RESEARCH ARTICLE OPEN ACCESS

# A Study on Developing Resilient Big Data Stream Processing Applications using Apache Kafka

Shreya Sahay*, Sneha M**

*(Department of CSE, R V College of Engineering, Bengaluru
Email:shreyasahay.cs17@rvce.edu.in)

** (Assistant Professor, Department of CSE, R V College of Engineering, Bengaluru
Email :sneham@rvce.edu.in)

----------------------------------------************************----------------------------------

## Abstract:

Big data is a vital resource across all industries today to garner useful insights into user behaviour as well as to perform near real-time analytics on large volumes of incoming data. Critical applications dealing with such data require a resilient architecture which ensures minimal loss of data. This paper explores and analyses different approaches that append to the power of Apache Kafka in order to develop resilient services capable of automatic error recovery.

*Keywords* **—Big Data, Stream Processing, Apache Kafka, Dead Letter Queue.**

----------------------------------------************************----------------------------------

## I. INTRODUCTION

Data has today become the core ingredient in human lifestyle. The world economic forum declared that at the beginning of 2020, the number of bytes in the digital universe was 40 times bigger than the number of stars in the observable universe and that by 2025, the amount of data generated each day is expected to reach 463 exabytes globally [1]. Such large volume of data is generated at a rapid pace every second and therefore renders traditional data processing techniques obsolete. This necessitates emergence of architectures and methods to specifically deal with this stream of data for processing in such a manner that minimum data is lost.

As stated in one of the fundamental papers [2], compared to traditional data, the features of big data can be characterized by 5V, namely, huge Volume, high Velocity, high Variety, low Veracity, and high Value. Sources of big data can be traced to the physical world, which is usually obtained through sensors, scientific experiments and observations and to the digital world, where data is acquired through every click, movement, search, etc. Handling such data using traditional techniques encounters numerous roadblocks and this led to the development of the Big Data technologies and tools to efficiently handle big data.

Stream processing [3] is a technology under Big Data that focuses on real time processing of data streams. It is similar to batch processing systems [4] but with a subtle difference. While batch processing involves batches of data that have already been stored over a period of time, and is run on regularly scheduled times or on an as-needed basis, stream processing [5] enables the user to feed data into analytics tools as soon as they get generated, facilitating real-time data processing and instant analytics results.

Messaging system transfers data from one process to another. There are many traditional messaging systems [6,7,8] but most of these cannot handle real time big data. IBM WebSphere [9] is one such messaging system that allows producers to send messages to more than one topic. Similarly, Hedwig [10] is a shared-nothing message broker. These traditional messaging systems have a broker or hub which keeps track of the subscriptions and what has been consumed so far. Distributed messaging systems focus on reliable message queuing by using a publisher-subscriber model where the consumers maintain

 ©IJSRED:All Rights are Reserved 

the subscription state about what has been consumed and the broker remains stateless. This paper explores one such distributed messaging tool, apache Kafka, and sheds light on building resilient applications using it for stream data processing in use cases that require very low message drop rates.

## II. APACHE KAFKA

Apache Kafka [11] is a distributed data streaming platform that can publish, subscribe to, store, and process streams of records in real time [12,13] and functions as a better alternative to traditional message processing systems like activeMQ. It provides a channel between two or more processes through which data communication takes place. The processes at either end of this channel are agnostic of each other and they only concern themselves with the channel. This makes it feasible to deploy Kafka across large scale applications with very little configuration required at the service layer.

### *A. Architecture of Apache Kafka*

Apache Kafka framework has three major components that contribute to its high fault tolerance and scalability. These are Kafka clusters, Kafka connect APIs and Kafka stream APIs [14]. Kafka clusters, responsible for reliable storage of data, are composed of brokers, producer clients and consumer groups. Kafka stream APIs provide developers tools and interfaces to perform analytics as well as manipulation action on the stream of data. The connect APIs help the processes involved in the message transfer to communicate with the Kafka clusters.Producer applications write data, also called messages, identified by keys, to a Kafka topic, identified by name. Each topic is split into partitions, identified by partition numbers and topic partitions are stored in brokers in a distributed manner. Within a partition, messages are always ordered and have an offset, starting at zero. Connecting to any one broker is equivalent toconnecting to the entire cluster. Each partition designates one of its replicas as a leader and the other copies act as In-Sync-Replicas(ISR). By default, Kafka adheres to at least once delivery semantic, implying that each message in the topic is acknowledged only after every partition replica receives it and therefore ensures no loss of data.

