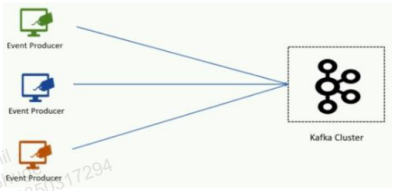
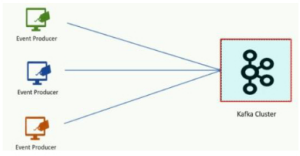
Producer Scalability

Apache Kafka was designed with the scalability. If we consider the POS example, then each POS system can create a KafkaProducer object and send the invoices. The below shows the scenario where multiple POS systems can send invoices in parallel.



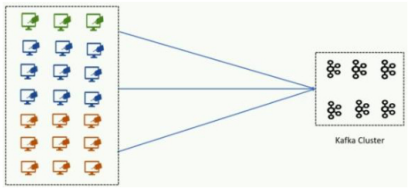
At the cluster end, it is the Kafka broker that receives the messages and acknowledges the successful receipt of the message.



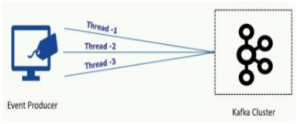
So, if we have hundreds of producers sending messages in parallel, we may want to increase the number of brokers in our Kafka cluster as shown above.

A single Kafka broker can handle hundreds of messages per second. However, we can increase the number of Kafka brokers in our cluster and support hundreds of thousands of messages to be received and acknowledged.

On the producer side, we can keep adding the new producers to send the messages to the Kafka server in parallel. This arrangement provides linear scalability by simply adding more producers and brokers.

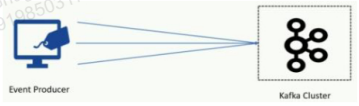


This approach works perfectly for, scaling up our over all streaming band width. However we can also scale an individual producer, using multi threading technique.



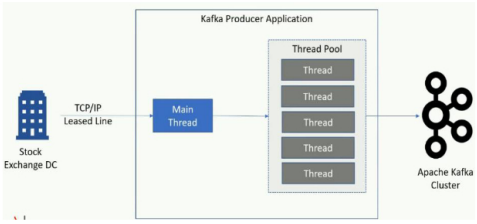
A single producer thread is good enough to support the use cases where the data is being produced at a reasonable pace. However, some scenarios may require parallelism at the individual producer level as well. We can handle such requirements using multi threaded Kafka producer.

The multithreading scenario may not apply to applications that do not frequently generate new messages. For example, an individual POS application would be producing an invoice every 2-3 minutes. In that case, a single thread is more than enough to send the invoices to the Kafka cluster.



However, if we have an application that generates or receives data at high speed and wants to send it as quickly as possible, we might want to implement a multithreaded application. Let's try to understand the need for multithreading with a realistic example.

Lets assume a stock market data provider application,



- The application receives tick by tick stock data packets from the stock exchange over a TCP/IP socket. The data packets are arriving at high frequency. So, we decided to create a multithreaded

data handler.

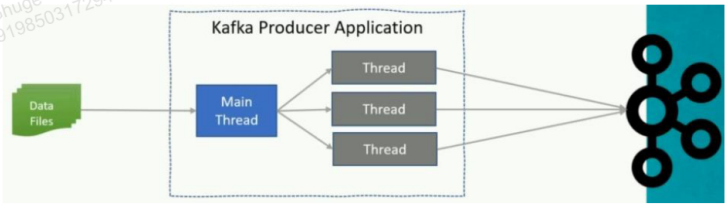
- The main thread listens to the Socket and reads the data packet as they arrive. The main thread immediately handovers the data packet to a different thread for sending the data to the Kafka broker and starts reading the next data packet.
- The other threads of the application are responsible for uncompressing the packet, reading individual messages from the data packet, validating the message and sending it further to the Kafka broker.

