Comprehensive Review of Stream Processing Tools

Premal Singh[1], Prof. Chaitra B H[2]

[1]*Department of Computer Science and Engineering, R.V. College of Engineering, Bengaluru, Karnataka, India*
[2]*Department of Computer Science and Engineering, R.V. College of Engineering, Bengaluru, Karnataka, India*

---***---

Abstract - *Stream Processing is a technology widely used in fetching continuous streams of data within a very short period of time. In the internet era, data is not limited to a small set of producers and consumers. There are a large number of producers and consumers. In the earlier methods, connections would be created between each producer and consumer mapped to the data being transferred and hence it became difficult to manage large amounts of data. Stream processing makes it easy to handle a large number of transactions by bringing the concept of message broker, which reduces the number of connections. Apache Kafka is a fault tolerant, scalable and extremely fast open-source stream processing software. RabbitMQ is a lightweight message broker that supports various different protocols. Apache ActiveMQ is a message broker, which supports flexible and powerful open source multiprotocol messaging. All these stream processing tools have their own advantages and disadvantages. This paper performs and in-depth comparative study of these stream processing tools, which are widely used to handle large amounts of data. Such a study will be helpful in identifying the suitable screen processing tool for the required application.*

Key Words: Stream Processing, Message Broker, Producer, Consumer

1. INTRODUCTION

Software industry enterprises often propose a requirement to make data communication within its services to be reliable, fault tolerant and extremely fast. With this reason, a lot of enterprises have started adopting stream processing software to provide solutions, such as (i) simple message passing through inter-service communication in microservice architecture, and (ii) whole stream processing platform applications. One of the examples of such a software application that is frequently used to book cabs requires a lot of real-time processing. These applications handle roughly around more than a trillion messages per day and results in data volume of Petabytes. Other similar applications, that widely use stream processing, include chat applications, social media applications and media streaming platforms. One of the reasons, why these tools are in much demand for these applications, is that this tool is based on the Publisher/Subscriber Messaging model. Publisher/Subscriber Messaging model is a messaging model, which performs asynchronous service-to-service communication. It is primarily used in serverless and microservices architectures. In this architecture, messages are exchanged without knowing the sender or the recipient. Publishers refer to a set of senders who would be sending messages to a topic. Topic refers to an intermediary channel that receives a message from a publisher, and sends it to the subscriber.

There are a wide range of applications, which implement a publisher/subscriber model. Bhawiyuga *et. al*. [1] developed a Publisher/Subscriber based software that would avail real time web access on constrained devices. This system includes sensor-actuator equipped devices, a web-based client and a Message Queueing Telemetry Transport (MQTT) broker. MQTT is a Publisher/Subscriber messaging protocol designed for constrained physical devices. With this, use of the Publisher/Subscriber model helps in fast, fault tolerant and reliable messaging between components of this system. Yong-Kyung Oh *et. al*. [2] discusses how message brokering services that are based on publish and subscribe paradigm are getting adopted in many areas such as remote facilities monitoring. In this paper, they have introduced a cloud-based message brokering service, which allows better scalability and dynamic load balancing between brokers. Kawazoe *et. al.* [3] present a testing framework for a message broker system for consumer devices based on Publisher/Subscriber Paradigm. There are various challenges that a tester needs to face while testing such an application. This paper describes how testing of such an application becomes easy with their framework. Hiraman *et. al.* [4] present a study of Kafka in processing large amounts of data stream. *Apache Kafka* is a very popular architecture used for processing stream data. Kato *et. al*. [5] discuss how heavy processing loads are one of the major issues for deep learning, and hence develop a prototype system of the proposed distributed stream processing infrastructure using *Ray*, a distributed execution framework and *Apache Kafka*, and demonstrate its performance. .Hong *et. al*. [6] presented a performance analysis of RabbitMQ and REST API respectively as the middleware responsible for messaging between the services of microservice web applications. Publisher/Subscriber paradigm for the large scale IoT systems face issues such as data confidentiality and service privacy. Duan *et. al.* [7] propose a security framework to bridge this gap.

In this paper, a comparative review of different stream processing tools has been performed. It has been particularly analyzed that how different applications widely used in domains such as IoT, Cloud Computing utilize the stream processing technology, to ensure that service to service communication becomes fast, fault-tolerant and scalable. The paper is organized as follows. Section II discusses about the concept of publisher/subscriber messaging model. Next, in Section III, the concepts and basic working of Apache Kafka, RabbitMQ and Apache ActiveMQ have been discussed. This is followed by presentation of a comparison between these tools in Section IV. Finally, Section V discusses some of the conclusions.

2.0 Publisher/Subscriber Messaging Model

In applications based on Distributed Systems, the components of the system often need to communicate with other components whenever any event takes place. In the Client/Server messaging system, clients send a request to the server, and then wait for its response [8]. An effective way to decouple senders from receivers is to use asynchronous messaging. This is implemented using a message queue, where the sender puts their message in the queue and the receiver fetches the message from the queue. With this, the need of blocking the sender to wait for the response can be avoided. Use of message queues however does not scale well for a large number of recipients. Hence, the problem that arises with conventional message queues is that when the number of consumers is large, it makes the entire system effectively slower.