Figure 1 clearly showcases a typical Kafka cluster. The producer applications can write to multiple topics at a time. Similarly, consumer groups can listen to multiple topics. The leader for each partition is what participates in all read and write activities whereas the other ISRs simply perform synchronization tasks. If the leader is down,

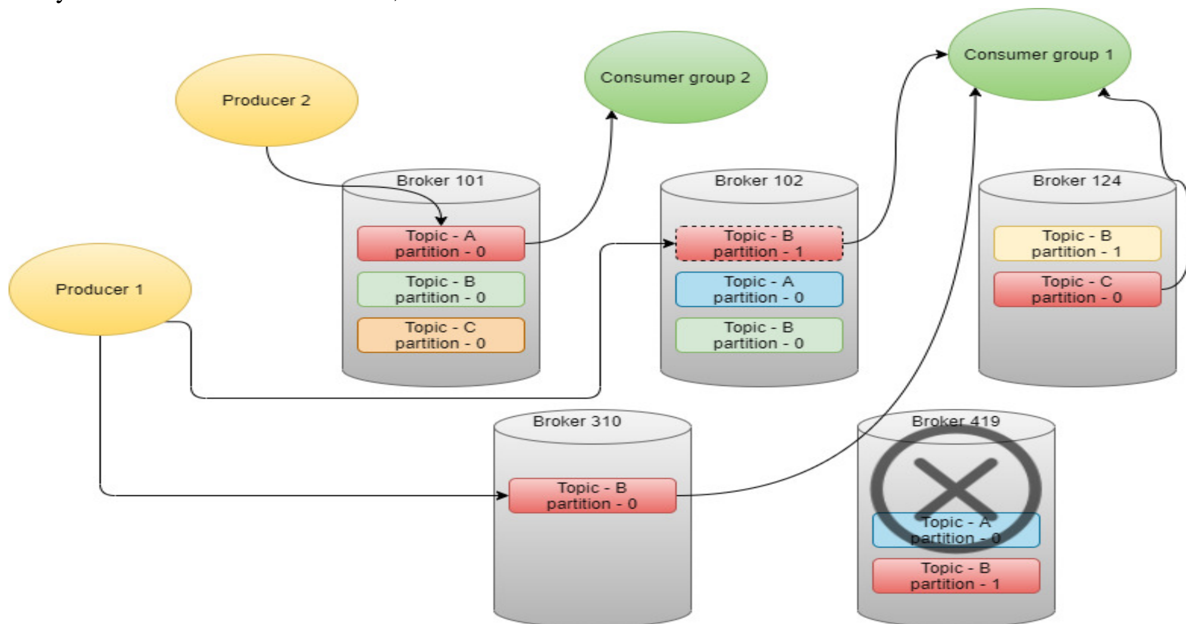


Figure 1: Kafka cluster architecture showing two different producers in yellow, two consumer groups (each with one or more consumer applications) in green and five Kafka brokers. Partition leaders are in red whereas the other coloured partitions correspond to ISRs. Broker 419 is shown as unavailable, which promoted the partition-1 of Topic-B on broker 102 as the new leader.

like the partition-1 for topic B in the figure, then some other replica gets promoted to be the next leader.

## III. BUILDING RESILIENT KAFKA APPLICATIONS

Despite being highly fault tolerant and scalable, Kafka applications [15] can run into possible errors and critical applications require that in such scenarios, the system be capable of handling the error and performing an automatic error recovery mechanism. Such errors can arise in scenarios like all brokers holding a partition going down simultaneously, producer applications unable to hit the broker IP, consumer applications unable to process the incoming data due to parsing errors among many others. Since Kafka provides the option of manually committing a message to indicate that it has been successfully processed or produced into the topic hints that delaying the commit or acknowledgement until data is successfully sent/processed is a possible solution. However, this approach, though technically possible in Kafka, runs into another issue- infinite retry loop. The producer or consumer will end up continuously retrying the same data without any success and eventually drop it. This paper aims to discuss in length about the multiple measures that developers can take in order to attain automatic error recovery with least message drops and fast processing of the stream data.

