Imp points

1) While sending a record give partition number- so that message will directly go to that partition if u don’t give that provide at least key so that partition will be calculated based on that key, if u don’t give key , then in round robin fashion it will send messages to all partitions

2) Concept of producer buffer - if producer is sending 1 million records in 2-3 seconds , it won’t send all 1 m records to broker 1 by 1 by hitting 1 million times– we have concept of producer buffer all the records will be in some buffer if buffer reaches max size then it will send in batch, but in real time if we are sending 1 individual message then how come it was sent so fast?? This I am also not sure.

3) Every message in kafka is automatically timestamped , producer api will set the time to the message which is going to send

4) add logic to reject duplicate messages incase of retries

5) Configure enable.idempotence= true for exactly once scenario to avoid duplicates, producer must enable this to avoid duplicates from producer retries, incase even if message already received by broker and if the acknowledgement was lost and if it wasn’t received by producer he may retry which is a duplicate scenario, so to avoid such scenarios keep the flag =true

Reference links

<https://kafka.apache.org/documentation/#producerconfigs>

<https://kafka.apache.org/documentation/#producerapi>

java docs <https://kafka.apache.org/36/javadoc/org/apache/kafka/clients/producer/package-summary.html>

Books <https://kafka.apache.org/books-and-papers>

Videos <https://kafka.apache.org/videos>

Misc <https://www.clairvoyant.ai/blog/unleash-kafka-producers-architecture-and-internal-workings>

Guidelines

6) Mandatorily Implement transactions in kafka-- send the messages using transactions – so that in a transaction either if all messages will be sent or no message will be sent, Caution: **- setting a unique transaction id is mandatory for each producer instance,**

And make sure consumer reads only committed data , set isolation levels to read\_comitted

**7) While consuming (ex:- acc transfer request )- make sure consumer should commit the offset frequently**

If he don’t do that- if that consumer went down and came back as he didn’t commit the offset he will read the same message again and debit the money from one of the account

So its danger consumer should always commit the offset

8**)Commit the offsets after consuming** configure commit.interval.ms=100/200ms so that consumer will commit the offsets what ever he has read for each and every 100-200ms

9) After producing each message , register the callback , so that the callback will be fired for each message, once the message successfully received,broker will send the acknowledgement back, producer will get back an acknowledgement when producer got it then callback will be fired

10) If you have more partitions, u can get more parallelism, means each producer can write 4 to different topics…and we can have more consumers in a consumer group, 1 consumer to consume from each partition, but don’t have too many partitions, if u have 10,000 partitions analyse how many consumers are there in a consumer group , if u have 2 , then each consumer should point to 5000 partitions

11) Replication factor should be minimum 2, usually 3, maximum 4, choose replication factor as N-1

By configuring min.insync.replicas=n-1, but more replication factor more will be the latency, because you should write to more number of brokers

12) Another beautiful one is acks=all, this one and min.insync.replicas should go hand in hand

Means broker will give back ack only when all ISR gave acks and at any time there should be minimum 2~3 ISR’s in the list that’s why configure min insync replicas,at anytime if that many ISR’s are not there producer cant send a message to the broker

13) **Always send the data in batches,** 1st put the data in producer buffers and send all data from partition wise buffer to broker at once by compressing data by using some compression techniques

Use some standard serialization frameworks like apache avro which is more compact

If u want to accept and store 200 messages at once then receive 20KB at once, for that producer should send 20KB data at once by **setting batch.size=20KB or linger.ms=9000** (means either wait for 9 sec or 20KB data to arrive ) whichever is met earlier data will be send as a batch ,

This may increase latency but it will increase the throughput as network bandwidth will be utilized effectively

14) Use the **right serializer like apache avro – which decreases final payload size as final payload doesn’t contain schema and data will always be checked with schema present in schema registry**

**Either send data in avro format or byte array, convert to byte array using ByteArraySerializer before sending the data to topic**

**15) Use Multi-threaded producer –**

**16) Retries-** we should retryonly transient exceptions (Retriable exceptions are those which will be success on a retry ex:- network temp down)

We have to think **what to do when all retries are exhausted**, is that message so critical are we going to save it In database, if yes when are we going to send that message again to target system?

Don’t retry continuously, we should give some time for broker to recover before we re-attempt so tune “**retry.backoff.ms**” accordingly change it to more than 5~10sec instead of default 100ms

17) **Ordering** – if ordering is so important at least write some set **of related/inter dependent messages to same partition**

**18) use compression - gzip**

Higher the replication factor, better the durability

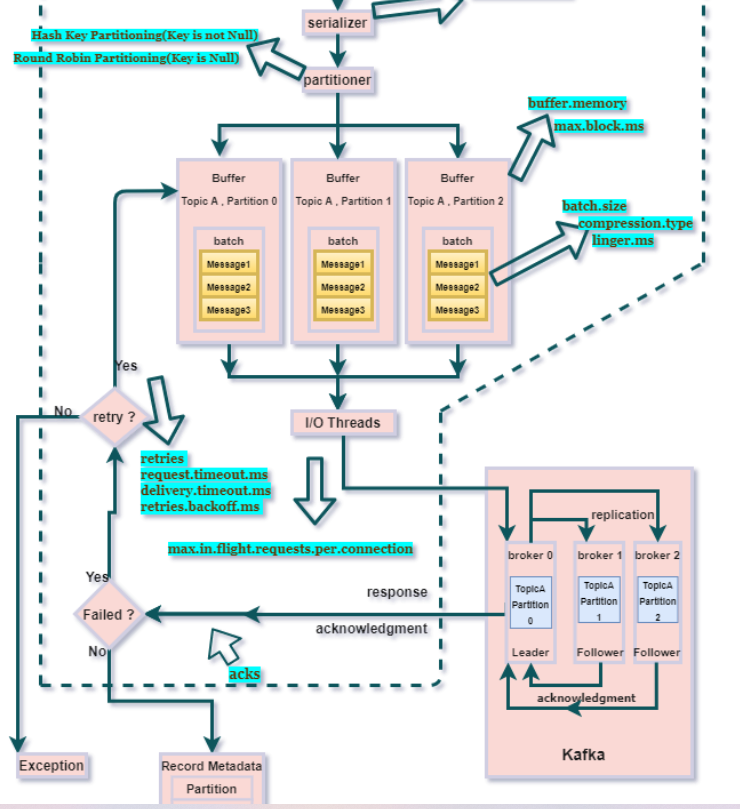
Always think about this worst cases

1. What if the broker went down without committing the offsets

Doubts

1. If producer send 1 million messages , then will we receive 1 million acknowledgements for all messages, if yes but how do we know which acknowledgement is for which message among those million messages

Flow of messages from producer to broker



--Think – if u write producer.send(producer record) 100 times, does framework should send this to broker 100 times think how much network roundtrip, to avoid that we have “buffer” component

1. If producer mentions partition number then message will directly go to that partition, else partition number will be decided based on hash value of key, if key is also null then default partitioner ( round robin) will takes place and distributes messages to all partitions equally, now sticky partitioner came
2. Producer.send (topic,partition,key,message) once we send 1st key & value will be serialized into byte array using key, value **serialize**r,

Then it will go to **partitioner**, here it will check if producer Record has mentioned any partition else it will decide the partition based on the partition strategy (Hash value of key or default partition number or round robin strategy or stick partitioner), once partition is decided then

Then those messages will be send to **respective partition buffer then (like people going to same location will be in same bus) exclusive buffer from** IO threads will send the messages from buffer memory to broker

A separate IO thread is responsible for sending those batch of records from each partition buffer to broker

* Broker after successfully receiving the message, he will send back RecordMetadata object as an ack to producer and it has the topic, partition, and offset of the record within the partition.
* If he didn’t sends back acknowledgement to producer then producer can resend the same message again to broker assuming broker didn’t receive the messages
* If the message fails to be received by the broker, it will return an error as a response,(if the message is not received by broker, then how we will get the error response)

IO threads will wait for either linger.ms (120 seconds) / batch.size(500KB) assume the data sits in buffer but it has to wait for some cfgd time or some size like 500KB if any of conditions are met

