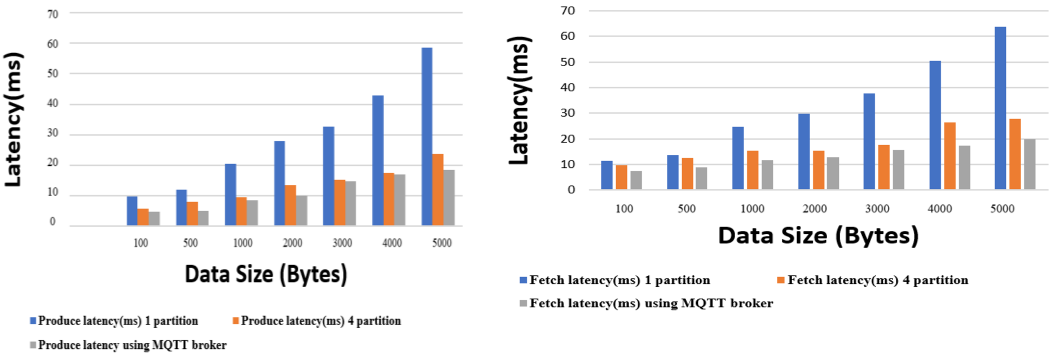

Figure 6. 15: Latency measurement for MQTT Broker and Kafka connected with one and four partition for one topic

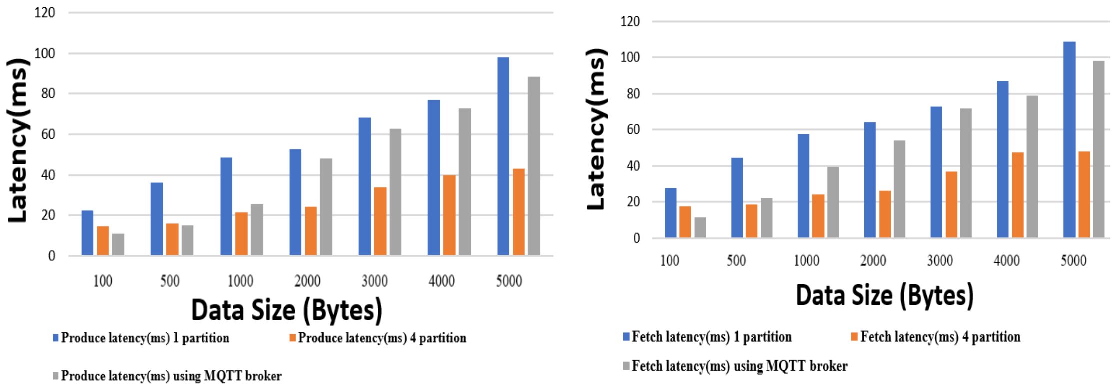

Figure 6. 16: Latency measurement for MQTT Broker and Kafka with one and four partition connected for two topics

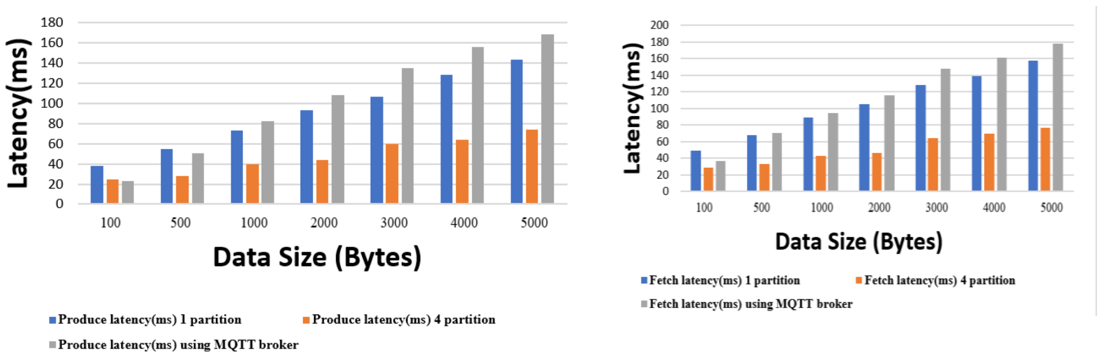

Figure 6. 17: Latency measurement for MQTT Broker and Kafka connected with one and four partition for three topics

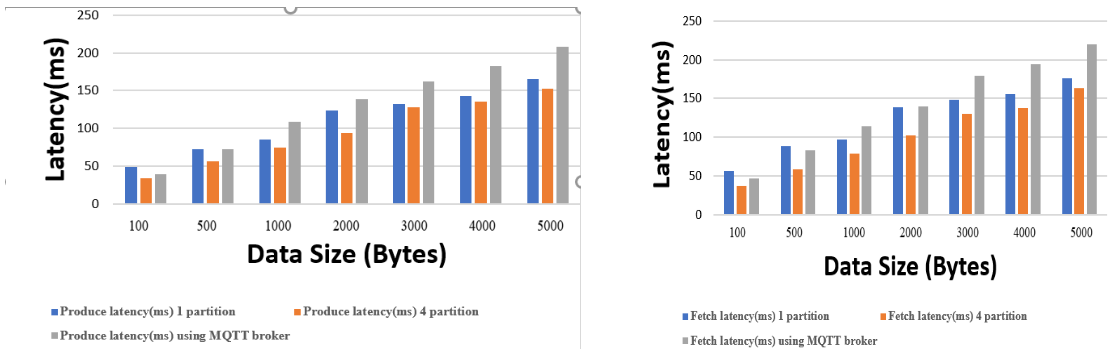

Figure 6. 18: Latency measurement for MQTT Broker and Kafka connected with one and four partition for four topics