Apache Kafka

- Similar scenarios are common in many other applications where the data arrives at high speed, and we may need multiple application threads to handle the load. Kafka producer is thread-safe.
- So, our application can share the same producer object across multiple threads and send messages in parallel using the same producer instance.
- It is not recommended to create numerous producer objects within the same application instance. Sharing the same producer across the threads will be faster and less resource intensive compared to instantiating a separate producer in each thread.

Problem Statement

Assume we have multiple data files. We want to create an application that takes a list of data files as an argument and does following.



1. Instantiate a Kafka Producer
2. Create one independent thread to process each data file. For example, if we are supplying three data files to the application, it must create three threads, one for each data file.
3. Start independent threads with one data file for each thread and share the same Kafka Producer Instance among all threads.
4. Each thread is responsible for reading records from one file and sending them to Kafka broker in parallel.
5. Wait for all threads to complete processing and finally, close the application.

Solution –Producer Threads

The solution to the given problem can be implemented in two parts.

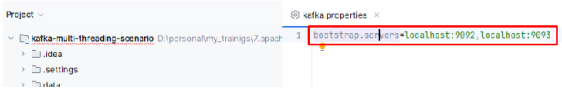
1. Create a reusable message dispatcher that can send messages to Kafka.
2. Create a main application that starts multiple threads using the dispatcher.

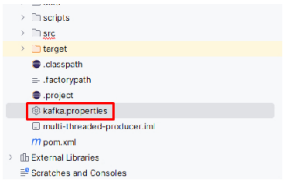
Apache Kafka

Configure the Kafka Clients dependency in pom.xml as shown below:

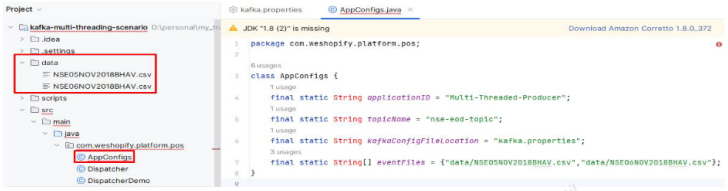
```
https://maven.apache.org/xsd/maven-4.0.0.xsd (xsi:schemaLocation)
<project xmlns "http://maven.apache.org/POM/4.0.0" xmlns:xsi "http://www.w3.org/200
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.durgasoft</groupId>
  <artifactId>kafka-producer-multi-threading</artifactId>
  <version>0.0.1 SNAPSHOT</version>

  <properties>
      <kafkaVersion>3.4.0</kafkaVersion>
      <lombokVersion>1.18.26</lombokVersion>
  </properties>
  <dependencies>
      <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients -->
      <dependency>
          <groupId>org.apache.kafka</groupId>
          <artifactId>kafka-clients</artifactId>
          <version>${kafkaVersion}</version>
      </dependency>

      <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
      <dependency>
          <groupId>org.projectlombok</groupId>
          <artifactId>lombok</artifactId>
          <version>${lombokVersion}</version>
          <scope>provided</scope>
      </dependency>
  </dependencies>
</project>
```

1. Create a Kafka.propeties file as shown below:





2. Create AppConfig java file as shown below:



Step-1: Creating a Dispatcher


```
1. public class Dispatcher implements Runnable {
2.
3.     private final KafkaProducer<Integer, String> producer;
4.     private final String topicName;
5.     private final String fileLocation;
6.     private static final Logger logger = LogManager.getLogger(Dispatcher.class);
7.
8.     Dispatcher(KafkaProducer<Integer, String> producer, String topicName, String fileLocation) {
9.         this.producer = producer;
10.        this.topicName = topicName;
11.        this.fileLocation = fileLocation;
12.    }
13.
14.    @Override
15.    public void run() {
16.        logger.info("Start processing " + fileLocation + "...");
17.        File file = new File(fileLocation);
18.        int msgKey = 0;
19.
20.        try (Scanner scanner = new Scanner(file)) {
21.            while (scanner.hasNextLine()) {
22.                String line = scanner.nextLine();
23.                producer.send(new ProducerRecord<>(topicName, msgKey, line));
24.                msgKey++;
25.            }
26.            logger.trace("Finished sending " + msgKey + " messages from " + fileLocation);
27.        } catch (Exception e) {
28.            logger.error("Exception in thread " + fileLocation);
29.            throw new RuntimeException(e);
30.        }
31.    }
32. }
```

Line-1: To create a producer thread, we are implementing Java Runnable Interface to define a class named as Dispatcher. The Dispatcher class implements the Runnable interface. The Runnable interface allows us to execute an instance of this class by a Thread.