Then only IO threads will take data from buffer and it will give to broker

1. Broker will save the message into log file, then he will send back the ack to producer (he will send back acknowledgement to producer only if all in sync replicas received the message )

Producer

1. Note:- 1 producer can send data to multiple topics,

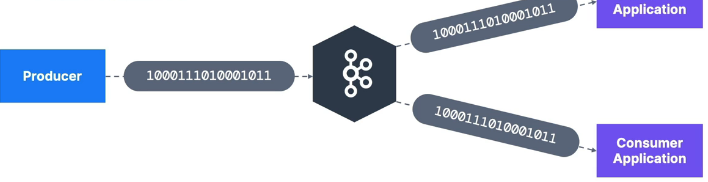
Ex:- producer.send(new Producer(“topicName-T1”,partitionnumber,Key,Value));

producer.send(new Producer(“topicName-T2”,partitionnumber,Key,Value)); //now same producer is sending data to another topic named t2

Producer always sends the data to the leader partition, similarly consumer always consumes from the leader partition

When you are creating a producer its mandatory to tell the application id, then only the broker will come to know from which app these messages are coming from, so set this ID while configuring the producer

Note:- kafkaTemplate.send(topic,new Employee()); if u send the data like this generally producer sends in the form of object, but internally **it will be serialized into bits and bytes** –I mean 0’s and 1’s only those will be sent to the broker



Even consumers will read those 0’s and 1’s and they will be deserialized to the objects

Sample producer code

|  |  |
| --- | --- |
| Its always recommended to send data using avro or byte[] using a binary serializer instead of json/plain text serializer |  |
|  |  |

Purpose of producer id is to track the source of message

import org.apache.kafka.connect.json.JsonSerializer;

Properties props = new Properties();

props.put(ProducerConfig.ACKS\_CONFIG, "all");

props.put(ProducerConfig.CLIENT\_ID\_CONFIG, "HelloProducer");// this tell which producer is this

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, localhost:9092,localhost:9093);

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, IntegerSerializer.class.getName());

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName());

//p.put(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***, JsonSerializer.class);

props.put(ProducerConfig.TRANSACTIONAL\_ID\_CONFIG,”diff for each producer instance use uuid.randomUUID()”);

props.put(ProducerConfig.TRANSACTIONAL\_TIMEOUT\_CONFIG,12000);

KafkaProducer<Integer, String> producer = new KafkaProducer<>(props);

Callback callback=(recMetadata,exception)->{  
 if (exception == null) {  
 int callbackExecutedCount= counter.getAndIncrement();  
 System.*out*.printf("\n %d. msg went to topic --> %s partition --> %d , offset --> %s ",  
 callbackExecutedCount,recMetadata.topic(),recMetadata.partition(),recMetadata.offset());  
 }  
};

logger.info("Start sending messages...");

for (int i = 1; i <= AppConfigs.numEvents; i++) {

producer.send(new ProducerRecord<>(AppConfigs.topicName, i, "Simple Message-" + i), callback);

// this is a fire and forget ,means it will send this record to producer buffer and control will come back

producer.send().get() //this .get() will wait until the message reaches to broker- This is a blocking call// If u are sending data at higher speed don’t use this get because the control blocks here itself

//Because 1st data will be sent to buffer, from buffer IO threads should pick the data and these should send to leader broker and follower brokers should also consume those message

and then broker will send ack only after successfully Writing those messages to log file and that ack should reach the producer, till that time this will be blocked

//If u use this .get() then u cant use producer batching because control will struck and waits until broker received this message

}

|  |
| --- |
| Producer Record |
| **Topic – Mandatory field because if u don’t know topic name where u will send the message** |
| **Message value - Mandatory field** |
| **Message key –optional (Based on the key partition number will be decided)** |
| **Partition – (if u give partition , msg straight away it will go to that partition only, else partition number will be decided based on the hash value of key- like even number key on 1st partition, odd number key on 2nd partition)** |
| **Timestamp** |

Partitioner

<https://cwiki.apache.org/confluence/display/KAFKA/KIP-480%3A+Sticky+Partitioner>

most probably use the default partitioner

As per above message for producer record ,

1. if we set partition number message will directly go to that partition,
2. else partition /folder will be decided based on the hash value of the key

new ProducerRecord<>("wellsfargoo",”skey”,"vammov"+e)

here skey is a string , hash code will be calculated on the key and that will decide the partition number

1. If key is also not there ah?? Then default partitioner will send all messages to all partitions(folders) in round robin fashion

properties.setProperty(“partitioner.class”,roundRobinPartioner.class);

You can also Create a custom partitioner and provide that name to that

Props.put(ProducerConfig.partitioner\_Class\_CONFIG,MyPartitioner.class.getName());

There are 2 default partitioner available

1. **Hash key based partitioning :-** same like hash map internal working , here also for every key hash code will be calculated and based on that partition number will be decided

**Round robin partitioner:-** if we didn’t send the key in the producer record then partition will be round robin fashion , means messages will be sent to all partitions(folders) this is worst performance as here there is no batching extra overhead of each record individually

**Till kafka 2.4 default partitioner is Round robin partitioner ,the main disadvantage is if they have 4** partitions , if we choose round robin partitioner ,each partition will get 1 message ,no batch will be fulled and since the batch is not full we cant do proper compression

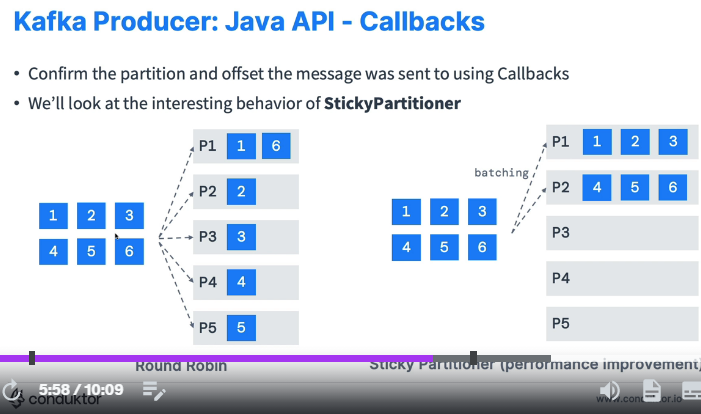
**Sticky partitioner**

In sticky partitioner ,until batch size is filled all the data will go to the same producer batch catch partitioner ,(for each & every producer we have producer batch once batch is filled then some compression techniques can be applied and size of payload will be lesser and can transmit faster)

High payload 🡪compress at a time and send that immediately

If u prefer round robin partitioner data will goto all the partitioner buffers and since data will be low in each batch then compression needs to apply on each batch and we need to apply on low data because we may hit linger.ms so prefer Sticky partitioner

Efficiency is high when u prefer sticky partitioner, because



In round robin partitioner he will send messages to all partitions equally

Whereas sticky partitioner will send some 10-15 messages to partition-1 and another 10-15 messages to partition-2 like that why? Because this is a batching and it’s a performance improvement it will send all the batch records at once

Kafka default batching size id 16kb ,but IF WE KEEP linger.ms=2000 (2 seconds) whether the batch is filled or not, after 2 seconds whatever is there in the batch those will be sent,

Serializer

Serialization is mandatory to send data over network, else u can’t send the data over the wire itself, Serialization is a process to convert the data into bytes on then it can travel over wire

Only serialized data will be kept in producer buffer as serializer is the 1st component in the architecture

We have 2 types of serializer

1. JSON serializer – heavy load on network, because it embeds lot of schema information while carrying the message ()
2. Avro serializer – fast and used mostly in big data environment – Fast because AVRO doesn’t embed schema information hence final **message size will be small** so it uses less bandwidth

kafkaProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");// even if u don’t send key we should configure this key serializer

kafkaProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Apache Avro is serialization framework which is used to serialize the data so that data can travel in the network

A separate serializer document is made in this folder

Even though u didn’t pass key , key serializer is mandatory (if u don’t pass u will get exception -  **Missing required configuration "key.serializer" which has no default value.)**

Message timestamp

For every message n Kafka time stamp will be there among below 2

Auto time stamping:- As you see, producer record have a field called TimeStamp

1. **Setting Producer time --Create Time –indicates when message was produced** TimeStamp set by producer – its create time like when producer is going to send a message then producer will set this time to the message

