**KAFKA MATERIAL**

**Kafka Installation:**

Download the binary from Apache Kafka official site, recommended for windows: **kafka\_2.12-2.8.1**

Steps to Start Kafka:

**Zookeeper**: Zookeeper works as a Service Registry in Kafka. It is the responsibility of the Zookeeper to maintain the list of brokers , track the status of nodes and messages.

Command to start the Zookeeper:

*.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties*

You can update the **dataDir** in zookeeper.properties to desired location. By default Zookeeper runs on port : 2181

**Kafka Server:**  Once the zookeeper is up and running you need to start Kafka server.

Command to start Kafka:

*.\bin\windows\kafka-server-start.bat .\config\server.properties*

You can update the kafka-logs location in server.properties to your desired location. By default, Kafka runs on port 9092.

*Note:* Apache supports only one node in community version.

**Topic:** Once Kafka is started it is time to create a Topic in order to produce or consume message. When we create a topic we need to provide the replication factor and partitions for that topic.

Command to create Topic:

*.\bin\windows\kafka-topics.bat --create --topic my\_topic --zookeeper localhost:2181 --replication-factor 1 --partitions 1*

Command to see list of Topics:

*.\bin\windows\kafka-topics.bat --list --zookeeper localhost:2181*

**Producer:**

Command to create a Producer : *.\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic my\_topic*

**Consumer:**

Command to start a consumer :

*.\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic my\_topic --from-beginnning*

**Broker:**

A Kafka consists of one or more brokers. Each broker consists of one or more partitions. Each partition can be placed on multiple machines for multiple consumers to read from a topic in parallel.

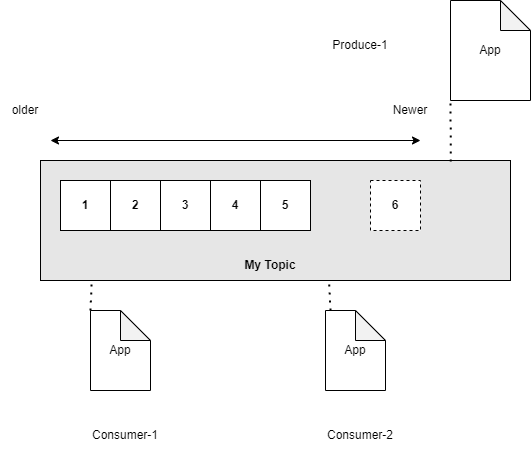
**Message Content:**

All the messages received by the topic for a kafka are time order. Kafka maintains couple of things in order for the consumer to reference the message and read it autonomously.

1.Timestamp – this the time stamp at which the message is received by Kafka

2. Referenceable Identifier – this is a unique identifier maintained by Kafka for every message, which allows the consumer to read them autonomously.

3. Payload – This is the actual payload in *Binary* format which needs to be read by consumer



**Consumer:**

Consumer read the message autonomously from message the way it does is by maintaining *Offset.* Consumer can read the message based on offset location.

When a consumer first interacts with Topic it establishes the connection and start from Message 1 and the offset moves as when the consumer consumes the messages.

If a topic has 6 messages and consumer starts reading the message 1 ,2 ,3 and so on. If a consumer loss a connection at message 3 then after the connection established, it can read from message 1 or continue from message 3.

Offset – This is same Referenceable Identifier that we discussed earlier. Offset needs to be maintained on the Consumer side so that it can start back at same point where it lost connection with Topic.

If multiple consumers are interested in same messages, then each would maintain its own offset. One consumer may be at message 1 other may be at message 6 and waiting for notification from topic for new message to continue.

Diagram

Description automatically generated

**Message Retention Policy:**

1. Message are retained by the Kafka cluster regardless of consumption by even a single consumer
2. The Default retention policy of Kafka is 7 days or 168 hours. Beyond this the message fall off from old messages to make room for new messages.
3. Retention policy is configured by hours. Retention policy is based on per topic which means that we can have hundreds of topics in a Kafka cluster, and each topic could have its own retention policy.

**Kafka as a Distributed Commit Log:**

Kafka maintains a distributed commit log, just like a RDMS. In RDMS we maintain a commit or transaction log which works as restart point in case of failure. Similarly, Kafka also maintains a commit log which serves the same purpose. While reading from topic if connection is lost, commit log would provide the last committed state and start reading again.

