Build a Fault Tolerant Distributed Application using Apache Kafka and Zookeeper

Surbhi Sharma
Masters of Engineering, Department of CSIS
Birla Institute of Technology, Pilani (Hyderabad Campus)
Student ID – 2020H1030148H

Himanshu Dhyani
Masters of Engineering, Department of CSIS
Birla Institute of Technology, Pilani (Hyderabad Campus)
Student ID – 2020H1030119H

Under the supervision of

Prof. Dr. Manik Gupta, Department of Computer Science

Birla Institute of Technology & Science, Hyderabad

Abstract

Apache Kafka is the implementation of software bus for stream of data. It is an open source distributed event streaming platform. Using Event streaming feature of Kafka and Zookeeper, we have tried to handle the twitter data. Using the Twitter API, all the tweets having a particular hashtag is being fetched and this live stream is then processed. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. A Producer consumer model is implemented where Producer producer the tweets in sequence and consumer(s) consume it concurrently. Two properties, namely, Fault tolerance and Concurrency have been implemented and verified.

INTRODUCTION

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. Kafka is written in Scala and Java. It is fast, scalable and distributed by design. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at right place at the right time.

Event streaming is applied to a wide variety of use cases across a plethora of industries and organizations. Its many examples include:

1. To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.

2. To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.

3. To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.

4. To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.

5. To monitor patients in hospital care and predict changes in condition to ensure timely treatment in emergencies.

6. To connect, store, and make available data produced by different divisions of a company.

7. To serve as the foundation for data platforms, event-driven architectures, and microservices.

There are several use cases of Apache Kafka which includes messaging, website activity tracking, metrics, log aggregation, stream processing, event sourcing and commit log.

BACKGROUND

Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments. Kafka combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution:

1.. To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems.

2. To store streams of events durably and reliably for as long as you want.

3. To process streams of events as they occur or retrospectively.

All this functionality is provided in a distributed, highly scalable, elastic, fault-tolerant, and secure manner. We can choose between self-managing your Kafka environments and using fully managed services offered by a variety of vendors.

Servers: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

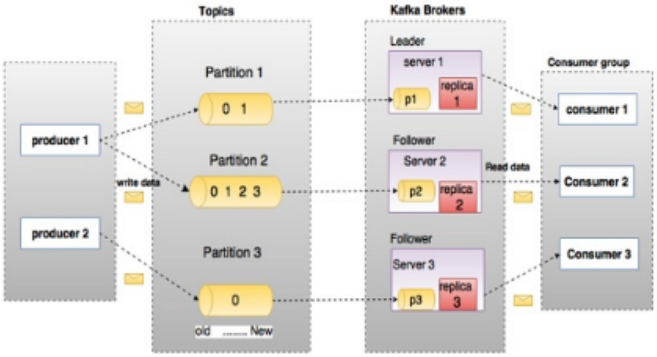
Clients: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala including the higher-level Kafka Streams library, for Go, Python, C/C++, and many other programming languages as well as REST APIs.

Producers are those client applications that publish (write) events to Kafka, and **Consumers** are those that subscribe to (read and process) these events. Since producers and consumers are independent of each other, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

Events are organized and durably stored in **Topics**. A topic is similar to a folder in a filesystem, and the events are the files in that folder. Topics in Kafka are always multi-producer and multi-subscriber. Events in a topic can be read as often as needed. Unlike traditional messaging systems, events are not deleted after consumption, instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Topics are partitioned, that is, spread over a number of "buckets" located on different **Kafka brokers**. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

To make your data fault-tolerant and highly-available, every topic can be **replicated**, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common production setting is a **replication factor** of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic-partitions.



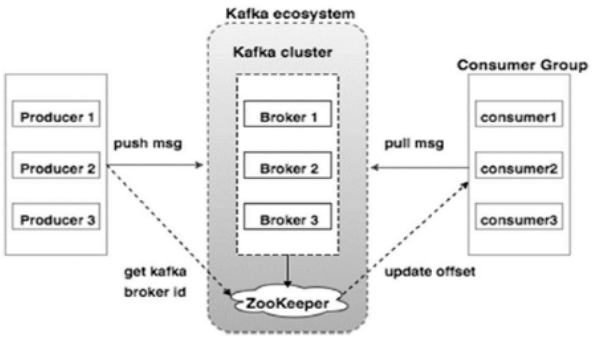
Kafka brokers use Zookeeper to manage and cordinate the Kafka cluster. Zookeeper notifies all nodes when the topology of the Kafka cluster changes, including when brokers and topics are added or removed. ZooKeeper also enables leadership elections among brokers and topic partition pairs, helping determine which broker will be the leader for a particular partition (and server read and write operations from producers and consumers), and which brokers hold replicas of that same data. When ZooKeeper notifies the cluster of broker changes, they immediately begin to coordinate with each other and elect any new partition leaders that are required. This protects against the event that a broker is suddenly absent.

Broker: Kafka cluster typically consists of multiple brokers to maintain load balance.

ZooKeeper: ZooKeeper is used for managing and coordinating Kafka brokers. Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.

Producers: Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker.

Consumers: Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset.



APPROACH

Our work was divided into broad steps comprising of setting up Apache Kafka and Zookeeper Framework. To work on Producer and Consumer, we have to implement Publish-Subscribe messaging model. The workflow of Publish-Subscribe messaging model includes:

1. Producers send messages to a topic at regular intervals.

2. Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.