**Message.timestamp.type=0 if u give zero now producer will set the time at which it is sent to broker to the message as**

- create time timestamp -this is the default and recommended one -

1. **Setting broker time --Log Append Time** Time stamp set by broker on arrival to the broker –In this case time stamp will be set by the broker when he received it

Message.timestamp.type=1

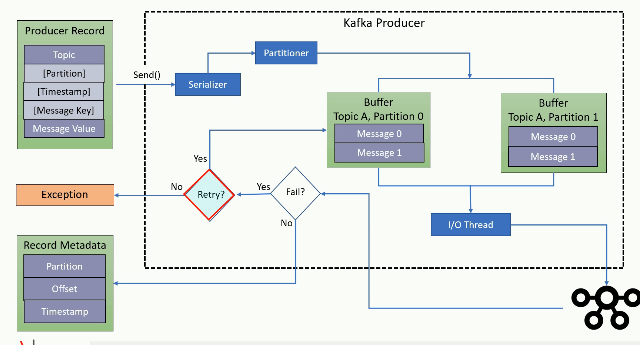
Producer buffer

we have a buffer memory for every folder partition

kafkaTemplate.send(topicName,message) // this will actually send the message to the buffer partition which resides in the producer memory, it will not send the message immediately to the broker, IO threads are real senders when the buffer max capacity is filled then IO threads will send the data from buffer to real kafka broker cluster.

The default producer memory is 32 MB , if u want you can expand the producer buffer memory

**Buffer.memory for this key** we have to set the value



Producer batching

**:-** assume a bus if one passenger comes will it start? Either it will wait for maximum 30 mins or it will wait for minimum 40 members, if any criteria is met early then only they will start bus

Similarly messages are stored in bus or producer buffer, then later based on our configurations io threads will send the messages from buffer to broker

If io thread after sending msg if it didn’t get the ack then it will resend the same message again

Producers often batch multiple messages together for improved efficiency when sending them to the broker. This batching helps reduce the number of network requests and potentially improves throughput

In case of success we will get the record metadata callback , once we got ack we can find that message was written to which folder(partition) , which file (segment)

Producer memory must be bit high, especially if incoming records are coming in larger volume buffer should be able to hold right. So make sure u have large buffer memory,

Blocking scenario :- Generally producer will send data to buffer and leaves as non blocking , if producer buffer is low and if huge volume of data is coming , producer needs to wait until , until the buffer is cleared means until io threads take data from buffer producer will get blocked, and once buffer is free , producer will push to buffer and leaves

If producer is sending 1 million records at a time producer wont really send them all at once

It will be stored in a producer buffer, once buffer max capacity is filled then only it will send and interesting thing is for every partition we have separate buffer inside the producer itself

Doubt:- let’s say if buffer max capacity is 1MB if we send 1-2 messages definitely it wont fill , in that case to full the capacity it take hours, but how are we getting instantly and why are we not waiting till buffer is filled, ex:- in our 1DSTR app if u send 1 message having bytes it is immediately coming na – may be due to linger.ms

1. Instead of buffer size concept, we should have had buffer count concept, so that when 200 messages came to buffer we will send all of them at once

Summary

1. Producer.send(key,value);- Seriailization of key and value , decide the partition number

First the key and value will be serialized by the respective serializer

If partition is there it will put that serialized message into that partition buffer ,else it will decide the partition based on the key , if key is also absent then in round robin fashion message will be sent to one of the partition

1. Serialized message will sit into the producer buffer
2. IO thread will send the message to the cluster
3. If the producer received the acknowledgement then no retry , if the producer didn’t received the acknowledgement then retry
4. It will retry for configured number of times, still not working ha then then throw an exception

**Scaling producer**

A single kafka broker is enough to receive 100-1000 messages per second

It says no need to create multiple **KAFKA Producer** objects in single application

Just create multiple threads so that all threads can use same kafka producer object

Refer sample project **“03-multi-threading-completed”**

If you have a trading application where u should send the data at high speed , like 10,000 messages per second then u may need to use multi-threaded producer

Ex:- if u have 2 files which Is having bulk data in each file

Then create 2 threads ,whereas each thread will read parallelly from each file and each thread after reading one line it will send that line to producer buffer internally once that entire producer buffer is reached then all will be sent at once, each producer will have 1 buffer , so as per our schedule lets have 2-3 threads all can happily share the same kafka producer, each thread can use same producer object

Producer exceptions

When data is coming at higher rate and if we have low producer buffer size and if we have a slow broker (consuming at slower rate) then producer will hold the data in buffer for some time and incoming data is huge if there is no space left in buffer, then producer will block for certain time / max.block.ms after that it will either drop the message or it will throw the exception. Then

By considering this case, a developer should think of retrying the message when the ack is not given to us

Producer retries /acks/ Duplicate message chances /message guarantee

Generally broker will send acknowledgement only when the message is received by leader & 1 ISR,

We can configure “retries” this many times message will be retried

And producer will wait for “request.timeout.ms” time before attempting a retry, if even after “request.timeout.ms” if broker didn’t send any ack, then it will be retired

And there are 2 types of exception 1) Retriable Exceptions (A “no leader” exception, connection down ) and non re-tryable exceptions (message size is too large than 1MB in those case no retry only direct exception will be thrown)

**When producer will send same message again? & when retries will happen?**

1. When all followers are out of sync, even if leader consumed the message as no ISR received that message, then leader will not send back ack to producer, producer will assume leader didn’t received the message, then producer will retry or re-send the same message to leader again In this case duplicates can occur, like if only team lead knows everything and if no team member knows as much as TL knows, incase if lead is OOO nobody

Can be an alternative to him, Hence to avoid this situation only kafka broker won’t send back ack to producer

1. Or When broker sent back the ack, but if acknowledgement is lost before reaching to producer then also as producer didn’t received ack, he will resend the same message to leader again

Then producer resend by assuming the leader didn’t received the message properly

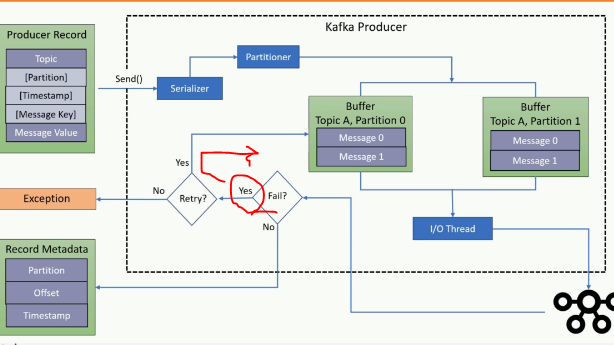
**Why leader won’t send ack to producer if all followers are out of sync?**

Because if leader keep on sending ack to producer incase if leader dies followers brokers won’t have the latest data so it leads to data loss, hence the architecture is like that,

It sends the ack only leader and at least one follower has the latest data or when at least one follower is in sync

Generally if we send 1 million messages we should get acknowledgements for each message we received, let’s say if we have configured acks=all means leader will send positive acknowledgements only when all followers / all ISR’s received it, let’s say leader received the message but followers didn’t received it incase failed acknowledgement will be sent back, and as usual producer will retry to send the same message again now I don’t know what happens then will broker accepts it?