**Kafka Partitions:**

1. Each topic maintains one or more partitions. This is completely configurable.
2. A partition is maintained on at least one or more Brokers
3. A partition provides following:
   1. Scalability
   2. Fault Tolerance
   3. Higher levels of Through put

Single partition:

In this the topic will have a single partition. This partition will be entirely on single machine. We can replicate the partition to achieve parallelism. When we create a topic Kafka creates a folder with *{topic}-{partition}* ex- “*my-topic-0*”. This folder contains “. index” and “.log” file. The .log file maintains the formatted message and corresponding “. index” will be used to identify the corresponding offset.

Command to create single partition topic :

*.\bin\windows\kafka-topics.bat --create --topic my\_topic --zookeeper localhost:2181 --replication-factor 1 --partitions 1*

Graphical user interface, application

Description automatically generated

Multiple Partitions:

We can have multiple partitions for a single topic. This creates 3 different “. Log” files for each partition and ideally these are managed by 3 different brokers. Each partition will receive a different message and store them in sequential order. This way messages in each partition is read parallel by different consumers.

**Zookeeper**: Zookeeper works as a Service Registry in Kafka. It is the responsibility of the Zookeeper to maintain the list of brokers, track the status of nodes and messages. When u give a command to create multiple partitions then Kafka creates multiple folders *my-topic-0, my-topic-1, my-topic-2* .

When Zookeeper connects with Kafka it does following steps:

1.Maintain the log of available services and their metadata. It also identifies the leader from the available brokers.

2. After identifying the leader it assigns partitions to the leader. Once the partitions are assigned Kafka creates logs for each partition. The brokers also maintain the metadata on the partitions for ex:- my-topic-1 maintains partition-1

Diagram

Description automatically generated

**Producer**: When a producer is created, it should have the information on at least one broker that way it can the information on the partition it should send message to.

Consumer: When a consumer is created, it should talk to Zookeeper to obtain the information on the partition and the metadata. The consumer also maintains the metadata on its end to read from topic in future. When we have multiple partition receiving multiple messages and multiple consumers are reading the messages then it is the responsibility of the consumer to maintain the order of messages.

**Partitions Trade-offs:**

If we are having too many partitions:

1. The greater number of partitions the more entries Zookeeper must maintain and talk to partitions and maintain their status
2. Message order can be complex

* Single Partition for global Partitioning
* Consumer-handling for ordering

1. Then more the partitioning the longer the leader fail-over.

**Reliability By Replication:**

We have seen how Kafka works in distributed environment, now let’s see when there can be a possibility of failure in Kafka. The failure can occur in cases like

1. Broker Failure
2. Network Failure
3. Disk Failure

Once any of the failure occurs, we need to look for 2 things

1. Make sure that Kafka provides a back up broker to continue receiving from Producer or provide data to Consumer.
2. Also make sure that Kafka has not lost messages that are not yet consumed by the Consumer.

*Note: Replication factor is configured per topic.*

**Command to create a multiple copy:**

*.\bin\windows\kafka-topics.bat --create --topic my\_topic --zookeeper localhost:2181 --replication-factor 3 --partitions 3*

From above command it creates 3 replicas for each partition. This is nothing but ISR (In Sync Replication) which says no of replicas available. If they are not in sync with the replication factor then it is the duty of the User to identify the reason for ex- Broker is down, then user has to manually check that and restart the broker.

Diagram

Description automatically generated

**Producer Internals:**

To create a producer, we need to know how a producer works internally with Kafka. To create a Producer, we need to have some basic information

1. Bootstrap-Servers - A Producer need available information on topics, broker and partitions related metadata from Zookeeper, this is obtained using the bootstrap-servers mentioned.
2. Serializer type – Messages sent by producer are serialized and then deserialized. In Kafka messages are consumer if de-serializer type (consumer) matches the serializer type (producer).
3. Timestamp – By default message creation timestamp is used on consumer side.
4. Partitioner- This defines which partition the record should be send to, A Producer can employ different strategies based on values sent in Producer.

Diagram

Description automatically generated

**Kafka Producer Partitioning Strategy:**

After serializer is completed, next thing that we need to discuss is how the Partitioning is done in a Producer. There are couple of strategies that producer follows based on the input provided in producer. The strategies followed by producers :

1. Direct - In Producer if we are providing the Partition then the producer checks with Broker. If the partition is available, then it sends the message directly to that partition. If the partition is not available, it will throw *“Illegal Argument Exception”* and stop the process.
2. Round Robin – In Producer if we are not providing any partition and send the message without any key-value information then it will choose the partition based on round robin bases. For ex- default shell script provided by Kafka.
3. Key Mod-hash – In this we are providing a key – value pair then it will choose partition based on key-value pair. This uses Default Partitioner class.
4. Custom – If the producer is providing key-value along with property “PARTITIONER\_CLASS\_CONFIG” whose value contains the custom Partitioner class then we go to the custom.

