Global Ordering of Messages in a Distributed Kafka Cluster

Sachin Sharma Nisha Murarka Shubham Rathi

***Abstract*—Current distributed systems generates terabytes of logs in a very short span of time. These logs contains essential data like system metric, user activities and events. Ordering of these logs are of pivotal importance to determine the correct state of the system and avoid ambiguities and inconsistencies. Apache Kafka is one of most widely used distributed message queue that addresses several key challenges in distributed log processing. However, it has its own shortcoming when it comes to ordering of messages. Even though Kafka ensures that the messages from single partition is delivered in order to the consumer, there is no such guarantee across different partitions. As part of this research we are exploring different ways to achieve global ordering of messages for a Kafka Topic.**

***Index Terms*—Messaging Queue, Distributed Messaging, Log Processing, Kafka, Scribe, Event Streaming, Message Prioritization, ZooKeeper, RabbitMQ, JMS, AMQP**

I. Introduction

With the onset of Web 2.0 and the advancement towards Web 3.0 we have seen a tremendous growth in decentralization of computer systems. These systems generates large amount of data or ”log”. The data can be generated several sources like (1) user activity events- login, content fetch, user actions, user transactions (2) System metrics like service calls, network, heap memory, disk utilization, CPU on each node in a cluster of nodes.

This kind of data analysis was done by earlier systems in an offline manner by scraping log files from production servers. But the current systems relies highly on the feedback from analysis of data in real-time to execute future actions. A very general example can be observed in trading platforms executing high volumes in stocks. These systems relies on the real time processing of transaction logs from ledgers. The performance of such systems relies heavily on accuracy and high availability of data.

Apache Kafka is distributed and scalable, and offers high throughput and also provides applications the capability to consume logs events in real time. Kafka consists of *consumer*, *producer*, and *broker*. Kafka defines *topic* as a stream of messages of a particular type. A producer publishes messages to a topic, which is then stored in a set of nodes called brokers. Consumers can subscribe to one or more topics from the brokers, and consumes messages by pulling data from brokers.

Since there are multiple consumers consuming messages from a topic, ordering of messages cannot be guaranteed at the client end. Kafka overcomes this by allowing to create partitions against a given key, which can be consumed a single consumer. This ensures the ordering of the message within the partitions but fails to provide ordering across partitions. The constraint of a partition being available to be consumed by a single consumer is a *tradeoff* on the distributed nature of the system.

II. Motivation

The motivation for this projects came from some practical application of Kafka as message pipeline for transaction data of users in a trading platform, banking transactions, social-media applications etc. For example, in a trading platform, the users events like buying and selling of stocks is considered as an event and pushed to Kafka for processing. The ordering of such events holds great significance, as we don’t want a Kafka-consumer to process the sell request of a stock ’X’ by user A before processing the buy request of the stock ’X’ by user A. The solution that Kafka provides for this problem is by creating partitions based on unique user identifiers. This will ensure that messages for a user can only be consumed by single a consumer and hence will never be processed out of order. This design fails for use cases where there is no clear definition for partitioning of topics using a key for e.g metric logs from a service.

Even after this, there are other drawbacks to use the partitions per user/key. Using partitions per user will require more memory to be reserved as a buffer for each partition. So, even if those users are not using the applications, the partitions will stay and not be used. This would result in wastage of memory and computing resource and increasing the cost of implementation. In addition, a partition will have multiple replicas and a leader is elected at a partition level. In case of broker failure or system reboot, leader election can take more time due to more number of partitions. This delay could result in system being unavailable for a longer duration. More partitions might also increase the latency of the system as there is only one thread per broker for replication and two brokers can have different replicas of the same partition. So it is only apt to look for alternative designs for overcoming global ordering in Kafka topics and not force use of partitions per user per key.

