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

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

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

Producer

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 ap these messages are coming from, so set this ID while configuring the producer

Sample producer code

Purpose of producer id is to track the source of message

Properties props = new Properties();

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());

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

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);

}

|  |
| --- |
| 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 will be decided)** |
| **Partition – (if u give partition , msg straight away it will goto that partition only)** |
| **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 to proper compression

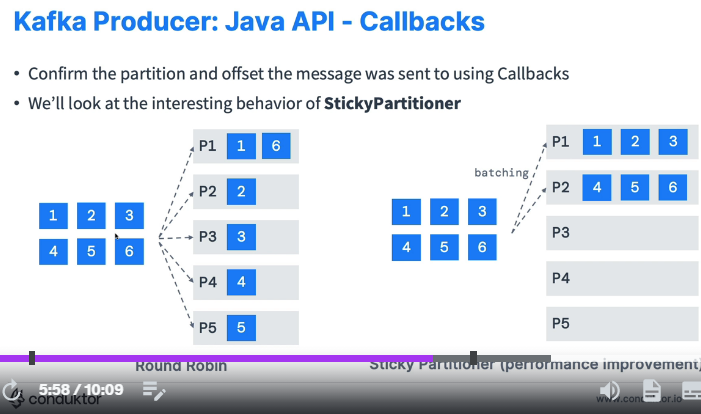
**Sticky partitioner**

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 batches in producer 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

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

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** 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 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

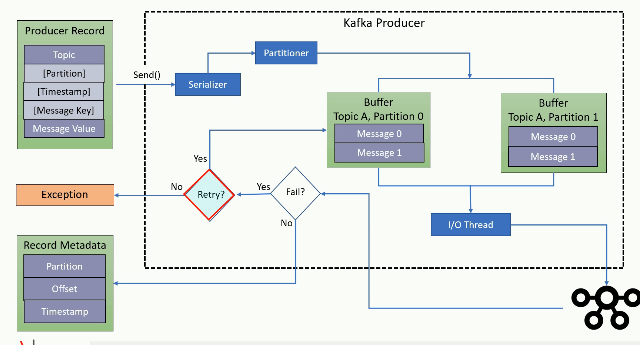
Message.timestamp.type=1

Producer buffer

kafkaTemplate.send(topicName,message) // this will actually send the message to the buffer partition which resides in the producer memory, it will not immediately send the message 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



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

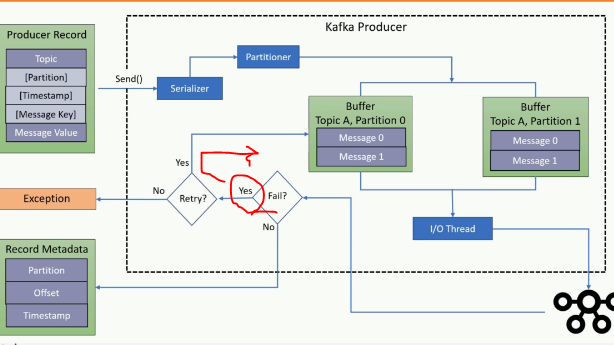
Doubt:- let’s say if buffer max capacity is 1MB if we send 1-2 messages definitely it wont fill , in that case to full the capacity it take hours, but how are we getting instantly and why are we not waiting till buffer is filled, ex:- in our 1DSTR app if u send 1 message having bytes it is immediately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

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

Producer retries

Generally if we send 1 million messages we should get acknowledgements for each message we received, lets say if we have configured acks=all means leader will send positive acknowledgements only when all followers / all ISR’s received it, lets 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



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

Duplicate message –atleast once vs atmost once

Below are the scenarios of coming duplicate messages

1. 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 replications – 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.