The solution to this scenario is a Publisher subscriber messaging model. Publisher refers to a sender, who sends the message. Subscriber refers to a recipient, receiving that message. Only difference is that the publisher publishes the message to a topic. A topic is a category of messages. Subscriber receives the message from the topic., The publisher therefore does not need to keep a track of the subscribers that need to subscribe to the message. The publisher only needs to send the message to the topic. Subscriber needs to subscribe to the topic [9]. For example, if there are x senders, and y receivers, and each sender wants to send a message to each receiver, there would be a requirement of xy connections. Figure 1 depicts this scenario.

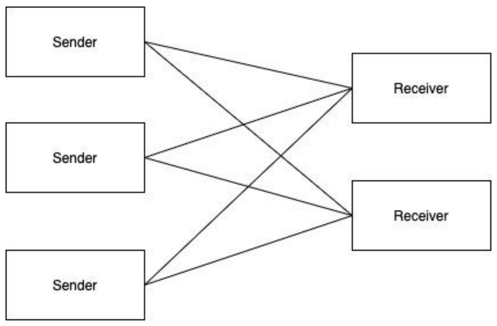

Fig. 1: All Senders communicating with all receivers

However, with a publisher/subscriber messaging model, the requirement will be reduced x+y messages, since there will be a broker to which all publishers and subscribers will be connected as depicted in Fig. 2.

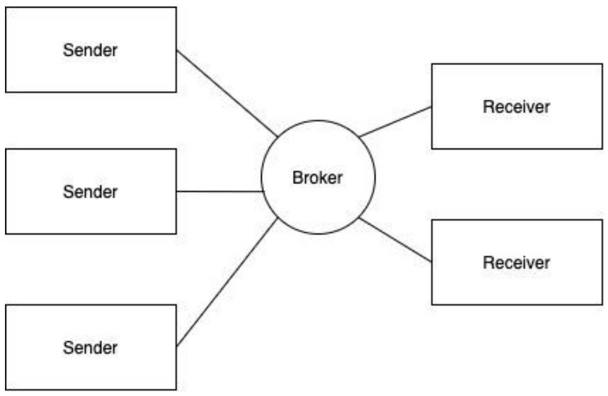

Fig. 2: All Senders communicating with all receivers through message broker

Publisher/Subscriber Messaging Model offers various benefits such as improving the scalability and reliability.

3.0 Open Source Stream Processing Software

This section discusses some of the most commonly used open source stream processing software for industrial applications such as Apache Kafka, RabbitMQ and ActiveMQ.

A. Apache Kafka

Apache Kafka is a stream processing software developed in Scala and Java. Kafka runs as a cluster on multiple servers. This Kafka cluster stores a stream of records in categories called topics. Each record is composed of a timestamp, key and a value. There are four major APIs provided by Kafka, which are Producer API, Consumer API, Connector API and Streams API.

Producer API allows a service can publish a stream of messages to Kafka topic(s) . Consumer API enable a service subscribe to a group of topics and process the message stream produced to them. The Streams API makes the service act as a stream processor, that is consuming an input stream from topic(s) and producing an output stream to output topic(s). Connector API allows executing reusable producers/consumers that connect Kafka topics to existing applications or data systems.

A topic is a common name to which specific messages are published. For example, if messages are needed to be categorized into classes, topics can be used to represent these classes. Consumers subscribe to the topic and receive messages from the specific category, which the topic maps to. Topics in Kafka can have any number of consumers. Each Kafka topic is distributed into partitions. With the help of partitions, topics can be parallelized by dividing the data in a particular topic across multiple brokers. Each message within a partition is identified using offset.

Each broker holds a number of partitions and each of these partitions can either be a leader or a replica for a topic.

Whenever any read/write is done, it goes through the leader. Leader is responsible for updating the other replicas. In case any leader fails, a replica takes over the leader.

Producer makes a write to a leader partition and that leader partition makes sure replica partitions get updated. This makes sure the system is fault tolerant.

Consumers read from any single partition, and therefore the throughput of message consumption gets scaled, as it is in message production. Consumers are also grouped into a group of consumer groups for a given topic.

Apache Kafka can be used either as a storage, messaging or a stream processing system.

B. RabbitMQ

RabbitMQ is a message broker software that is also widely used by industries for stream processing. Advantages of using RabbitMQ are that it is a lightweight message broker. It can be easily deployed on premises, and in the cloud as well. Another advantage of using RabbitMQ is that a lot of messaging protocols are supported.

Producer and Consumer refer to the sender and receiver in RabbitMQ. Producer publishes a message to the exchange. The exchange receives the message. The Exchange has the responsibility for routing that message to the receiver. RabbitMQ takes different message attributes such as routing keys. Routing key is the key for the exchange, which tells how the message needs to be routed. Concept of routing key is similar to that of the topic in Apache Kafka. The messages remain in the queue. The consumer handles the message.

C. ActiveMQ

ActiveMQ is a message-oriented middleware developed by Apache. ActiveMQ is based on Java. It makes use of Java Message Service (JMS) API.