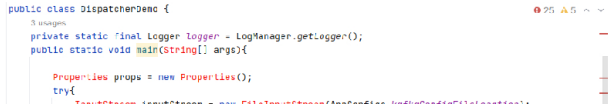
Line-8: The constructor takes a KafkaProducer object as an argument. It also takes a topic name as well as a data file location.

Line-15-to-31: Then we implement a run() method that scans each line of the file and send each line to the Kafka cluster using the producer.send() call.

Step-2:Creating Main

The next part of the puzzle is to implement a main() method to create threads and start them. The main() method is implemented in the **DispatcherDemo** class as shown below:

3. Write DispatcherDemo.java file as shown below:



```
        props.load(inputStream);
        props.put(ProducerConfig.CLIENT_ID_CONFIG,AppConfigs.applicationID);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    }catch (IOException e){
        throw new RuntimeException(e);
    }

    KafkaProducer<Integer,String> producer = new KafkaProducer<>(props);
    Thread[] dispatchers = new Thread[AppConfigs.eventFiles.length];
    logger.info("Starting Dispatcher threads...");
    for(int i=0;i<AppConfigs.eventFiles.length;i++){
        dispatchers[i]=new Thread(new Dispatcher(producer,AppConfigs.topicName,AppConfigs.eventFiles[i]))
        dispatchers[i].start();
    }
```

```
1. public static void main(String[] args) {
2.     if (args.length < 2) {
3.         System.out.println("Please provide command line arguments: topicName EventFiles");
4.         System.exit(-1);
5.     }
6.
7.     String topicName = args[0];
8.     String[] eventFiles = Arrays.copyOfRange(args, 1, args.length);
9.
10.    Properties properties = new Properties();
11.    try {
12.        InputStream configStream = ClassLoader.class.getResourceAsStream(kafkaConfig);
13.        properties.load(configStream);
14.        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
15.                ntegerSerializer.class.getName());
```



```
16.            properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
17.                    StringSerializer.class.getName());
18.        } catch (IOException e) {
19.            logger.error("Cannot open Kafka config " + kafkaConfig);
20.            throw new RuntimeException(e);
21.        }
22.
23.        logger.trace("Starting dispatcher threads...");
24.        KafkaProducer<Integer, String> producer = new KafkaProducer<>(properties);
25.
26.        Thread[] dispatchers = new Thread[eventFiles.length];
27.        for (int i = 0; i < eventFiles.length; i++) {
28.            dispatchers[i] = new Thread(new Dispatcher(producer, topicName, eventFiles[i]));
29.            dispatchers[i].start();
30.        }
31.
32.        try {
33.            for (Thread t : dispatchers)
34.                t.join();
35.        } catch (InterruptedException e) {
36.            logger.error("Thread Interrupted ");
37.        } finally {
38.            producer.close();
39.            logger.info("Finished dispatcher demo - Closing Kafka Producer.");
40.        }
41.    }
42. }
```

Line-1-to- 9 :

The main method takes a topic name and a list of data files as command line arguments. We have included two sample data files in the GitHub repository. Let's assume that we execute the program and supply those two data files in the argument list.

Line-10-to-24: Rest of the code is straightforward. We define Properties object and set basic producer configurations. Then we instantiate a KafkaProducer using those configurations.

Line-26-to-30: Now, all we need to do is to create two Dispatcher threads and pass one file to each thread. We also share the same KafkaProducer instance to both the Dispatcher threads. Those two threads will execute and send the data from their respective files to the Kafka cluster. However, both threads will be sharing the same producer instance.