3. Consumer subscribes to a specific topic.

4. Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.

5. Consumers will request the Kafka in a regular interval (like 100 ms) for new messages.

6. Once Kafka receives the messages from producers, it forwards these messages to the consumers.

7. Consumers will receive the message and process it.

8. Once the messages are processed, consumers will send an acknowledgement to the Kafka broker.

9. Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read the next message correctly even during server outrages.

10. This above flow will repeat until the consumer stops the request.

11. Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

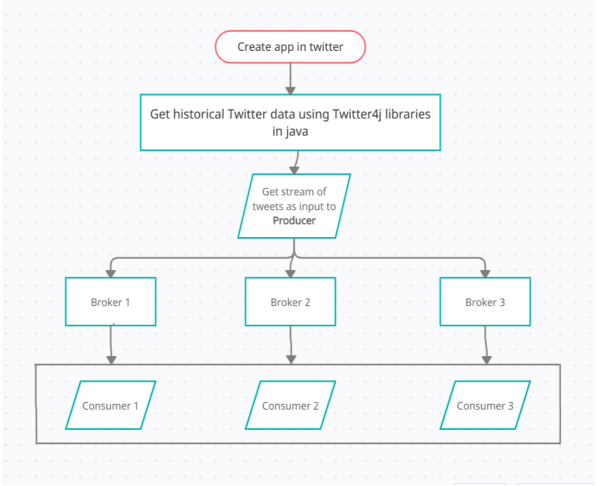
Following the above steps, the Producer and Consumer will write and read the data from Kafka broker. Since we had to work with stream of tweets that was fetched from twitter in real time, we have scheduled producer to produce or write the tweets in sequence into Kafka broker, and multiple consumers will consume or read these tweets concurrently from the Kafka broker.

IMPLEMENTATION

For the implementation of the model, we first installed Java (version 8), ZooKeeper Framework, Apache Kafka, IntelliJ IDEA ide. Our implementation is divided into 5 steps:

1. Create app in twitter.
2. Java code to fetch data related to the hashtag entered using Twitter4j libraries .
3. Create producer java code in ide.
4. Create 3 broker in console
5. Create 3 consumer in console

Below flow diagram shows the working flow of the project.



Create app in Twitter: Twitter is a popular social network where users share messages called tweets. Twitter allows us to mine the data of any user using Twitter API. The data will be tweets extracted from the user. The first thing to do is get the consumer key, consumer secret key, access key and access secret key from twitter developer available easily for each user. These keys will help the API for authentication.

Twitter4J: Twitter4J is an open source Java library, which provides a convenient API for accessing the Twitter API. It also ensures the security and privacy of a user – for which we naturally need to have OAuth credentials configured in our app. We need to start by defining the dependency for Twitter4J in our pom.xml

Java Producer: Basically, there are four steps to create a java producer

- Create producer properties
- Create the producer
- Create a producer record
- Send the data.

Multi-Broker: We have created 3 brokers which will work on 3 partitions with 3 replicas of each partition. Multi brokers provide fault tolerance.

Multi-Consumer: We have created 3 consumers in the console, consuming stream data concurrently.

DISADVANTAGES

1. No Complete Set of Monitoring Tools: It is seen that it lacks a full set of management and monitoring tools. Hence, enterprise support staff felt anxious or fearful about choosing Kafka and supporting it in the long run.
2. Issues with Message Tweaking: As we know, the broker uses certain system calls to deliver messages to the consumer. However, Kafka's performance reduces significantly if the message needs some tweaking. So, it can perform quite well if the message is unchanged because it uses the capabilities of the system.
3. Not support wildcard topic selection: There is an issue that Kafka only matches the exact topic name, that means it does not support wildcard topic selection. Because that makes it incapable of addressing certain use cases.
4. Lack of Pace: There can be a problem because of the lack of pace, while API's which are needed by other languages are maintained by different individuals and corporates.

RESULTS & CONCLUSION

We have successfully implemented two properties-Fault tolerance and Concurrency in kafka. We have verified fault tolerance properties by shutting down one of the brokers in the middle of the stream sending process to the broker by producer. We have also verified the concurrency property by showing the three consumers working in a group and consuming messages independently and parallely.

ACKNOWLEDGMENT

We are very thankful to Prof. Dr. Manik Gupta for her constant guidance and help throughout the project. It would have been difficult to accomplish this without your support. She has been constantly available to share references, to guide and assist us with the Distributed Systems concepts and helping us understand the terminologies, concepts and techniques

REFERENCES

1 *Kreps, J., Narkhede, N. and Rao, J., 2011, June. Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (Vol. 11, pp. 1-7).*

2 *Kleppmann, M. and Kreps, J., 2015. Kafka, samza and the unix philosophy of distributed data.*

3 *Wang, Z., Dai, W., Wang, F., Deng, H., Wei, S., Zhang, X. and Liang, B., 2015, November. Kafka and its using in high-throughput and reliable message distribution. In 2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS) (pp. 117-120). IEEE.*

4 *Thein, K.M.M., 2014. Apache kafka: Next generation distributed messaging system. International Journal of Scientific Engineering and Technology Research, 3(47), pp.9478-9483.*

5 *Le Noac'H, P., Costan, A. and Bougé, L., 2017, December. A performance evaluation of Apache Kafka in support of big data streaming applications. In 2017 IEEE International Conference on Big Data (Big Data) (pp. 4803-4806). IEEE.*