Ok for now understand that it will retry when it gets a negative acknowledgement



Duplicate message –atleast once vs atmost once

Below are the scenarios of coming duplicate messages

1. Lost ack in the network 🡪Generally when we send a message by the producer , broker after receiving the message he should send acknowledge for the received message , incase broker while sending back acknowledgement to the producer if some issue came, then that ack wont reach to the producer in that case producer will retry sending the message which is already received
2. In case of ISR are lag behind broker– if leader didn’t give acknowledgement due to lag of follower broker but leader received🡪 generally replicas are considered as follower brokers , generally these followers will be little behind the leader , like if leader consumed or read 100 messages followers would have read 70-80 messages only , lets say if acks=all means when a message received by the broker 1) leader broker will consume happily 2) when these followers are out of sync and since they didn’t consume properly since ack=all ( means send back ack only when all followers consumed the message) if one of broker has missed to consume it and in this case timeout as we didn’t send the ack back to the producer and producer will think msg not received and he will resend he wont consider that message was already consumed by the leader broker or not , from his side since ack didn’t came he will resend in this case this is duplicate message will be sent to broker even though leader has already consumed the same msg he will get it.
3. How many retires- if producer gave as retries=233434, then if producer didn’t get ack then for this many times & until this much time (delivery.timeout.ms=120000) producer will keep on sending duplicate messages

Among these even of leader broker consumes if any ISR didn’t consume then producer will simply retry until delivery is timed out

1. d

Incase if we configure zero retries= atmost Once , then even if producer didn’t received the ack then no retry will be performed (this is the 1 worst solution because possibility of loosing the message is high)

So this is bad because even if something went wrong it won’t retry leads to data loss

**Best solution is u implement the duplicate record identification** mechanism based on the co-relation id or ask producer to send a unique identifier to identify each message

Kafka is by default atleast once- means retry always

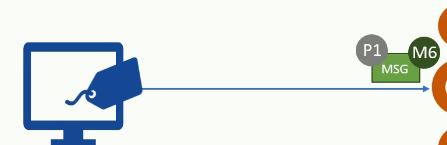
Solution for duplicate retry-idempotent producer

Enable.idempotence=true

This solution is only for retry mechanism, when message successfully received by the broker and if broker while sending the acknowledgement if that ack is lost , producer will retry sending same message na eventhough the message successfully received ,producer will send same message again this is the duplicate scenario

How does broker solves this duplicate problem??and how it works internally

1. Enable.idempotence=true

If you configure this parameter, producer while sending the message for each and every message it will append the **producer instance id** and **sequence number** 

For a particular session, the Kafka leader broker assigns a **producer id(PID)** to each producer . Also, the producer assigns a monotonically increasing **sequence number (SqNo)** and to each message before sending it. The Kafka broker keeps track of the largest **PID-SqNo** combination on a per partition basis. When a lower sequence number is received, it is discarded and that’s how it avoids duplicate messages

The configuration**acks = all** and**retries = Integer.MAX\_VALUE** helps to achieve the ‘at-least-once-delivery’ of the message. Hence, that’s how an idempotent producer works and achieves the ‘exactly-once’ delivery of the message.

if producer sends the same message again with same producer instance id + **sequence number** the broker will reject

This idempotence is only guaranteed for producer retries,

Note:- If application itself is wantedly sending the duplicate message This solution won’t work

This is the solution only for retry mechanism, if u configure this while retrying ,even if producer send duplicates, broker will reject the duplicates due to producerID+sequence number combo would have already received earlier and sequence also wouldn’t have matched with current sequence number

 Best option

Question:

1) assume leader received the message and followers are out of sync so broker didn’t gave ack to producer now bec of idempotence even if producer resends then broker will simply ignore / broker will give ack without accepting the message?? But how come broker will give ack without writing to log file, after writing only it will be able to send the metadata to producer, only thing is ISR may not be in sync

2) what if we keep both these flags Enable.idempotence=true && acks=all (leader will send back only if all ISRs received the message),

in this case even if no follower or ISR received the message and broker wont send back ack to producer, due to enable idempotence=true will broker consider this message already came? If it rejects message

then what is the use of acks=all where leader wantedly didn’t send back ack to producer, so I think there is a conflict between properties Test it

Due

1. **Consumer:** Utilize transactional processing with isolation.level=read\_committed and commit offsets within the transaction

So make sure consumer only reads committed data

1. . Message Deduplication Filters:

This approach involves maintaining a record of processed messages on the consumer side. Here's how it works:

Include a unique identifier (ideally a message ID) within each message payload.

Implement a deduplication filter in your consumer logic. This filter checks the message ID against a stored record (e.g., database table) of processed messages.

If the ID is found, the message is considered a duplicate and discarded.

Transactions in Kafka

Why and when I have to use transactions in kafka?

When to use transactions:- if u are sending 10 messages and If u want all or no messages then use transactions

Note:- when u begin a transaction and started sending messages even though u don’t commit txn, messages are written to broker with abort flag because of which consumers can’t read them until that txn is committed

1. Transaction **means – either all or nothing** - It says either all messages within the transaction are committed or nothing is committed , Transactions should be committed if all parts are succeeded ,
2. Every producer should have different transaction id whenever u are sending the data in transactions props.put(ProducerConfig.TRANSACTIONAL\_ID\_CONFIG, UUID.randomuuid().toString());

Means for 1 producer will have 1 transactional id even if he is going to perform 100 transactions, this is the id for that producer instance to uniquely identify which producer sent this message ,ex:- in production server if we have 4 producer apps each one should have only 1 txn id

1. only committed data can be consumed by the consumer, if u have sent to broker and if u didn’t commit the data that data may present in the broker , but the consumer cannot consume it for this u might need to set the isolation level to read\_comitted
2. Transactional id which is set to the producer instance must be unique even in cluster environment- each producer instance in a cluster must have a different transaction id to uniquely identify that producer why?? Because when you are writing the messages with transaction the messages will be written to broker and u will get acks also ,but only thing is those messages will be written with abort flag and each message will have producer instance id, so that if producer aborts the transaction they will be erased ,

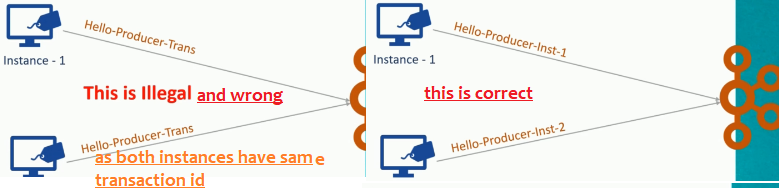
If all producers’ instances are writing with same id and if all are using transactions are aborted then whose messages to revert?? So transaction id is mandatory

!!!! Whenever u are using the transactions in Kafka make sure u should have different transaction id for each producer, especially if u scale the application in our prod if we

Have 4 instances of our application then we will have 4 producers then each producer should have different transaction number

Internally it is similar to database commit, like first all instructions will be executed in database once u commit changes are saved, here also in a transaction if we are sending

100 of messages first all messages will go to the kafka and after committing it will persist



You can’t run 2 instances of producer with same transaction id-try to know what will happen if we keep same

props.put(ProducerConfig.TRANSACTIONAL\_ID\_CONFIG,” Hello-Producer-Trans”);

if u set as above all producer instances when this appn running on a cluster all instances may take same transaction id

*props.put(ProducerConfig.CLIENT\_ID\_CONFIG, AppConfigs.applicationID); //This is not important only transactional id is important*

props.put(ProducerConfig.*TRANSACTIONAL\_ID\_CONFIG*, AppConfigs.*transaction\_id*); //This is important and 2 producers cant have the same transaction id and 1 producer needs to call only 1 time

kafkaProducer.initTransactions();*// it is like registration*kafkaProducer.beginTransaction();

kafkaProducer.send(new ProducerRecord<>(*topicName1*,1,"Mesage-10") );

kafkaProducer.commitTransaction();

*kafkaProducer.initTransactions();// Here again this init is not required for 1 producer only 1 time*

kafkaProducer.beginTransaction();

sending in transactions

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("transactional.id", "my-transactional-id");

Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());

producer.initTransactions();

try {

producer.beginTransaction();

for (int i = 0; i < 100; i++)

producer.send(new ProducerRecord<>("my-topic", Integer.toString(i), Integer.toString(i)));

producer.commitTransaction();

} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {

// We can't recover from these exceptions, so our only option is to close the producer and exit.

producer.close();

} catch (KafkaException e) {

// For all other exceptions, just abort the transaction and try again.

producer.abortTransaction();

}

producer.close();

How Kafka broker knows there is another producer with same transaction id

producer.initTransactions(); //when every producer starts that producer should execute this line when this line is executed it will tell

broker its like a registration ,incase if another producer with same id came by calling initTransactions()it will abort the new producer

with existing txn id

This method needs to be called after transactional id is set,

1. This will check any transactions initiated by previous instances of producer are completed or not with the help of transactionID that’s how broker identifies with that transaction id any producer already present or not, if it is already there then it will throw error

What is the use of transaction id??