### A. Fine tuning Producer-Consumer Properties

Apache Kafka itself provides multiple configurable parameters which, when fine-tuned optimally, can ensure minimal message loss on both the producer as well as the consumer side. On the producer side, relevant properties that can be configured to achieve resilience are acks, retries, batch. Size, delivery.timeout.ms, request.timeout.ms and retry.backoff.ms [16].

Acks property determines the durability of the messages. It specifies how many acknowledgements the partition leader needs before labelling a message write as complete. The request.timeout.ms property specifies how long the producer client waits for acknowledgement before it classifies a message as failed. If response is not obtained within this duration, the message is either retried or marked as failed and dropped. Retries property sets the number of times an unacknowledged message will be sent for retry in case of transient API errors. Timeout errors also are sent for retry under this configuration. A very closely related property to retry is that of retry.backoff.ms which specifies the duration that producer waits between each retry. Using these properties, the producer waits for request.timeout.ms before sending the first retry (if specified) and then sends retries in intervals of retry.backoff.ms until either the message is successful or delivery.timeout.ms is reached or retries get exhausted, whichever happens first.

Consumer side failures occur in case of processing failures which the Kafka broker itself cannot identify. Therefore, there are very few properties that can be configured to provide better error recovery in consumers. One such property is auto.commit.interval.ms which specifies how frequently the messages need to be committed. Setting a high value implies that the consumer will commit after long durations and therefore, in case of the broker failing, all messages from the last committed offset will have to be reprocessed. This is compute intensive and leads to duplicate message processing. Therefore, an optimal value needs to be set to ensure a balance.

### B. Using External Data stores and Spark Jobs

Kafka properties can provide only limited support by means of retry. If the error is not resolved even after retries, the message is marked as fail and discarded. The system then moves to the next message. If this error persists for long, then a large number of messages can result in dropped state. Such scenarios can be dealt with if there exists some mechanism to store the messages that get dropped and then pick them up for retrying later. For implementing such a setup, an external data store like Cassandra can be used to maintain two tables. One for producer dropped messages and another for consumer failed messages. At the producer end, whenever a message is dropped, it can be written to a table and similarly, every time the consumer message is discarded, it can be persisted in another table. Inside the consumer, the message needs to be manually committed so that the consumer moves onto the next message in the queue. Apache spark jobs [17] can be written to read from these data stores at regular intervals and attempt a

 ©IJSRED: All Rights are Reserved 

retry. An attribute indicating the number of retries can also be appended to the table so that retry is attempted a fixed number of times and beyond which, the message can finally be dumped into logs for manual reprocessing. The benefit of using this approach is that the client does not wait attempting retries for the same message, rather, it sends the message to a table from where, it becomes the responsibility of the spark job to execute. The client can continue processing other messages in the queue and ensure high throughput. A drawback, however, is that since spark jobs are scheduled at intervals, it is possible that to do out of order processing for some data if one instance of it is already in failed state and an updated instance is successfully processed. This out of order delivery can be a cause of concern in some applications.

### C. *Implementing Dead Letter Queues*

In message queuing systems [18], a dead letter queue is a queue where messages that could not be successfully delivered are sent to for reprocessing by manual or automated methods. In apache Kafka, a dead letter queue can be implemented as a separate topic in the Kafka cluster onto which a consumer can write messages which get failed in processing. This technique is especially useful in cases where the message gets dropped because of deserialization errors or processing interruptions after successfully retrieving the message from the producer. To handle these message on the DLQ topic, a separate consumer can be defined which periodically polls the DLQ topic and then either attempts a retry or sends the message with its metadata to another system for manual review and analysis of these messages.

This approach provides all the advantages of using a database as described above. In addition to it, creating new topics incurs no overhead, and the messages produced to these topics can abide by the same schema. Leveraging this capability, it becomes possible to eliminate the latency and space consumption issues that is faced by using data stores.