III. Related Work

Distributed systems now involves thousands of components distributed across the world with varying location and behaviour. To overcome the rigid and static nature of point to point synchronous application there arise a requirement of dedicated *Middleware Infrastructure* to convert them into large scale asynchronous application. Traditional systems like Meta Scribe[1] has specialized log aggregators that receives data from frontend machines over sockets and periodically dump them to HDFS for offline processing, this does not allow analyzed data to be used in real time. Other systems like IBM Websphere[2] messaging queue maintains the global ordering of the messages within a queue but provides a poor throughput due to delivery guarantees that requires acknowledgement for every message exchanged. Such delivery guarantees might be an overkill for logs that are non critical in nature. Messaging service such as RabbitMQ[3] and ActiveMQ[4] also maintains global ordering of data but performs poorly on large scale data as it doesn't allow sending multiple messages into a single request. This means each message requires a full TCP/IP round trip which is a costly operation.

Kafka[5] allows real time processing of the data by providing the consumers, access the messages from Brokers as soon as it is published. Brokers provide a pull mechanism to the consumers for accessing the data which prevent the consumers from getting overwhelmed when the network traffic is high. Kafka provides a superior throughput as compared to other messaging and log processing system by using a simple architecture design which leverages Zookeeper for offloading key distributed tasks like 1) Replication of data partitions 2) leader consensus 3) Maintaining membership registry for consumers and brokers 4) Keeping track of consumer data offset for each partition.

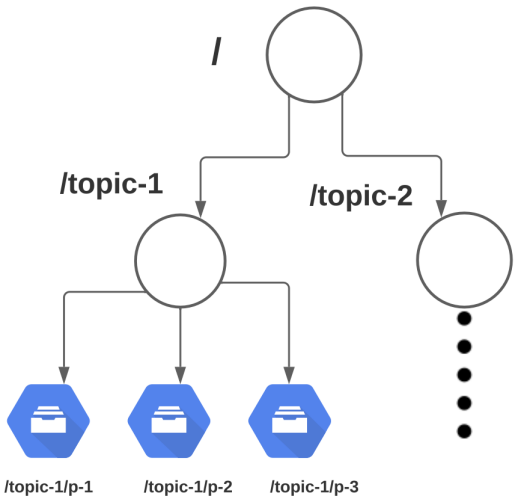

Fig. 1. A Standard file system for Zookeeper Namespace

As show in figure Zookeeper maintains the membership registry in file directory structure. The broker, consumer and ownership registry are ephemeral so if any server is added/deleted load balancing of partitions is done with-in the members of registry. It also maintains a persistent offset registry that helps in data recovery in case a consumer dies.

Kafka provides the solution for distributed messages with a higher throughput by parallel streaming of data using partitions. The messages within the partitions are consumed by a single consumer within a consumer group and thus maintains the order of the messages. However messages from the same topic in different partition can be consumed out of order causing issues for applications that requires global ordering of the messages.

At LinkedIn a cluster of Kafka is deployed in a Datacenter for offline analysis, that processes the data using HDFS. This data caters to application that requires ordering of the message but in an offline fashion. Some Log application run a pre-processing task on the data before dumping it into the producer application. This is an overhead in the application development pipeline. We aim to propose a solution that leverages the distributed parallel processing of data with high throughput from Kafka with the features of streaming queues to provide logical ordering of the messages across partitions.

IV. Project Proposal

In this research project we aim to explore various ways to maintain global order of messages for a given topic of messages. Creating partitions requires keys which might not be logical for uses cases mentioned above. Partition also restrict the number of consumers to a single consumer node. We aim to ensure *global ordering* irrespective of the consumer count.

Several other messaging technologies like AMQP (Advanced Message Queuing Protocol), JMS provide clients with options of *Message Prioritization* such that messages can be consumed/processed in different order based on their importance. For e.g. a useful usecase will be in applications where customer queries are getting addressed and a business might require to consume/process most severe cases first. Kafka being designed as an event streaming platform, misses on important features like message prioritization, which we also want to address by providing a middle layer between the consumer and the brokers.

A. Architecture and Design

A brief architecture of Kafka contains brokers with topics and partitions interacting with producers and consumers. In this sections we will discuss some key aspects of these components which will help in setting the base for our new architecture design.

Producer: Producer in Kafka is responsible for distributing messages between different partitions. The number of partitions within a topic is defined at the time of creation. The **Partitioner** class [11] by default calculates the hash function of the message key to decide which partition the message belongs to.