To avoid this w have 1 worst solution that is configure zero retries= atmost Once

So this is bad because even if something went wrong it wont retry leads to data loss

**Best solution is u implement the duplicate record identification** mechanism based on the co relation id

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 problem??and how it works internally

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

So broker when that message successfully received the message it will store that id+sequence number and if producer sends the same message again with same producer instance id +offset id the broker will reject

This solution won’t work if application itself is wantedly sending the duplicate message

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

Enable.idempotence=true

Transactions in Kafka

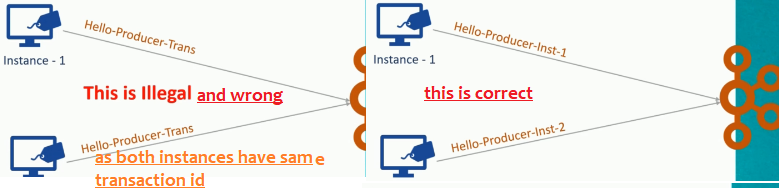
1. Transaction **means – either all or nothing** - It says either all messages within the transaction are committed or nothing is committed
2. Transactions should be committed if all parts are succeeded ,
3. only committed data can be consumed by the consumer, if u have sent to broker and if u didn’t 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
4. 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 u don’t give if all producers instances are writing with same id and if all are using transactions 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 cant 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

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

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

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?

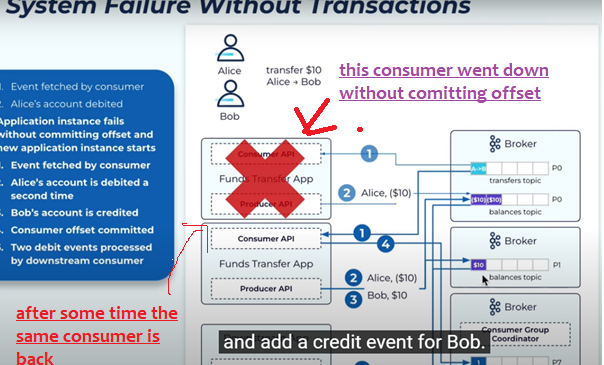
Hence each producer instance should have different transaction id

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

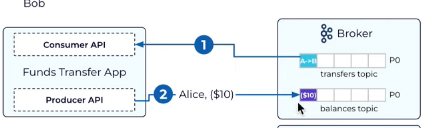
Producer.commitTRansaction(); producer.abortTransaction();

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

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

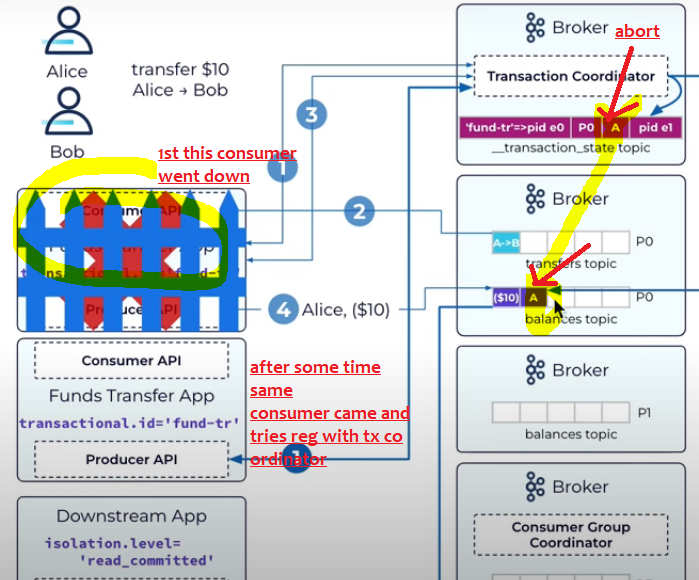
And after sometime it is back and since last time that consumer didn’t commit the offset it will read the same transfer message again **from Alice 🡪 Bob and** puts **Alice money debit request message** into the topic second time which is wrong () and keeps money credit message to Bob in another topic

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*.**

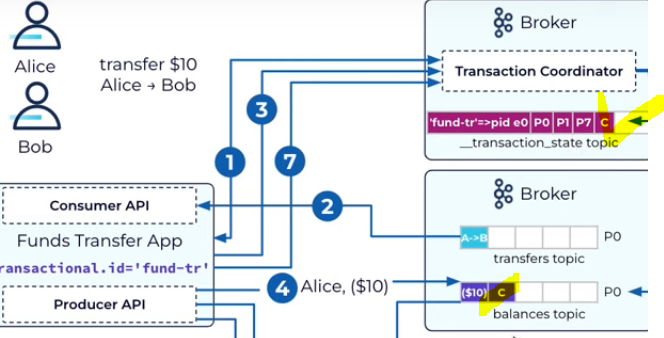
Q) U may get a doubt if these messages are already written what if they are consumed in between ,