**Table I: Comparison of using only Kafka configurations, data stores, DLQ and a combination of data store and DLQ on basis of their behaviour at producer processes in events of failure**

| *parameters* | *Kafka configurations* | *External data store* | *DLQ* | *DLQ + External data store* |
|---|---|---|---|---|
| *Blocking /Non-blocking* | Blocking retries | Non-blocking retries | Non-blocking retries | Non–blocking retries |
| *Additional latency inducing factors* | Blocking retries adds to latency. | database read-write access for every retry increases memory access time. | Wait time configured before labelling a message as failed is the only cause of latency introduction. | Wait time configured before labelling a message as failed is the only cause of latency introduction. |
| *Additional software support* | None | A dbms like MySQL, Cassandra, Mongo DB etc. set up and interfaced with the application | None. | A dbms like MySQL, Cassandra, Mongo DB etc. set up and interfaced with the application |
| *Additional space requirements* | None | Storage space to hold database along with its contents as well as metadata. | None | Storage space to hold database along with its contents as well as metadata for messages that failed even after retry from DLQ. |
| *Additional logic implementation* | Set up respective configuration properties | Write Spark jobs to read from database periodically and remove entries from it once message is successfully sent. | Create another Kafka topic to send retry messages to.<br>Create a consumer group to poll DLQ topic at intervals. | Create another Kafka topic to send retry messages to.<br>Create a consumer group to poll DLQ topic at intervals.<br>After retry, if message is still failed, then write to data store. |
| *Retry after manual review* | Not possible | Possible | Possible before retry attempts get exhausted | Possible |

 ©IJSRED: All Rights are Reserved 

| | | | | |
|---|---|---|---|---|
| ***Log of all failed messages*** | Not available | All failed records exist persistently in database | Failed records exist only as long as retry count does not expire. At end of retry, message is acknowledged and dropped. | All failed records exist persistently in database |

## IV. COMPARISON OF ERROR RECOVERY STRATEGIES IN APACHE KAFKA

These error recovery techniques were implemented and evaluated on a heavy traffic route of the order of 3 million messages per hour. The following sections give a holistic comparison among the three discussed strategies and a fourth strategy of using an external data store in supplement to the DLQ implementation on general parameters followed by specific error scenario response by each strategy in both producer end of the application as well as the consumer end.

### *A. General Comparison*

Table I tabulates the behavior of all three above mentioned strategies and a combination of using both DLQ as well as external data store on general parameters like the nature of operations, factors that introduce latency, additional resources required to implement these strategies, provision for manual review and persistent logging of failed messages. Using just Kafka configurations was easy to set up with no major changes. However, the blocking nature of retries makes the application extremely slow and adds to latency. On the other hand, both DLQ as well as data store methods perform non-blocking retries which significantly reduces the latency. However, both DLQ as well as data store requires additional software support and additional interfacing with existing application. Accessing data store is an expensive operation and therefore adds to the latency which can be avoided in DLQ method. Using a combination of both DLQ and data store provides the best of both worlds with low latency and high resilience.

### *B. Error handling Behaviour at Producer Applications*

Table II tabulates how the three techniques perform on producer side of the application in resolving errors that are encountered. Serialization errors, timeout exceptions, broker and cluster unavailability are the major causes of failure at the producer process. In these scenarios, timeout exceptions and serialization errors are high probability occurrences and hence it is important to notice that using Kafka configurations results in message loss in both cases. DLQ provides support temporarily while using data store ensures not a single message gets lost. In case of broker or cluster unavailability, both DLQ as well as Kafka configurations fail to provide support since they require an active connection to a Kafka broker in a cluster. Using data store overcomes this shortcoming. However, the high data store access time make sit infeasible to implement. Therefore, a mix of DLQ and data store gives the right balance. Primarily, DLQ tries to handle the erroneous message, and upon failure of DLQ in cases of broker or cluster unavailability, messages get diverted to the data store and can be processed later.

**Table II: Comparison of using only Kafka configurations, data stores, DLQ and a combination of data store and DLQ on basis of their behavior at producer processes in events of failure**

| *Type of error* | *Probability of occurrence* | *Kafka configurations* | *External data store* | *DLQ* | *DLQ + external data store* |
|---|---|---|---|---|---|
| | | | ***At the producer*** | | |
| ***Serialization errors*** | High | Records will be lost. | Persistent records | Records lost after retries get exhausted. | Records persisted in data store after being marked as failure at the end of DLQ retries. |
| ***Timeout exceptions*** | High | Records will be lost | Persistent records | Persistent records until acknowledged | Records persisted in data store after being marked as failure at the end of DLQ retries. |
| ***Broker unavailable*** | Low | Records will be lost | Persistent records | Persistent records | Records persisted in data store after being marked as failure at the end of |