Consumer: A consumer group subscribes to topics it wants to receive. With in each consumer group, the partitions are distributed among different consumer such that each partition is consumed by a single consumer. The **Assignors** class [11] implements the logic to assign partition to consumers.

Broker: In Kafka every broker is called a bootstrap server[14]. A Kafka cluster is composed of multiple brokers(servers). Each broker is identified with its integer id and it contains certain topic-partitions. The interesting thing about Kafka is that at any given instance the client needs to connect with only one broker and gets connected to the entire clusters. Each broker knows about all brokers, topics and partitions through maintaining a metadata across all servers. Zookeeper play a key role in maintaining all the brokers. It stores a list of all brokers and is also responsible for performing leader election for partitions.

V. Design Implementations

We are proposing three different designs to achieve ordering of the messages and then evaluate the performance hit as compared to the current kafka implementation.

A. Aggregator and Sorter

In this approach, we propose to buffer the messages and sort them in order. The messageKey field available in ProducerHeader.java [11] file that is assigned as a unique key indentifier to each message can be used for the same. Using multiple partitions forces the consumer to maintain a buffer containing message from all partitions. If only one consumer was being used a local cache could have been maintained. However high-load scenarios often require multiple consumers, with each one reading from a single partition. Therefore, this buffer will have to be kept outside the consumer layer. The middle ware layer will poll the messages in sequence from the consumers and sort the messages which are available in order. For example if the messages arrive out-of-order it will only deliver the message that are in-sequence and out-of-order messages will be retained in the buffer until missing sequences arrive. Another way to maintain the sequential delivery is to poll the message in sequence. Though, this will remove the overhead of buffering the messages but it can be extremely slow.

Although the Aggregator and Sorter approach solves the global ordering message problem but it violates the parallelism in a distributed system. Furthermore, there is no limit on buffer size for this. If a Consumer is slow at a time, then it has to either keep buffering the message or wait until the missing message arrives.

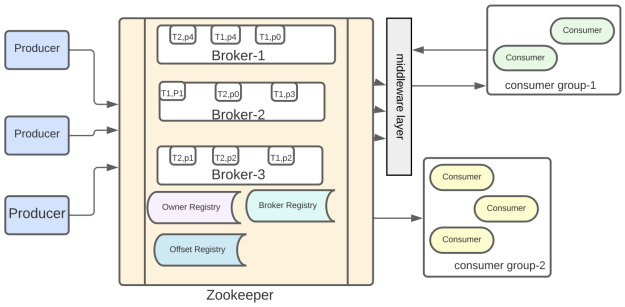

Fig. 2. Proposed Architecture Design

Algorithm 1 Aggregator and Sorter

```
Input: Consumers(All the consumers),
buffer: PriorityQueue, writeSize: Integer
while Messages in Consumers do
    for each consumer in Consumers do
        message = consumer.GetMessage()
        buffer.put(message)
        if buffer.size() ≥ writeSize then
            Deliver messages from buffer till continuous messages are available.
        end if
    end for
end while
```

B. Single Consumer within a consumer group

A naive solution to maintain order of the message will be to streamline the delivery of the message. This can be achieved by either creating a single partition per topic or by having a single consumer within a consumer group assigned to all the partitions of a topic. Having a single partition for a topics is not scalable, reason being the broker having the leader partitions can get easily overwhelmed if the network traffic increases. Therefore we propose the single consumer to be a better alternative.

In case of a single consumer, we will poll the messages from partitions in a round robin sequence and deliver the messages that are arriving in-order [13]. The messageKey field is used for determining the sequence of the messages. Out of order messages will be buffered within the consumer and then delivered in-order on receipt of prior sequenced messages.