Every producer instance should have different transaction id , if same producer was offline and came back after some time and if there are any unfinished transactions they will be aborted and all the messages will be written in abort mode , by default all messages are written abort mode

Assignment:- what will happen when 2 producer instances have transaction id

**If 2 producers have same transaction id then u will get below error**

org.apache.kafka.common.errors.ProducerFencedException: Producer attempted an operation with an old epoch. Either there is a newer producer with the same transactionalId, or the producer's transaction has been expired by the broker.

On say producer.initTransactions(); , producer.beginTransaction()

Whether u start the transaction or not, by default all the message will be written to broker with abort flag, once u commit the flag will be changed to **“c”** for each message and only messages which are marked as committed can be read by the consumer

When u say kproducer.rollback(); as every producer is identified by its transaction id, as messages are already written to the broker, broker finds all the messages written with that transaction id and all those messages will be rollbackedby the broker ,

Now think if 2 producers have same transaction id which is the uniquely identifiable , when 2 producers wrote messages to the broker, if 1 producer issued rollback , as messages are written with same transaction id which producer messages should be rollbacked?

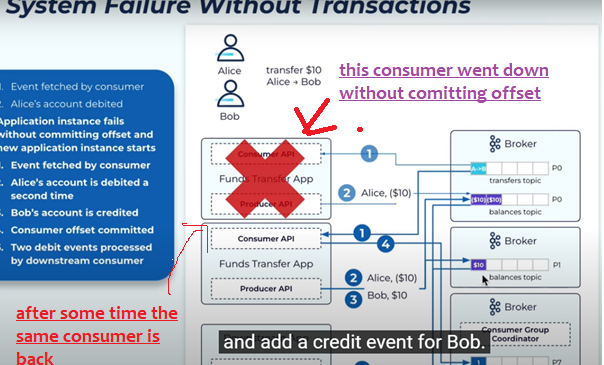
Hence each producer instance should have different transaction id

Ans:- application wont even start if another producer also have same transaction id So in prod also when second instance cant start- because when u say

*What is producer.beginTransaction()?*

These transactions are initiated by the producers, if the same producer instance has initiated the transaction and if he didn’t close that transaction, this producer.beginTransaction() statement will close or it may abort un finished transaction if exists

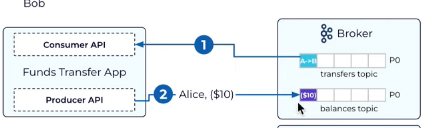
Producer.commitTRansaction(); producer.abortTransaction();

s

In the above image see , as there is no transaction enabled debit request message has been kept 2 times into the queue.

#### Without transaction what is the issue

Let us say a money transfer event, there are no transactions,



1. Step -1 in above message is a message transfer event which the consumer must read
2. consumer after consuming the message it should produce 2 events –debit event, credit event
3. consumer -1 – while transferring money, he should have kept both debit and credit messages into topic, but c1 kept debit message and went down without committing and without keeping credit request
4. after some time again it kept both credit and debit, so here 2 times debit event was kept, to avoid this we should have either sent both at once or just don’t send any message here we should have used tranactions

First consumer will read a message called message transfer event from A --> B and consumer didn’t commit the offset after reading and consumer kept a money debit message from Alice into another topic, and suddenly it went down

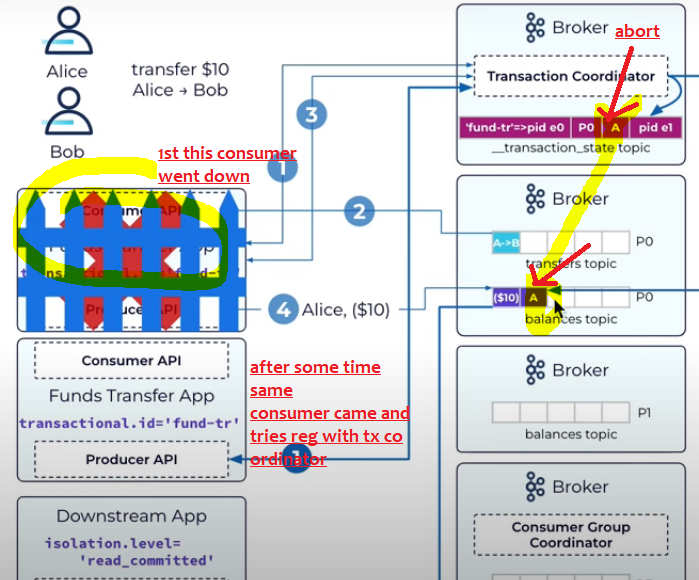
And after sometime it is back and since last time that consumer didn’t commit the offset it will read the same transfer message again from Alice 🡪 Bob and puts Alice money debit request message into the topic second time which is wrong () and keeps money credit message to Bob in another topic

If first time itself if transaction are there, since credit message to BOB has failed, debit tx wouldn’t have committed issue would have solved, hence transactions are very important, since there is no transaction that’s why this duplicate message came

So when transactions are there even when application fails it will guarantee only single time message will be delivered

#### Exceptions during a transaction

1. first every producer instance should have a transaction id even in a cluster environment also with that this producer it will register to a transaction co-ordinator and get the unique producer instance id



Now see as per above image 1 st consumer read the message and stored in topic and didn’t commit offsets and went offline and after same consumer came and registers with transaction co ordinator and see this transaction id is already registers and *after seeing all that transaction co ordinator tries to abort all pending transactions and those messages which are* ***already written to the broker will be marked as aborted*.**

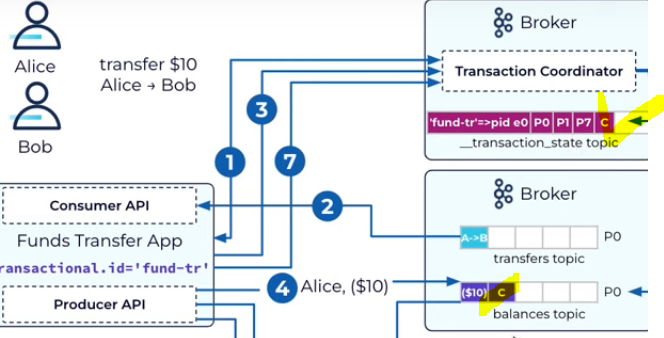
Q) U may get a doubt if these messages are already written what if they are consumed in between ,

A) Simple these messages are not in committed state, consumer reads those messages which are in committed state

#### 2- Successful or committed transaction

Success or failure , during a transaction in both cases messages will be written to kafka broker

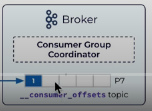
If success and when you **commit the transaction**, then commit marker will be marked on that message see below message, if transaction is aborted then message will be **marked as abort** refer above



Note:- even though both messages are in the broker ,consumer can see only committed messages

Committed offset storage

When the consumer committed the offsets all will be stored in the broker

offsets should be maintained by broker, not consumer, like the team lead should know how far the team member has worked

if consumer stored the offsets , if that consumer went down we don’t know how far the consumer has read, if broker know then he will give the offset number to new consumer to consume from that instance

Callbacks – Acknowledgements

It’s always recommended to attach a callback, for every message we send we will get acknowledgement. When we receive the acknowledgement, our callback method will be fired

In local for debugging purpose , try getting acknowledgements for each message,

Advantage is:- once we attach a call back we will come to know if a record is failed then we can save that record in database for later purpose to see what to do