 ©IJSRED: All Rights are Reserved 

| | | | | | DLQ retries. |
|---|---|---|---|---|---|
| ***Cluster unavailable*** | Low | Records lost | Persistent records | Records lost | Records persisted in data store directly since DLQ topic also will not be available if cluster is down. |

### C. *Error Handling behaviour at Consumer Applications*

Table III lists the major error scenarios at a consumer process and their probability of occurrence. Using only Kafka configurations performs terribly with messages being lost in all the erroneous cases. This is totally unacceptable for any critical application. DLQ helps to persist the messages for as long as its retries do not get exhausted or timeout is not reached. Beyond this, DLQ cannot provide support. Although the frequency of polling the DLQ topic can be configured to attempt retries at longer intervals for elongated time period, one the total reties is

exhausted, the message swill be marked as processed and consumer will move ahead. This implies, although the messages are persistently present in the topic, they cannot be segregated as success or failures. In order to get a log of all failed messages, the only option is to retry all messages in the topic from offset 0 and keep a track of failures elsewhere. This problem is resolved by using a data store where messages can be stored persistently and each message can be marked as success or failure. It is also possible to keep only failed messages and delete any message from the data store that gets successfully processed.

**Table III: Comparison of using only Kafka configurations, data stores, DLQ and a combination of data store and DLQ on basis of their behaviour at consumer processes in events of failure.**

| *Type of error* | *Probability of occurrence* | *Kafka configurations* | *External data store* | *DLQ* | *DLQ + external data store* |
|---|---|---|---|---|---|
| | | | *At the consumer* | | |
| ***Deserialization errors*** | High | Records lost | Records logged. Can be manually reviewed to get rid of error | Records lost after retries get exhausted | Records persisted in data store after being marked as failure at the end of DLQ retries. |
| ***Timeout exceptions*** | High | Records lost | Persistent records | Persistent records until retry gets exhausted or message gets acknowledged | Records persisted in data store after being marked as failure at the end of DLQ retries. |
| ***Cluster unavailable*** | low | no new messages received until cluster becomes available | No new messages received. Previously messages put for retry not impacted. | No new messages received. Messages put for retry previously lost. | Records persisted in data store directly since DLQ topic also will not be available if cluster is down. |

| Processing errors | Medium | Erroneous message will be dropped and acknowledged to get next message. | Erroneous message will be stored in data store and acknowledged to fetch next message. Messages logged in data store will be read later for modification or reprocessing. | Erroneous message will be written to DLQ topic and acknowledged to fetch next message. Messages logged in DLQ will be retried after an interval and reprocessed but with same data. | Erroneous message will be written to DLQ first and retry be attempted. If , even after retry, the message fails to get processed, it will be written to the database for future reference and manual examination. |
|---|---|---|---|---|---|

## V. CONCLUSIONS

Apache Kafka is a fault tolerant and highly scalable distributed system which can be used to meet the requirements of data intensive applications not just for analytics but also for real time stream processing. Companies like Netflix, LinkedIn, Uber leverage the befits provided by Kafka. This work analyzed three different approaches which can be used to further strengthen the resilience and fault tolerance of apache Kafka in data streaming applications.

Using just the Kafka configurations resulted in higher drop rates especially at the consumer end. The second method of using an external data store ensured there were very few message drops at the cost of drawbacks like high memory latency from database read and write operations and space requirements to store the data store. This can be a bottleneck in applications that want to be light weight in terms of space requirements but deal with large messages being transferred. The third method of DLQ ensured fast access, less space requirements and lower drop rates than just Kafka Configurations. However, this method encounters two issues - any failure at the Kafka cluster or broker level can break the application and secondly, failed messages are not permanently logged.

This paper identifies that using DLQ supplemented by an external data store seems most appropriate. In such a setup, all messages that need to be reprocessed are first sent to DLQ for retries, and only if the messages fail to get processed even after retries, then, those messages are logged in the data store which gives us a benefit of durable persistence while reducing database access to only such instances that fail even after retry. This method emerged as the optimal balance between space, time and resiliency for an application.