They are not creating their own producer instances, and that's what is recommended. We should share the same producer across threads.

Line-32-to-40: Rest of the code is simple. We wait for all the threads to complete and finally, close the producer to free up the resources.

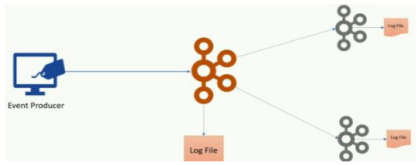
In this example, we also tried to demonstrate an essential idea of loading the Kafka producer configurations from a file. At line 12 and13 in the Code, we load some producer settings that are kept outside the code.

This simple technique allows us to make our application configurable and change the configurations later. For example, when we are deploying our application in a different environment, we can change the broker list in the configuration file, and the same code works in the new environment.

At Least Once Vs At Most Once

Apache Kafka provides messages durability guarantee, by committing the messages, at the partion log. The durability simply means, once the data is persisted by the leader broker in the leader partition we cant loose that message till the leader is alive.

However if the leader broker goes down we may loose the data. To protect the loss of records due to the leader failure, kafka implemented replication using followers.

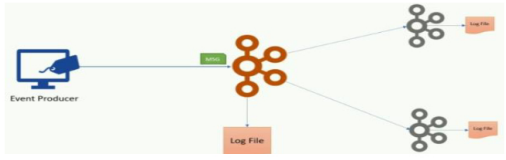


The followers will copy the messages from the leader, and provides fault tolerance in case of leader failures. In the other words, when the data persisted to the leaders as well as to the followers in the ISR list, we considered the messages as to be fully committed.

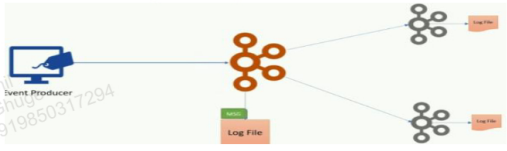
Once the message is fully committed, we cant loose the record until the leader, and all the replicas are lost which is unlikely case, however in all these cases, we still have a possibility of committing the duplicate messages due to the producer retry mechanism.

If the producer I/O thread fails to get a success acknowledgement from the broker, it will retry the same message. Assume that I/O thread transmits the message to the kafka broker

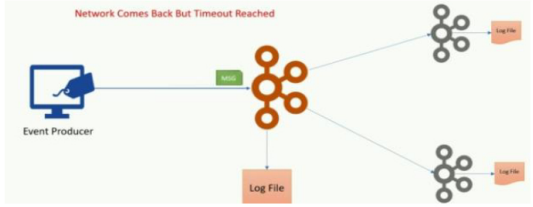
Apache Kafka



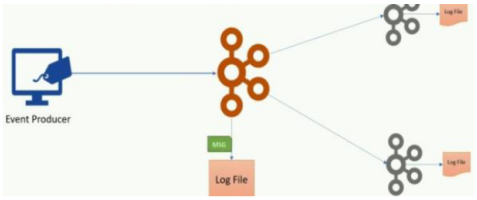
And it stores into the partition log.



The broker then sends the successful acknowledgement and the response doesn't reach the I/O thread, due to a network error, in this case the producer I/O thread will wait for the acknowledgment, and ultimately send the record again by assuming it as failure and then broker again receives the data



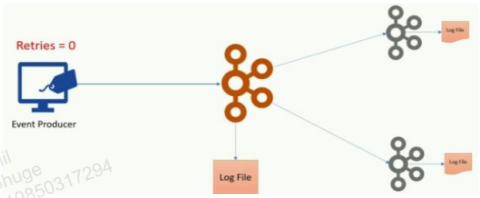
But it doesn't have a mechanism to identify that the message is an duplicate of an earlier message. Hence the broker saves the duplicate record causing a duplication problem.



This implementation is known as "**At least once"** semantics, where we cant loose the messages because we are retrying until we get the success acknowledgment. However we may have duplicates because we

do not have a method to identify a duplicate message. For that reason, Kafka said to provide "At least once" semantics.