If u just simply log that message it is of waste

Just add a callback method

public static void sendAllMessagesToSamePartition(Properties p){  
 KafkaProducer<String,String> producer=new KafkaProducer<String, String>(p);  
 producer.initTransactions();  
 producer.beginTransaction();  
 System.*out*.println("txn began started writing all messages");  
 AtomicInteger counter=new AtomicInteger(0);  
 Callback callback=(recMetadata,exception)->{  
 if (exception == null) {  
 int callbackExecutedCount= counter.getAndIncrement();  
 System.*out*.printf("\n %d. msg went to topic --> %s partition --> %d , offset --> %s ",  
 callbackExecutedCount,recMetadata.topic(),recMetadata.partition(),recMetadata.offset());  
 }  
 };  
 IntStream.*range*(1,6000).forEach(num->{  
 *//Here we are sending all messages to partition number 1, so all msgs will be sent to p1  
 // and key is unique for every message  
 // and as we are already giving partition num,so it will not decide the partition based on the key* ProducerRecord<String,String> record=  
 new ProducerRecord<>("EmployeesInfo","orey naina--"+num);  
 *// this callback method will be executed each and every time for each and every message* producer.send(record,callback);  
 });  
*//even though u didnt committed all messages will be written to broker,  
// once u committed it will change the flag to committed for each and every message* producer.commitTransaction();  
 try {  
 Thread.*sleep*(4000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 System.*out*.println("sent all 6000 messages and committed too and got acks for "+counter.get());  
 }

As you attached a callback method , if it is a successfully received by the broker producer will get back the acknowledgement

When producer got ack the callback method will be fired automatically

In the above if you see , I have sent 6000 messages continuously to the broker and attached a callback and mostly considering success , 6000 times the callback method will be executed

Sample log:-

sent all 6000 messages and committed too and got acks for 5999

Refer file name

“Send10KMessagesWithAcksCallback.java”

QA’s

1. What is use of producer id-
2. to identify who send this message to the broker   
   b) And while doing a transaction general even though u didn't commit, messages are written to topic in abort mode with producer id , if u didn't commit or if u abort the transaction, these messages will be in abort mode itself and those messages should be deleted right??? So broker will delete all the messages with that producer id + transaction id

if u dont have the producer id , whose messages will be deleted, so its mandatory

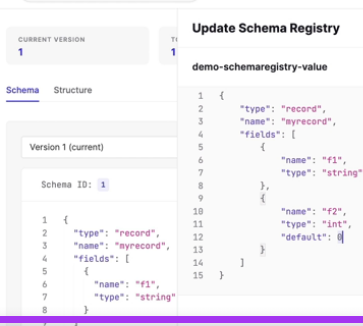
and make sure every producer instance should have separate transaction id

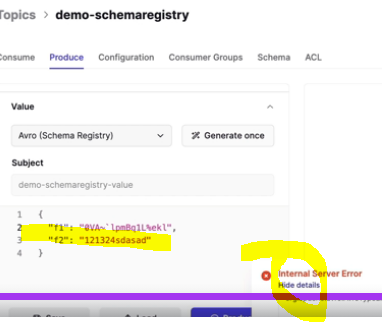
Data formats-avro,json

Json data- if u serialize this both key and value will be serialized and it will be larger message hence it is bit slow, if u use json for exchanging data, schema information will also be sent

Avro- it transmits data in binary and message size will be small and hence data will travel very fast, so prefer avro over json

If u put that avro schema in schema registry, it imposes strict schema ,and u cant send any other data that is not compliant with that schema

if requirements changes, then u should update schema registry values

Here if u see, u are sending string,string, but as per above schema registry we should send String,int since there is a mismatch producer itself will fail to send the data to the topic

Simple theories

1. ProducerRecord<String,String> record=  
    new ProducerRecord<>("EmployeesInfo","orey naina--"+num);

While creating producerRecord if we send the partition number , that record straight away goes to that partition , if u miss it will decide the partition number based on the key’s hash value , if key also not there then value will be sent to all partitions in round robin fashion

Methods in kafka producer

kafkaProducer.flush()

generally if u do kafkaTemplate.send() it will send those records to producer buffer

and IO threads are responsible to take the records from the buffer and send them to broker

flush() if u use flush method , it will immediately send all the records present in buffer

KafkaProducerRecord.send() it will send the data to buffer

KafkaProducer.flush() it will send all the records in buffer to broker immediately

Producer properties

Properties.set(“batch.size”,400); if u set this //I don’t know but try it

I think for that pa

For Every partition, producer will maintain producer local cache and will have individual batch size

Means this is batch size for each partition

properties.setProperty(“partitioner.class”,roundRobinPartioner.class); means all records will be sent to all partitions // this is no recommended in production because all messages will be sent equally to all partitions no batching extra overhead of sending each record individually

**Safe producer settings**

Since kafka 3.0 producer is safe by default, here ack is acknowledgement

Calculate size of 1 message, how many messages will come for a minute so based on all these factors set these below properties

|  |
| --- |
| 1. Create a topic with min.insync.replicas=2 so that always 2 brokers must be in sync leader+follower, if these 2 are not in sync then even if producer tries to send a msg then that will not accept because not even single follower is in sync hence it will throw an exception |
| 1. Set these producer properties , in general 1st data will be sent only to buffer from there IO threads will pickup the data and sends to broker, it will not send msg 1 by 1 it will store in buffer and it will send all the data at once   **Linger.ms** =10,000 () instead of sending message 1 by 1 IO thread will wait for this linger.ms 10 sec and sends all the data present in buffer to broker  **Batch.size** = size of partition buffer batch 100KB instead of waiting above seconds , u can specify batch.size if this is met in 2 sec then IO threads will compress and send all data at once  **Buffer.memory** = 300 KB this is the total buffer memory for producer to store the records in all partition buffer’s =(we have buffer for each and every partition, this is the memory included for all partitions) – note if batch size =20KB then buffer size must be more than that  If broker is down or if he is consuming at slower rate, then buffer size will be filled up easily  **block.on.buffer.full**= true (means producer program will be blocked until buffer is free, false means message will be discarded & producer.send() will throw an exception )  So calculate the incoming message size and configure these settings accordingly ,  Now producer is be blocked, but how long it should block???????????? It is until max.block.ms  **Max.block.ms** = 100s now producer will wait for this much time before dropping the message / throwing an exception, because producer cant block forever right |
| 1. Acks= all means broker after receiving message , all those ISR – insync replicas also should receive the message else broker wont send the ack , |
| 1. Request.timeout.ms = 500ms means producer will wait for this many seconds for ack from broker before resending the same message, (ee time lopala broker nundi ack ravali else message will be resent )   if broker didn’t send an ack before this time then producer will resend the same message again   * If retries are enabled (retries>0) then if producer didn’t received ack for this much time from broker then message will be resent immediately |
| 1. Retries= 100 means at max producer will retry sending same message 100 times if the broker is not at all sending ack even after 100 times then message is considered as failed |
| 1. retries.backoff.ms – defines the delay between 2 consecutive retries (it will give the pause between retries so that broker recovers) by default 100ms and we should retry only for recoverable exceptions   **Higher value**- giving the broker time to recover and potentially reducing unnecessary retries.  **Lower value:** Might lead to rapid retries if set too low, putting strain on the broker and impacting overall performance. |
| 1. delivery.timeout.ms is the total time for retry and wait time b/n 2 consecutive retries |
| 1. Enable.Idempotentence=true, if leader broker received and consumer also consumed but if ISR didn’t received producer will same message so this is called duplicate message   And if u want to avoid this thenuse this idempotence - it will internally uses producer id +sequence id |
|  |

Most important settings

**Linger.ms, Batch.size, acks = all, min.insync.replicas=2, retries = Integer.MAX\_VALUE**

**Enable.idempotence = true m buffer.memory**

Acks -- This configuration instructs the producer to consider a write successful

1. delivery.timeout.ms=120000 means until this time is reached producer will keep on retrying

above says stop retrying the same message after 2 mins (120000ms=120 seconds=2mins)

once 2 mins is reached don’t retry anymore

1. Acknowledgements between producer and broker

Consider acks=all for strong message durability so that all the leaders + ISR will definitely receive and written to log file

producer property acks=0, acks=1, acks=all (acks=-1)

props.put(ProducerConfig.ACKS\_CONFIG, "all");

this property must be used along with min.insync.replicas=2

1. acks=0 zero means producer wont wait for broker acknowledgement he will send the msg into producer buffer and will blindly consider a message received by broker

he will leave he won't even consider whether broker is online or offline , he won't consider even if broker is not able to accept the message or not just it will put the message into buffer and leaves this is a worst approach because no message guarantee if u are ok to loose messages then prefer this approach,  
acks= 1 means leader broker will send back ack to producer when leader broker alone received message successfully and written to log , leader will not consider any ack from its followers