## REFERENCES

[1] Seed Scientific, 28 Jan. 2021, “How Much Data Is Created Every Day? [27 Powerful Stats].” Retrieved fromhttps://seedscientific.com/how-much-data-is-created-every-day/

[2] Xiaolong Jin, Benjamin W. Wah, Xueqi Cheng, YuanzhuoWang, Significance and Challenges of Big Data Research,

[3] ] Lopez, Martin Andreoni, et al. “A Performance Com-parisonofOpen-SourceStreamProcessingPlat-forms.”2016 IEEE Global Communications Confer-ence (GLOBECOM), 2016, pp. 1–6. IEEE Xplore, doi:10.1109/GLOCOM.2016.7841533.

[4] Ali, Ahmed Hussein, and Mahmood Zaki Abdullah. “RecentTrends in Distributed Online Stream Processing Platform forBig Data: Survey.”2018 1st Annual International Confer-ence on Information and Sciences (AiCIS), 2018, pp. 140–45. IEEE Xplore, doi:10.1109/AiCIS.2018.00036.

[5] S. Shahrivari, “Beyond Batch Processing: Towards Real-Time and Streaming Big Data” Computers, vol. 3, no. 4, pp.117–129, Oct. 2014.

[6] V. M. Ionescu,” The analysis of the performance ofRabbitMQ and ActiveMQ,”2015 14th RoEduNet Interna-tional Conference - Networking in Education and Research (RoEduNet NER), 2015, pp. 132-137, doi: 10.1109/RoE-duNet.2015.7311982.

[7] Henjes, Robert & Menth, Michael& Christian. (2006).Throughput Performance of Java Messaging Services UsingSun Java System Message Queue.

[8] Dobbelaere, Philippe, and Kyumars Sheykh Esmaili.“Kafkaversus RabbitMQ: A Comparative Study of Two Industry Ref-erence Publish/Subscribe Implementations: Industry Paper.”Proceedings of the 11th ACM International Conference onDistributed and Event-Based Systems, Association for Com-puting Machinery, 2017, pp. 227–38. ACM Digital Library, doi:10.1145/3093742.3093908

[9] E. Bayeh,”The WebSphere Application Server architectureand programming model,”IBM Systems Journal, vol. 37, no.3, pp. 336-348, 1998, doi: 1147/sj.373.0336.

[10] ASF Infrabot. (9 July, 2019). HedWig - HADOOP2-Apache Software Foundation. Retrieved from https://cwiki.apache.org/confluence/display/HADOOP2/HedWigAccessed 7 June 2021.

[11] Garg, Nishant.Learning Apache Kafka - Second Edition. Packt Publishing Ltd, 2015.

[12] Posta, Christian. “What is Apache Kafka? Why isitsopopular? Should you use it?” TechBeacon,https://techbeacon.com/app-dev-testing/what-apache-kafka-why-it-so-popular-should-you-use-it. Accessed21May2021.

[13] Redhat,” Whatisapachekafka”, Retrieved from https://www.redhat.com/en/topics/ integration/what-is-apache-Kafka. Accessed 24 May 2021

[14] Sakr, Sherif.Encyclopedia of Big Data Technologies.Springer Berlin Heidelberg, 201912] Posta, Christian. ” What is Apache Kafka? Why isitsopopular? Should you use it? ”TechBeacon, https://techbeacon.com/app-dev-testing/what-apache-kafka-why-it-so-popular-should-you-use-it.Accessed21May2021.

[15] Kumar, Manish, and Chanchal Singh. Building Data Streaming Applications with Apache Kafka. Packt Publishing Ltd,2017.

[16] “ApacheKafka.” ApacheKafka, Referenced from https://kafka.apache.org/ documentation// #producerconfigs. Accessed 7 June 2021.

[17] E. Shaikh, I. Mohiuddin, Y. Alufaisan and I. Nahvi,”ApacheSpark: A Big Data Processing Engine,”2019 2nd IEEEMiddle East and North Africa COMMunications Confer-ence (MENACOMM), 2019, pp. 1-6, doi: 10.1109/MENA-COMM46666.2019.8988541.

[18] Bouchenak S., de Palma N. (2009) Message Queuing Systems. In: LIU L., ̈OZSU M.T. (eds) Encyclopedia of DatabaseSystems.Springer, Boston, MA. https://doi.org/10.1007/978-0-387-39940-9154

 ©IJSRED: All Rights are Reserved 