Kafka also allows us to implement at most once semantics by configuring the retries to zero. In that case, we may loose some records, but we will never have a duplicate record committed to kafka logs



Exactly Once

If don't loose anything and at the same if we don't create any duplicates, then we should go for exactly once semantics. To meet exactly once requirement, Kafka offers an idempotent producer configuration. All we need to do is to enable idempotent and kafka takes care of implementing exactly once.

To enable idempotent, we should set the enable.idempotance producer configuration to true.

```
enable.idempotence=true
```

Once we configure the idempotence, the behaviour of the produce API will change and will do two things.

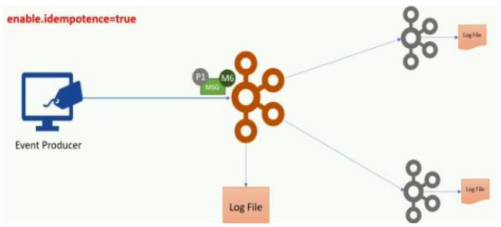
1. Internal ID for Producer Instance
2. Message Sequence Number

1. Internal ID for Producer Instance

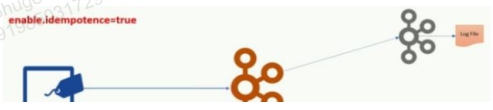
It will perform an initial hand-shake with the leader broker and asks for a unique producer ID. At the broker side, the broker dynamically assign the unique ID to each producer. The next that happens is the message sequencing.

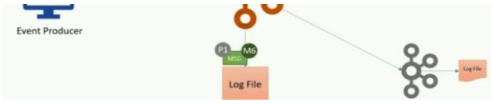
2. Message Sequence Number

The producer API will start assigning a sequence number to each message. This sequence numbers starts from zero and monotonically increments per partition. Now when the I/O thread sends the messages to the leader broker, the message is uniquely identified by the producer id and the sequence number



Now the broker knows that the last committed message sequnce number is "x" and the next expected message sequnce number is "x+1"





This allows the broker to identify the duplicates as well as missing sequence numbers. So settings "**enable.idempotence=true**" will helps us to ensure that the messages are neither lost nor duplicated, this will takes care by the producer API and the broker internally all the thing that we need to do is to just set the property to activate this behaviour.

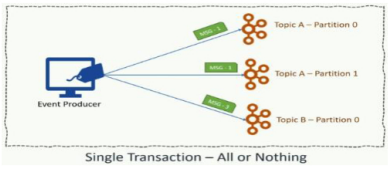
However we must have to remember one thing, if we are sending the duplicate messages, at our application level, this configuration cant protect us from the duplicates. That should be considered as a bug in our application. Even if two different threads or two producer instances, are sending duplicates, that too even an application design problem.

Idempotance is only guaranteed for the producer retries and you should not try to resend the message, at the application level. Idempotance is not guaranteed for the application level resend or duplicates send by the application it self.

Transnational Producer

The transactional producer goes one step ahead of idempotent producer and provides the transnational guarantee, i.e. an ability to write to several partitions atomically. The atomicity has the same meaning as in databases, that means, either all messages within the same transaction are committed, or none of them are persisted.

Apache Kafka



Single Transaction – All or Nothing

We can implement a transactional producer using the steps shown in the Code below:

Step-1: create two topics as shown below:

topic-1-create
topic-2-create

Step-2:

All topics which are included in a transaction should be configured with the

replication factor >=3 and

min.insync.replicas >=2

```
%KAFKA_HOME%\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --topic hello-producer-2 --partitions 5
--replication-factor 3 --config min.insync.replicas=2
```

%KAFKA_HOME%\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --topic hello-producer-2 --partitions 5 --replication-factor 3 --config min.insync.replicas=2