, here also there are chances of data loss , because when leader went down if replicas are out of sync , then its wrong to make that out of sync replica as leader as it is very far

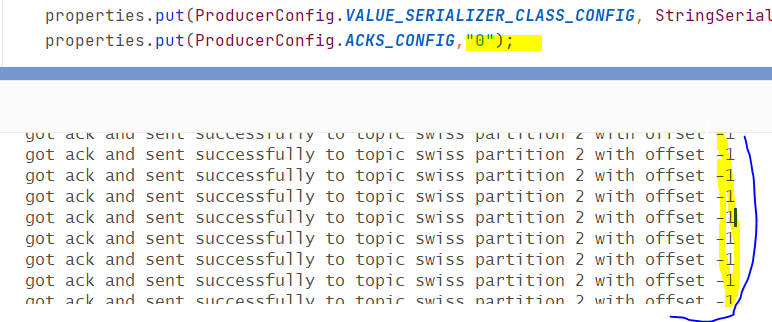
one among those replicas became leader as these followers are pretty far from leader they may not have all those messages as they are out of sync, hence this is also 100% no reliable , this is reliable only as long as there all replicas are in sync

acks=all (acks=-1)means broker will give back acknowledgement to producer only when leader & all followers / insync replicas also received it,

if any ISR didn’t received the message the broker won’t give the acks to producer & in that case producer will retry this can lead to duplicate message delivery   
10) this property must be used asking with min.insync.replicas=1/2...acks=all or acks=-1, means producer will wait for leader+follower broker , broker will send back acks only when all leader and followers has received and acknowledged that message this is the default property in kafka 3.0 thia acks=all will go have in hand with min.insyn.replicas=2 if min in sync replicas=0 its waste of giving acks=all , means ur saying give ack only when all follower broker also recived and if u give min.insync.replicas=0, then no followers are there it will give ack when leader broker rxd it, so make sure when u are giving ack=all u should give mi.insync.replicas=2../3

**Illegal combination 🡪 acks=all min.insync.replicas=1** this is waste we configured acks=all means we kept as broker should give ack only when all leader+ISR’s received that message, whereas as

min.insync.replicas=1 as & when leader receives msg he will acknowledge, so even though we gave **acks=all it internally works as acks=1**



See here when we gave acks=0, then producer will just put into buffer & it wont wait for ack from broker, hence it will get offset number as -1 for all records

1. Min.insync.replicas this is the broker property

. It's a topic configuration that affects data durability and availability. Which we should give while creating the topic

This is a most recommended setting when u apply setting acks=all means broker will give acks to the producer only when all the IN-SYNC-REPLICAS also received the message ,

so when u kept acks=all, so to make sure min.insync.replicas=2 if u keep this property broker will always ensure minimum 2 broker are there in the ISR list , so that if leader is down any follower can become leader ,

if while creating topic if we give min.insync.replicas=2, but if the replicas are too much far behind/lag the broker then the producer can’t even send the message (but good thing is even in that time consumers can happily consume ) -- if producer tried to send a message at this time then broker will this throw exception saying not enough replicas 🡪 until replicas came back producer can't send message due to min.insync.replicas =2 constraint, producers that attempt to send data will receive NotEnoughReplicasException

if replication factor=3 and min.insync.replicas=3 then we dont tolerate any broker down, if any1 broker is down producer can't send the message so best option is if replication factor is 3 min.insync.replicas=2(3-1) atmost we can tolerate 1 broker down.

**Recommended is Min.insync.replicas=2** (1 leader + 1 follower (backup leader) ) – in general all the team members need not to be in sync with team leader, atleast 1 must be a backup for team lead

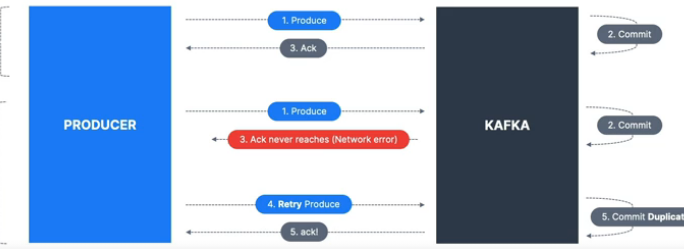
min.insync.replicas=2: the topic must have at least 2 ISR up, and therefore we can tolerate at most one broker being down (in the case of replication factor of 3), and we have the guarantee that for every write, the data will be at least written twice, now leader will provide ack back to producer only when 2 brokers (leader+ 1 ISR) also received the message

This means with a replication factor of 3, min.insync.replicas should be 2.

1. Idempotent producer

producerProps.put(“enable.idempotence”,true); this is to avoid duplicates especially in case of retries this s is exactly once scenario

this will ensure no duplicates, if u keep this even if producer retries by sending the same message broker can automatically identifies it as a duplicate and it can stop consuming it and it can simply send ack

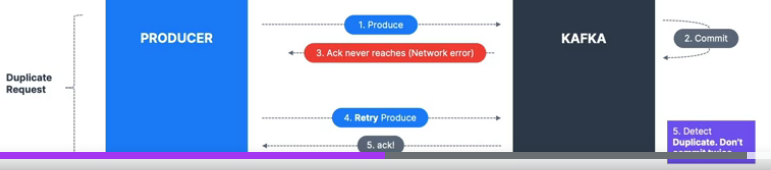


Incase of retry, producer can send duplicates

In above example, in 2d scenario broker successfully received the msg and he gave acks whereas the acks didn’t reached producer and producer has retried and broker received the same message again,

this is the duplicate message scenario to avoid this they introduced

Producer idempotence



When u enable the idempotence producer while sending the message producer will send along with some sequence number and next time if same msg came broker can easily identify whether the message with this sequence number has already came or not. If broker notices as duplicate this time he will not consume and whereas he will simply acknowledge it without consuming it as it has already been consumed

1. Max inflight requests

max.in.flight.requests.per.connection=5(default) means maximum number of requests for which producer can wait for ack before resending the same message

, it tells how many messages a producer can send to broker without waiting for ack from the broker.

for example for every message sent by producer he must get ack , else he will retry sending same message, bec of this property maximum for this many number of number of request producer will wait for ack

Risk is – ordering, if more unack messages are there means then prob of un-ordering will be there

max.in.flight.requests.per.connection allows the producer to send more messages concurrently, potentially without waiting for ack from broker improving throughput. However, this comes at the cost of increased risk of out-of-order deliver

1. Request timeout

**request.timeout.ms –** This is the maximum time producer will wait for receiving an ack from broker after sending a message

if producer didn’t receive ack from broker within this time then producer will resend the same message

(metadata.fetch.timeout.ms) – this tells how long producer should wait for metadata ack from broker before attempting a retry

Timeout.ms – it is the time leader broker will wait for acks from ISR to confirm the message acceptance, after this leader will send an error saying message is not consumed by ISR

1. Retrying until timeout

Represents number of times producer will retry sending the messages until he gets an acknowledgement from broker

in case when acks=all leader will not send ack if any one of ISR didn’t copy that message from leader, then since producer didn’t received ack the producer will again resend the same message

retries=233434 means ,until this many times producer will retry ,whereas we can configure time also

retries.backoff.ms ~ pause time b/n 2 consecutive retries

this is a producer property retry.backoff.ms=200 seconds means producer should wait 200 ms before he retries next time

delivery.timeout.ms=120000 =( request.timeout.ms ~ max time to wait for ack+ retries.backoff.ms)

delivery.timeout.ms

means until this time(here 2 minutes) is reached producer will keep on retrying , after 2 minutes also if producer is unable to deliver the message to the broker then producer will stop retrying as timeout 2 mins (120000ms=120 seconds=2mins) reached

1. Batch size and linger.ms
2. Configure linger.ms= which is bus max waiting time properties.put(ProducerConfig.LINGER\_MS\_CONFIG,1000); //
3. Configure batch size, if before bus max waiting time if all persons on boarded into the bus, bus can start immediately, so before linger.ms if all batch size filled then records will be sent immediately, generally producer responsibility is just to push to producer buffer at producer side