C. Batch Commit and Broadcast Protocol

This approach proposes maintaining ordering by using consensus algorithm among producers and consumer groups independently. In order to achieve this we introduce a global batch size for a set of ordered messages at producer level. In a single poll operations the consumer receives messages in multiples of this batch size. We are proposing 3 major changes 1) Producers use Raft consensus algorithms to assign a batch number to a set of messages and write them into the broker in

a round-robin manner. This batch number provides us with a consistent sequence id across all the components in the system. 2) Now, instead of using key based allocation of partitions we will use RoundRobinPartitioner(already available in Kafka) to push the messages in the partitions. 3) Consumers will use atomic broadcast protocol to deliver the message in a sequential order using batch number. Consumer can still continue to poll multiple batches of messages but while delivering the message to the application it will deliver them in the order of the batch number and broadcast the next batch to be delivered or the last delivered batch number. Now the consumer having the messages with next batch number will deliver it to the client and broadcast to all other consumers.This gives us, highly efficient Kafka streams that can provide global ordering of the messages without any new layer or bottleneck at any layer

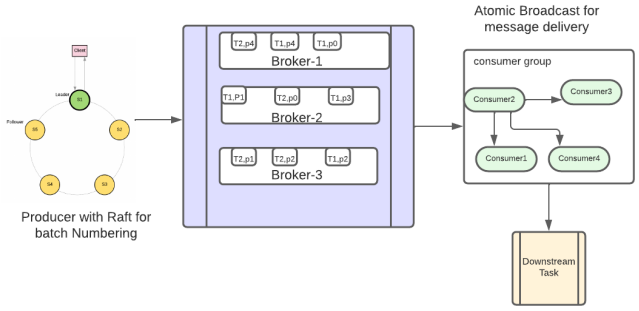

Fig. 3. Architecture Design

VI. Experiments And Results

We conducted comparative experimental study on the framework developed based on the designs mentioned in the previous sections. We are comparing our implementation's latency and throughput with that of a native Kafka framework. Since our designs act as a wrapper on top of Kafka we expect higher latency in message delivery. We tried to use comparable setting whenever possible in all the design settings. Our metric is based on Macintosh machine with 8-core CPU with 4 performance cores, 4 efficiency cores, 8-core GPU and 16-core Neural Engine. To replicate multi-producer and multi-consumer environment we spawned multiple threads with the same group Id.

In the Aggregator and sorter design implementation we used multi-threading to emulate multiple producers and multiple consumers. On receiving a client request the producer acquires a lock from a distributed lock service and generates a sequential token. The distributed lock ensures that no two producers generate duplicate or out of order tokens. The producers then commits the message to the broker. The broker contains the messages with the message key set as the sequence token id. On reading the message from the broker, the consumer will push the message in a distributed queue that maintains the messages in order and delivers sorted messages. We ensure that the messages are delivered only after a certain buffer size is reached. This guarantees that the messages are delivered in a globally ordered batch.

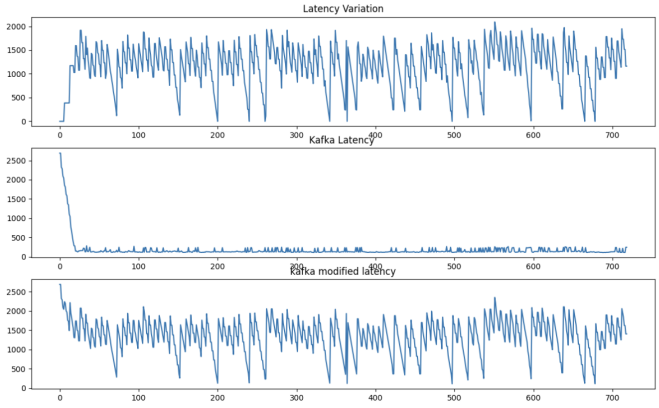

Fig. 4. Multi Consumer Aggregator and Sorter Design

To evaluate the performance of this design implementation we sent a burst of 700 messages. The average latency per request for native Kafka was approx. 6.9 ms while the modified Kafka wrapper provides an average latency of 60ms. In Fig 4. we have plotted Request id vs their corresponding latency in $\frac{1}{100}^{th}$ of a second. As observed from the Fig 4. the latency variation between native Kafka and modified Kafka is around 20 ms.

For a single consumer a single thread was used to read from all the partition. A local buffer is maintained that is responsible for sorting the messages received based on message key and deliver them to the downstream process in a globally sorted order.