Step-3: Configure the AppConfig.java with the no.Of events count as 2 just to quickly verify the records.

```
class AppConfigs {
    final static String applicationID = "HelloProducer";
    final static String bootstrapServers = "localhost:9092,localhost:9093";
    final static String topicName1 = "hello-producer-1";
    final static String topicName2 = "hello-producer-2";
    final static int numEvents = 2;
    final static String transaction_id = "Hello-Producer-Trans";
}
```

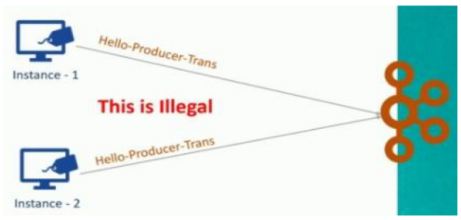
Step-4: HelloProducer.java

Apache Kafka

```
public class HelloProducer {
    8 usages
    private static final Logger logger = LogManager.getLogger();
```

```
    public static void main(String[] args) {

        logger.info("Creating Kafka Producer...");
        Properties props = new Properties();
        props.put(ProducerConfig.CLIENT_ID_CONFIG, AppConfigs.applicationID);
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, AppConfigs.bootstrapServers);
        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, AppConfigs.transaction_id);

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

- Here when we set the transaction id at line no.24, idempotance automatically is enabled. Because transactions are dependent on idempotance.
- Transactional_id_config must be unique for each producer instance that means we cant run two instances of producers with the same transactional_id, if we do so then one of those transactions will be aborted. Because two instances of the same transaction are illegal. The primary purpose of the transaction id is to roll back the older unfinished transactions for the same transaction id, Incase of producer application bounces or restarts.




```
        logger.info("Starting First Transaction...");
        producer.beginTransaction();
        try {
            for (int i = 1; i <= AppConfigs.numEvents; i++) {
                producer.send(new ProducerRecord<>(AppConfigs.topicName1, i,  value: "Simple Message-T1-" + i));
                producer.send(new ProducerRecord<>(AppConfigs.topicName2, i,  value: "Simple Message-T1-" + i));
            }
            logger.info("Committing First Transaction.");
            producer.commitTransaction();
        }catch (Exception e){
            logger.error("Exception in First Transaction. Aborting...");
            producer.abortTransaction();
            producer.close();
            throw new RuntimeException(e);
        }

        logger.info("Starting Second Transaction...");
        producer.beginTransaction();
        try {
            for (int i = 1; i <= AppConfigs.numEvents; i++) {
                producer.send(new ProducerRecord<>(AppConfigs.topicName1, i,  value: "Simple Message-T2-" + i));
                producer.send(new ProducerRecord<>(AppConfigs.topicName2, i,  value: "Simple Message-T2-" + i));
            }
            logger.info("Aborting Second Transaction.");
            producer.abortTransaction();
        }catch (Exception e){
            logger.error("Exception in Second Transaction. Aborting...");
            producer.abortTransaction();
            producer.close();
            throw new RuntimeException(e);
        }

        logger.info("Finished - Closing Kafka Producer.");
        producer.close();

    }
}
```

- We must wrap all our send() API calls within a pair of beginTransaction() and commitTransaction(). In case we receive an exception that we can't recover from, abort the transaction and finally close the producer instance.
- All messages sent between the beginTransaction() and commitTransaction()will be part of a single transaction.
- One last important point to note here, The same producer cannot have multiple open transactions. We must commit or abort the transaction before we can begin a new one.
- The commitTransaction() will flush any unsent records before committing the transaction. If any of the send calls failed with an irrecoverable error, the commitTransaction() call will also fail and throw the exception from the last failed API. That means, we can safely implement an asynchronous send and should not even implement a call back method because a successful commitTransaction() indicates that all messages are sent successfully and acknowledged.
- In case a message is failed, commitTransaction() will anyway throw and exception. When the commitTransaction() throws an exception, our application should abortTransaction() to reset the

state.

Apache Kafka

- The producer is thread-safe. So, we can call the send() API from multiple threads. However, we must call the beginTransaction() before starting those threads and either commit or abort when all the threads are complete.