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

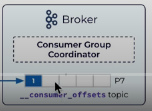
If success and when you **commit the transaction**, then commit marker will be marked on that message see below message, if transaction is aborted then message will be **marked as abort** refer above



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

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

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

1. Acks=all(means broker will give acks to producer only when he and all ISR’s received the message)
2. Min.insync.replicas=3/any value (minimum 2 isrs must be present total 3 with broker) this is a broker or topic level setting

The above both must be used combinely else there wont be any meaning to acks=all

1. enable.idempotence=true –to avoid duplicate message 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

1. retries=MAX\_INT if u keep this producer will retry endless times until the delivery.timeout.ms is reached
2. 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. max.in.flight.requests.per.connection=5
2. Acknowledgements between producer and broker

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

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

acks=0 zero means producer wont wait for broker acknowledgement he will send the msg into broker and 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 this is a worst approach because no message guarantee if u are ok to loose messages then prefer this approach,  
acks= 1 means producer will wait for leader acknowledge alone, only leader broker not follower broker, here also there are chances of data loss , because when leader went down if replicas are out of sync , when 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 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  
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

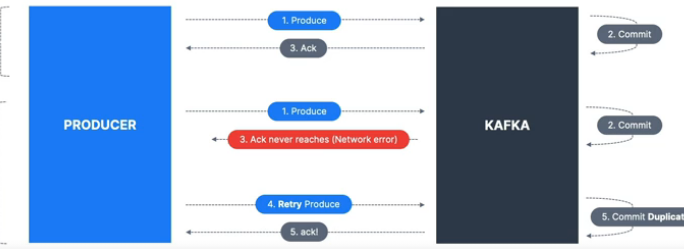
1. Min.insync.replicas this is the broker property

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 , lets say if no replica is in sync with the leader then as per acks=all , since nobody is there in the ISR list, broker will give back ack immediately as no one is present in the ISR list, so when u kept acks=all, so to make sure some one is present in the ISR list 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 ,

min.insync.replicas=2 means 1 broker +1 follower when u give this at any time uf there are no enough replicas if producer set the config as min.insync.replicas=2and if there are no 2 replicas then if producer tried to send it will this throw exception saying not enough replicas until replicas came back producer can't send message due to min.insync.replicas =2 constraint, but good thing is too that time consumers can happily consume  
13) 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

1. Idempotent producer

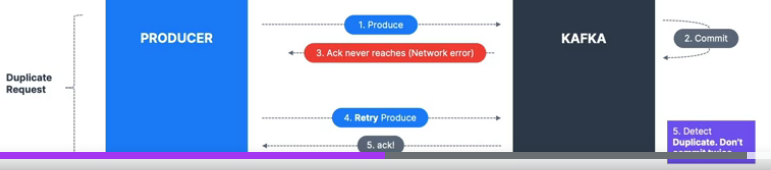
producerProps.put(“enable.idempotence”,true); this is to avoid duplicates especially in case of retries



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 means at a time 5 batches will be in flight(data movement between producer and broker) generally for each and every partition of a topic we will have producer caches at the producer side ,generally until linger.ms is reached or batch.size num of records reached records will be in that cache then once linger.ms is crossed of batch size filled then all the batches data will be send and those batches count is 5

1. Retrying until timeout

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

delivery.timeout.ms=120000 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

Retrying with delay

retry.backoff.ms=1000 means, producer will wait for 1000ms before each retry

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

1. Batch size and linger.ms

properties.put(ProducerConfig.LINGER\_MS\_CONFIG,1000);

properties.put(ProducerConfig.BATCH\_SIZE\_CONFIG,Integer.valueOf(32\*1024));32KB

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, 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 all those will be sent to broker immediately

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. Compression at producer level

<https://blog.cloudflare.com/squeezing-the-firehose/>

For best compression techniques configure  
compression.type=snappy 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

**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
2. Introduce producer level compression