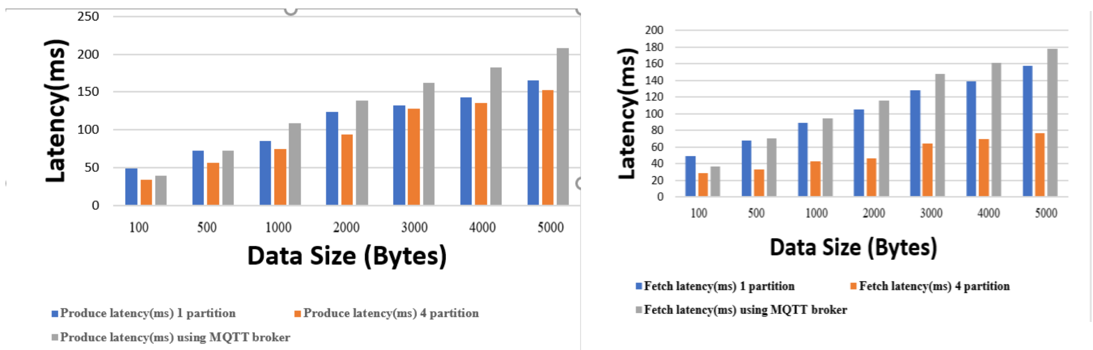

Figure 6. 19: Latency measurement for MQTT Broker and Kafka connected with one and four partition for five topics

CHAPTER 7: CONCLUSION AND LIMITATIONS

The MQTT Broker with Kafka Connect is successfully implemented. As we compared the traditional method i.e. MQTT Broker and new approach method i.e. MQTT Broker with Kafka connect we can conclude from output that from new approach method we can reduce the latency and increase the throughputs. We can scale our network of connected IOT devices according to our need with maintaining low latency and high throughputs. These results show that latency reduces as we increase the number of partitions, since partitions are a unit of parallelism. Across the board, as the number of partitions increases both the publish and the fetch latency decreases.

In future, the thesis work can be extended by investigating machine learning approach for topics and partitions management with our server. We can optimize our framework to use less CPU resources.

Limitation

Integrating MQTT with Kafka Connect for processing IOT data is carried out considering latency, throughputs and CPU consumption. This thesis does not consider the more parameters like response time of server and high availability of computing system.

References

1. B. Weiss, U. Hunkeler, A. Munari, W. Schott, and L. Truong, "A publish/subscribe messaging system for wireless sensor communication." 2016 IEEE Ninth International Conference on. IEEE, 2014.

2. C. Rodríguez-Domínguez, K. Benghazi, M. Noguera, J. L. Garrido, M. L.

3. Rodríguez, and T. Ruiz-López, "A communication model to integrate the request response and the publish-subscribe paradigms into ubiquitous systems,"

4. D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on. IEEE, 2014.

5. Omer Koksal,Bedir Tekinerdogam, "Feature-Driven Domain Analysis of Session Layer Protocols of Internet of Things" in 2017 IEEE International Congress on Internet of Things.

6. Rishika Shree, Tanupriya Choudhury, Subhash Chand Gupta, Praveen Kumar,

7. "KAFKA: The Modern Platform for Data Management and Analysis in Big Data Domain" in 2017 2nd International Conference on Telecommunication and Networks (TEL-NET 2017)

8. MatthiasJ.Sax, MatthiasWeidlich, Johann-ChristophFreytag, "Streams and Tables:

9. Two side of same coin". in 2017 IEEE International Congress on Internet of Things.

10. Anindya Dey, Matthew Tolentino, "Characterizing the Impact of Topology on IoT Stream Processing"

11. "Kafka The Definitive Guide REAL-TIME DATA AND STREAM PROCESSING AT SCALE" by Neha Narkhede , Gwen Shapira & Todd Palino.

12. Godson Michael D'silva, Siddhesh Bari, "Real-time Processing of IoT Events with Historic data using Apache Kafka and Apache Spark with Dashing framework" in2017 2nd IEEE International Conference On Recent Trends in Electronics Information & Communication Technology (RTEICT), May 19-20, 2017, India

13. Priyanka Thota, Yoohwan Kim, "Implementation and Comparison of M2M Protocols for Internet of Things" in 2017 4th Intl Conf on Applied Computing and Information Technology

14. NICOLAS NANNONI, "Message-oriented Middleware for Scalable Data

15. Analytics Architectures"

16. Yeva Byzek, "Optimizing Your Apache Kafka Deployment" , 2019 Confluent, Inc.

17. Internet Engineering Task Force (IETF), Mar 2017 [Online]. Available:

18. www.ietf.org.

19. https://mosquitto.org[Accessed on 1/06/2019]

20. https://kafka.apache.org[Accessed on 15/07/2019]

21. https://www.wireshark.org/ [Accessed on 22/08/2019]

22. https://www.java.com/ [Accessed on 27/08/2019]

23. https://www.scala-lang.org[Accessed on 28/08/2019]