ActiveMQ sends messages between client applications - producers and consumers. Producers generate messages and Consumers receive and process these messages. The ActiveMQ operates and routes the messages either through the destinations as a queue or as a topic. If there is a single consumer, the destination would be a queue. If there are multiple consumers, the message can be routed with the help of a topic which is based on the Publisher/ Subscriber model.

JMS is the communication standard used by ActiveMQ. ActiveMQ sends messages asynchronously. Each ActiveMQ message is based on the JMS specification. Therefore, it is made up of headers, optional properties and a body.

High throughput is one of the benefits that ActiveMQ provides. Since producers do not need to wait for any acknowledgements, they do not get blocked while sending messages. ActiveMQ is very flexible. Clients might not be available temporarily. They can be dynamically added to the environment. ActiveMQ clients can operate independently. They do not need to directly interact. They can interact with the ActiveMQ broker.

4.0 Comparison of Kafka, RabbitMQ and ActiveMQ

Apache Kafka is better than RabbitMQ in terms of performance. Kafka uses sequential disk I/O to enhance its performance. Therefore, this makes it a better option for implementing queues. RabbitMQ, on the other hand, requires more resources. Kafka is based on a pull model. Kafka Consumers request a batch of messages from any offset. RabbitMQ is based on the push model and hence, it stops the consumers from overwhelming. Kafka provides ordering of messages, whereas RabbitMQ does not support message ordering. In Kafka, messages can remain there forever, unless any message retention policy is specified. RabbitMQ on the other hand is a queue, so once a message is consumed, it does not remain in the queue anymore. Kafka provides delivery guarantees but RabbitMQ does not guarantee atomicity. Kafka does not prioritize messages, however RabbitMQ allows specifying message priorities.

ActiveMQ is used in enterprise projects. It is used to store multiple instances. RabbitMQ, on the other hand, is a message broker and runs in a low-level AMQP protocol. ActiveMQ can be implemented with two brokers. RabbitMQ requires only one broker. Message Patterns available in ActiveMQ are publish/subscribe and message queue. RabbitMQ supports Message Queue, publish/subscribe, RPC and routing.

Kafka producer does not wait for acknowledgements from the broker, whereas ActiveMQ maintains the delivery state of every message sent. Kafka has better storage efficiency when compared to ActiveMQ. ActiveMQ uses more space than Kafka. Kafka is a pullbased messaging system, whereas ActiveMQ is a push-based messaging system. As a result of this, throughput in Kafka is higher than that of ActiveMQ. [10]

5. CONCLUSIONS

This paper presents a general study and comparison on open source stream processing softwares used by various software applications. Based on the comparisons done on Kafka, RabbitMQ and ActiveMQ, it can be concluded that each software can deliver an effective messaging system depending specifically on the use case. Stream processing is implemented in various IoT and cloud-based applications to provide fault tolerant, highly scalable low latency solutions.

REFERENCES

[1] A. Bhawiyuga, D. P. Kartikasari and E. S. Pramukantoro, "A publish subscribe based middleware for enabling real time web access on constrained device," 2017 9th

International Conference on Information Technology and Electrical Engineering (ICITEE), Phuket, 2017, pp. 1-5

[2] F. Y. Oh, S. Kim, H. Eom, H. Y. Yeom, J. Park and Y. Lee, "A scalable and adaptive cloud-based message brokering service," 13th International Conference on Advanced Communication Technology (ICACT2011), Seoul, 2011, pp. 498-501.

[3] H. Kawazoe, D. Ajitomi and K. Minami, "A test framework for large-scale message broker system for consumer devices," 2015 IEEE 5th International Conference on Consumer Electronics - Berlin (ICCE-Berlin), Berlin, 2015, pp. 24-28.

[4] B. R. Hiraman, C. Viresh M. and K. Abhijeet C., "A Study of Apache Kafka in Big Data Stream Processing," 2018 International Conference on Information , Communication, Engineering and Technology (ICICET), Pune, 2018, pp. 1-3.

[5] K. Kato, A. Takefusa, H. Nakada and M. Oguchi, "A Study of a Scalable Distributed Stream Processing Infrastructure Using Ray and Apache Kafka," 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 2018, pp. 5351-5353.

[6] X. J. Hong, H. Sik Yang and Y. H. Kim, "Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application," 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, 2018, pp. 257-259.

[7] L. Duan, C. Sun, Y. Zhang, W. Ni and J. Chen, "A Comprehensive Security Framework for Publish/Subscribe-Based IoT Services Communication," in IEEE Access, vol. 7, pp. 25989-26001, 2019.

[8] D. Serain, "Client/server: Why? What? How?," International Seminar on Client/Server Computing. Seminar Proceedings (Digest No. 1995/184), La Hulpe, Belgium, 1995, pp. 1/1-111 vol.1.

[9] Xiwei Feng, Chuanying Jia and Jiaxuan Yang, "Semantic web-based publish-subscribe system," 2008 IEEE International Conference on Service Operations and Logistics, and Informatics, Beijing, 2008, pp. 1093-1096.

[10] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), Craiova, 2015, pp. 132-137.