Diagram

Description automatically generated

Java code for Producer: 

**Record Accumulator:**

The Record Accumulator comes into picture after the produces goes through the Partitioning phase. The Record Accumulator uses Batch processing. It accumulates records and stores them in queue and send them at once. There are couples of configurations that determine when the send is triggered.

1. Batch Size - This is a configuration that determines size of each batch in bytes.
2. Buffer Memory- This is a configuration that determines the total size of all batches.

Diagram

Description automatically generatedSend action is triggered if the batch size limit is reached or by configuring the lingering time in milliseconds.

**Delivery and Ordering:**

1. Broker Acknowledgements – property *“acks”*
   1. 0: fire and forget - risky
   2. 1: Leader acknowledgement – balanced
   3. 2: replication quorum acknowledgement – less performance
2. Broker responds with error
   1. Property – *“retries”*
   2. Property – *“retries.backoff.ms”*
3. Ordering
   1. Message order is maintained only in partition. No global order is maintained across partitions.
   2. Error Acknowledgements – If we are sending 2 messages with retries and we got acknowledgement for 2nd message then 1st message then order is lost. To overcome this we can set the “*max.inflight.request.per.connection*” to 1 with high throughput.
   3. Delivery Semantics – In order to maintain the order, we can use following *at-leas-once, at-most-once, only-once*

**Consumer Internals:**

Diagram, timeline

Description automatically generated

To read messages from partitions, Consumer need to either subscribe to topic or assign itself to partition. Both has their own advantages and disadvantages.

1. **Subscribe –**
   * + - 1. For the subscribe we need to subscribe to whole topic.
         2. If we subscribe, then the consumer will read from all partitions of topic.
         3. Consumer can subscribe to multiple topics

*Code Snippet: *

1. **Assign-**
   * + 1. If we want assign consumer to a single/multiple partition, then use assign.
       2. We can assign multiple partitions from multiple topics to a consumer.

*Code Snippet:*

**Note**: Both the Subscribe and Assign cannot be incrementally added, they have to be provided as a *List*.

**Poll Loop:**

Creating a consumer and subscribing/assigning doesn’t let consumer to consume the data, we need to poll which define the work of consumer. We can use Poll method get data for past 1 day or for some amount of time.

*Code Snippet: *

**Offset Behaviour:**

Offset is used to identify the latest message that has been consumed by Consumer. When a consumer reads a message and acknowledges it back then the partition maintains the last ack it received as offset.

Offset has some terminology that needs attention :

* 1. Committed Offset – This is position till where the message is consumed and received ack.
  2. Uncommitted Offset – This is the position where message consumed but did not receive ack.
  3. Current Position- This is the current message position that is being consumed.

**Diagram

Description automatically generated**

Poll is a synchronous method it processes only one message at a time. If we try to use multi-threaded program to run it, will throw an exception.

**Offset Management:**

1. When we use read a message it does not mean it is committed
2. We can ask Kafka to take care of commit by using “*enable.auto.commit”* to true and “*auto.commit.intreval”* to “X” milliSeconds.
3. If we use default behaviour i.e. “*enable.auto.commit”* to true then we could run into issue if consumer is slow.

For ex: Consumer read till message 3rd and ack it back, so it is committed , now we read 4th ,5th but it took less than 2000ms so it in un-committed state. Now the consumer read the 6th message and it more than 3000ms then the consumer will send ack saying 4,5,6 are completed and now completed offset would be 6th position.

If in this case the message failed due to unknown reason, then the message 6 is lost as Kafka thinks 6th message is already completed and it would start from 7th message.

**Commit Sync and Async:**

1. Commit Sync –
   1. This is used after looping through the records. It can be used in for loop but that would increase latency.
   2. As the name suggests it would wait for the response from Kafka and only then it would continue, if we are using it inside for loop we need to wait after every record consumption from Kafka which is not very effective.
   3. The Commit Sync provides with auto retry in case of failure and tries until a success is received at intervals of 100ms (default). We can change this by using property *“retry.backoffs.ms”.*

Graphical user interface, text, application

Description automatically generated

1. Commit Async-
2. Commit Async does not wait for ack from Kafka
3. It does not maintain the order in which it is receives the acks, it cannot do retry on its own.
4. If there is an exception in ack then it is the responsibility of the user to handle the exception.
5. Commit Async provides us with fallback to act on the status of the response from Kafka.

Graphical user interface, text, application

Description automatically generated

**Advanced Concepts:**

1. **Custom Serializer:**
2. **Custom Partitions:**
3. **Compression:**
4. **Advanced Settings:**
5. **K-Stream**
6. **Ksqldb**