Its IO threads responsibility to send the data from producer buffer to broker

properties.put(ProducerConfig.BATCH\_SIZE\_CONFIG,Integer.valueOf(32\*1024));32KB //says if this many persons came in the bus bus will start immediately

1. Configure the compression mechanism -:- so that all the batched records will be compressed

properties.put(ProducerConfig.*COMPRESSION\_TYPE\_CONFIG*,"snappy");

1. Choose the sticky partitioner, so that instead of round robin partitioner ,if u choose this all records will be in the same producer buffer partition cache,mentioned batch.size records will filled fastly before linger.ms , so batching and compression will be done with maximum number of records

Linger means staying there more than the required time

Its same like bus max waiting time is 30 mins, if bus filled fully within 20 mins , bus will start immediately

This is a producer property “linger.ms =9000” and says how long to the records will be in producer cache until 9 sec all these will be batched and will not be sent to the broker , here all the records will be in producer cache /batch for 9 seconds , but if the before 9 seconds if the batch is filled then that batch will be sent to the broker, by 9 seconds whether single record came also bus will start immediately, until that much time all the records might stay in the producer batch , if u give linger.ms=2mins then until 2 minutes all those records will stay in the batch only,but meanwhile if batch.size=1000 if all those 1000 records came within 1/2sec all those will be sent to broker immediately, its like for every 2 seconds bus will start whether people came or not

Set producer config “batch.size= 26KB” means max 26KB records and this should be used in conjunction with [linger.ms](http://linger.ms/) if the batch filled means if 2000 records came in 2 seconds batch will be sent immediately increasing the batch size increases the compression

Any message bigger than the batch size will not be batched

1. Buffer.memory MAX.BLOCK.MS and buffer.size in bytes

Buffer.memory = 300 KB this is the total buffer memory for producer to store the records in all partition buffer’s =(we have buffer for each and every partition, this is the memory included for all partitions) – note if batch size =20KB then buffer size must be more than that

If broker is down or if he is consuming at slower rate then our producer buffer will keep on increasing until it is full, once it is full and if it is producer is still having more data then if we set

block.on.buffer.full= true (means producer will be blocked until buffer is free, false means message will be discarded & producer.send() will throw an exception )

here the producer will be blocked until max.block.ms – this is the maximum time that producer will be in blocked state after this producer either discard message or it will throw exception

Producer will be in blocked state in below scenarios

1. Producer buffer fully filled- always producer will send the data to buffer, but if that buffer is filled then producer will be in blocked state due to property (block.on.buffer.full= true)
2. Metadata request - The producer might also block on max.block.ms while fetching metadata about partitions (e.g., using partitionsFor() method) before sending messages. This ensures the producer has up-to-date information about the topic and partitions

So calculate the incoming message size and configure these settings accordingly ,

Now producer will be blocked ,but how long it should block???????????? It is until max.block.ms

Problem of less buffer size

Lets say if that time broker is down or if broker is slowly accepting the records ,assume data is coming at rate of 1Million records per second, but here if producer buffer is small of 1MB,buffer cant store the 1M records and for that solution is increase the buffer size ,

If broker is not accepting any messages

Ok even after increasing the buffer size ,if broker is not accepting any message the messages still will be there in buffer what if again buffer size got filled, then ProducerRecord.send() this call is blocked means until the buffer is cleared it will be blocked for max.block.ms=10\*1000 seconds

If there is any space is buffer data will be pushed to buffer ex:- same like airplane which can take only few people to destination at a time so that airplane is a buffer

If no space in buffer, but data is coming keep on and if broker is already down, then

|  |  |
| --- | --- |
| If block.on.buffer.full =true – then program will be blocked and will wait until buffer is free | If block.on.buffer.full = false means it wont block it will simply discard that message it won’t send that message to broker |

But program will not be blocked for ever, so it producer will wait for max.block.ms time after that either it will throw an exception or block that message

Even after 10 seconds if the buffer is still full and if data is coming into producer buffer then exceptions will come

Exceptions are thrown when

1. Producer buffer is filled
2. Max.block.ms=6\*1000 seconds reached and still buffer is fill and data is still coming and producer.send() is blocking
3. Or even if broker is not accepting any data

If u hit these exceptions

1. Max.request.size vs message.max.bytes

|  |  |  |
| --- | --- | --- |
| Max.request.size (producer side config) default 1MB | message.max.bytes (broker side configuration) default 1MB | Batch.size(producer level config)  Prefer only this batch.size instead of max.request.size which is confusing for me |
| This defines maximum size of a single request ( it can have one or more records)  This controls batching of messages before sending it to broker  Note:- this max.request.size should be more than message.max.bytes then only more than 1 message will be included in the batch | As name indicate it is the maximum size of a single message  Ex:- if any message is more than 1MB exception will be thrown | This defines the max size of a batch of records that will be accumulated before sending to broker, once either batch.size is met or if linger.ms is met which ever is earlier  Then messages are sent as a batch with compression  The producer will attempt to create batches up to the batch.size limit, but the final size of the request is still constrained by max.request.size  If max.request.size is 1MB and batch.size is 512KB, the producer will try to accumulate messages up to 512KB before sending them |

1. Compression at producer level

<https://blog.cloudflare.com/squeezing-the-firehose/> By default, messages are sent uncompressed (Message compression is disabled by default)

For best compression techniques configure  
compression.type=snappy (this snappy is invented by google) or lz4

properties.put(ProducerConfig.*COMPRESSION\_TYPE\_CONFIG*,"snappy");

Producer while sending the message we can send in the form of batches, for best speed, increase the batch size and perform compression, so compressed messages will be sent

By enabling compression, you reduce network utilization and stor‐ age, which is often a bottleneck when sending messages to Kafka.

**Advantages of compression**

Much smaller request size –message will be compressed 4x times smaller, 1MB to 256 KB

Since size reduced data will be transferred fast and compressed messages will be stored in the partition and files which leads to better utilization of disk

Only disadvantages is

Some cpu is needed to compress the data and since compressed messages are stored in broker files at consumer side also we need to decompress and extract it

**Compression level set at broker side**

There is also a setting you can set at the broker level (all topics) or topic-level

- **compression.type=producer** (default), the broker takes the compressed batch

from the producer client and writes it directly to the topic's log file without recompressing the data

-**compression.type=none**: means broker is against to compression now all batches are decompressed by the broker

**compression.type=1z4**: (for example)

• If it's matching the producer setting, data is stored on disk as is

. If it's a different compression setting, batches are decompressed by the broker and then re- compressed using the compression algorithm specified

For a high throughput producer

1. Increase linger.ms- means producer will wait for the batches to fill up before sending them to the broker, if u give linger.ms=9\*1000 means until 9 sec records will be there in producer buffer only,meanwhile if batch.size=10\*1024 (10KB records )came in to the buffer then by 8 sec it self IO thread can send the data from buffer to broker
2. Introduce producer level compression
3. Sticky partitioner

properties.setProperty(“partitioner.class”,StickyPartioner.class);

there are many advantages and always prefer Sticky partitioner over Roundrobin partitioner for best performance

Generally producer will maintain buffer for each and every partition at producer side after linger.ms is reached or if before linger.ms=9sec itself batch.size=2KB records came then batch of that records will be sent from producer buffer to broker partitions

If u choose sticky partitioner, all the records will be placed in the same producer partition buffer cache (*so in this case, key based partitioning wont happen ,because if u choose key based partitioning all the records may go into multiple producer partition buffer which is against to the sticky partitioner*) then before linger.ms=9000 ms if batch of records came then compression can be applied on all these records and can be sent immediately whereas if u choose roundRobin partitioner for linger.ms=9000ms /9sec only a few records will come to each partition because of round robin nature records will be split across multiple producer buffers and records count in each producer buffer will be decreased hence batching will be done only few records which is having less performance than doing compression on multiple records so to do compression on multiple records prefer sticky partitioner

then it means brokers are down or brokers are not responding to the requests

1. receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the OS defaults will be used. It is a Conguring Producers | 51 good idea to increase those when producers or consumers communicate with brokers in a different datacenter because those network links typically have higher latency and lower bandwidth.

1. fgh