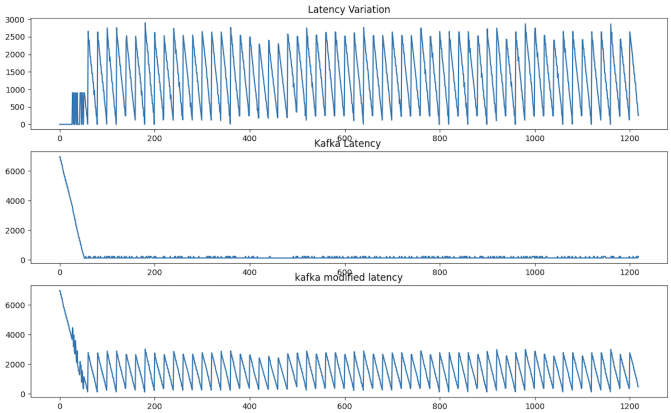

Fig. 5. Single Consumer Design

The average latency per request for a single consumer design with 3 partitions was observed around 16ms. The difference in the latency between native Kafka and modified Kafka is around 9 ms.

For batch commit and broadcast protocol we have used Raft algorithm on the producer side to generate sequential token ids.

Instead of assigning the token ID to per message we assign it to a batch whose size can be configured in native kafka. The consumer also reads from broker with the same batch size as producer. For delivering the message to downstream application we are using atomic broadcast protocol with time-out. After reading the messages from broker consumer wait for a broadcast message that informs the next batch sequence ID to be committed. If the timer expires then the consumer broadcasts the lowest batch sequence number that is present in its buffer. On receiving this broadcast message the other consumers also respond with lowest batch sequence number, each consumer locally calculates the lowest batch sequence number to decide which consumer should commit first. The consumer with lowest batch sequence number delivers the message and broadcasts the next batch sequence number to be committed.

VII. Conclusion And Future Work

Based on our experiments, we observed single consumer performs better than Aggregator and Sorter for this message burst size. Therefore single consumer would be better option for applications with low message frequency and partitions. Aggregator and Sorter performs better as the message frequency and number of partitions increase. We also observed the batch commit and broadcast protocol has lower latency in generating the sequence ID as compared to distributed locks. On the consumer end using a buffer in the first two designs introduce a fixed latency of waiting for the sorter and aggregator layer to reach a certain buffer size. With the atomic broadcast protocol with re-transmit we observed better performance. This could be because we are delivering messages on real time instead of buffering and using batch size > 1. The atomic broadcast protocol is faster but it comes with complexity of maintaining group membership. This is handled in Kafka by zookeeper.

The current implementations are tested on multi threaded environment, to better analyze the performance we can replicate these designs in a distributed environment with machines in different geographical settings. The current testing metrics checks for latency per request for one batch size, in future we want to calculate other metrics like number of requests served per second and try different batch sizes. As of now our design implementations are present as wrapper on top of Kafka. We would like to implement them as library that can be used by applications which require global ordering. The designs we implemented can also handle message prioritization given we can provide an algorithm that generates token based on certain priority rules.

Kafka's design is based on the concept of commit logs. It is like a data structure which organizes data in form of journal with new records always appended to its tail. Each of these record is assigned a position number which is its unique key. All records are immutable and allows read back and forth. Immutable nature makes it impossible for a record content to be altered nor any change in the position can be made. Through our design implementation we tried to adhere to these principles of Kafka and came up ways to handle global ordering and message prioritization.

References

[1] Facebook's Scribe, http://www.facebook.com/note.php?note_id=32008268919
[2] IBM Websphere MQ:http://www01.ibm.com/software/integration/wmq/
[3] https://activemq.apache.org/
[4] http://www.rabbitmq.com/
[5] https://cs.uwaterloo.ca/ ssalihog/courses/papers/netdb11-final12.pdf
[6] https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
[7] https://dl.acm.org/doi/pdf/10.14778/2824032.2824063
[8] https://dl.acm.org/doi/10.1145/3448016.3457556
[9] https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster
[10] https://kafka.apache.org/books-and-papers
[11] https://kafka.apache.org/documentation/
[12] https://www.dataversity.net/how-to-overcome-data-order-issues-in-apache-kafka/
[13] https://www.confluent.io/blog/prioritize-messages-in-kafka/
[14] https://www.linkedin.com/learning/learn-apache-kafka-for-beginners/intro-to-apache-kafka