**Kafka Python Tutorial for Fast Data Architecture**

In this Kafka Python tutorial, we will create a Python application that will publish data to a Kafka topic and another app that will consume the messages.

**Fast Data Series Articles**

1 [Installing Apache Mesos 1.6.0 on Ubuntu 18.04](https://dzone.com/articles/installing-apache-mesos-160-on-ubuntu-1804)

2 [Kafka Tutorial for Fast Data Architecture](http://www.admintome.com/blog/kafka-tutorial-for-fast-data-architecture/)

3 Kafka Python Tutorial for Fast Data Architecture

This is the third article in my Fast Data Architecture series that walks you through implementing Bid Data using a SMACK Stack. This article builds on the others so if you have not read through those, I highly suggest you do so that you have the infrastructure you need to follow along in this tutorial.

This article will walk you though pulling website metrics from Clicky.com. I have another article where we will pull metrics from Google Analytics and publish the metrics to Apache Kafka: [Kafka Python and Google Analytics](https://dzone.com/articles/kafka-python-and-google-analytics).

In order to demonstrate how to analyze your big data, we will be configuring a big data pipeline that will pull site metrics from Clicky.com and push those metrics to a Kafka topic on our Kafka Cluster.

This is just one pipeline that you might want to implement in your Big Data Implementation. Website statistics can be a valuable part of your data as this can give you data about web site visitors, pages visited, etc. Combine this data with other data like social media shares when you perform your data analytics and you would be able to make some pretty neat business decisions about when is the best time for you to post site updates to social media in order to attract the most visitors. That is the main benefit of implementing big data: not necessarily the raw data itself but the knowledge you can extract from that raw data and make more informed decisions.

In this example, we will pull the '**pages**' statistics from the Clicky.com API and push them to the **admintome-pages** Kafka topic. This will give us JSON data from AdminTome's top pages.

**Clicky Web Analytics**

In order to fully follow along in this article, you will need to have a website linked to Clicky.com. It's free so why not. Register your site at [clicky.com](http://clicky.com/101045340). I personally use it because it has better metrics reporting for blogs (like abandon rate) than Google Analytics gives. You will need to add some code to your page so that clicky can start collecting metrics.

After your page is sending metrics to clicky you will need to get some values in order to use the Clicky API and pull metrics from our Python application. Go to preferences for your site and you will see two numbers that we will need:

• Site ID

• Site key

Don't publish these anywhere because they could give anyone access to your website data. We will need these numbers later when we connect to the API and pull our site statistics.

**Preparing Kafka**

First, we need to prepare our Kafka Cluster by adding a topic to our Kafka cluster that we will use to send messages to. As you can see from the diagram above, our topic in Kafka is going to be admintome-pages.

Login to the Mesos Master you ran Kafka-mesos from. If you followed the previous article, the master I used was mesos1.admintome.lab. Next, we will create the topic using the kafka-mesos.sh script:

$ cd kafka/

$ ./kafka-mesos.sh topic add admintome-pages --broker=0 --api=http://mslave2.admintome.lab:7000

Notice that the API parameter points to the Kafka scheduler we created using kafka-mesos in the last article. You can verify that you now have the correct topics:

$ ./kafka-mesos.sh topic list --api=http://mslave2.admintome.lab:7000

topics:

name: \_\_consumer\_offsets

partitions: 0:[0], 1:[0], 2:[0], 3:[0], 4:[0], 5:[0], 6:[0], 7:[0], 8:[0], 9:[0], 10:[0], 11:[0], 12:[0], 13:[0], 14:[0], 15:[0], 16:[0], 17:[0], 18:[0], 19:[0], 20:[0], 21:[0], 22:[0], 23:[0], 24:[0], 25:[0], 26:[0], 27:[0], 28:[0], 29:[0], 30:[0], 31:[0], 32:[0], 33:[0], 34:[0], 35:[0], 36:[0], 37:[0], 38:[0], 39:[0], 40:[0], 41:[0], 42:[0], 43:[0], 44:[0], 45:[0], 46:[0], 47:[0], 48:[0], 49:[0]

options: segment.bytes=104857600,cleanup.policy=compact,compression.type=producer

name: admintome

partitions: 0:[0]

name: admintome-pages

partitions: 0:[0]

And there is our new topic ready to go! Now it's time to get to the fun stuff and start developing our Python application.

Now that we have Kafka ready to go we will start to develop our Kafka producer. The producer will get page metrics from the Clicky API and push those metrics in JSON form to our topic that we created earlier.

I assume that you have Python 3 installed on your system and virtualenv installed as well.

To get started we will need to setup our environment.

$ mkdir ~/Development/python/venvs

$ mkdir ~/Development/python/site-stats-intake

$ cd ~/Development/python/site-stats-intake

$ virtualenv ../venvs/intake

$ source ../venvs/intake/bin/activate

(intake) $ pip install kafka-python requests

(intake) $ pip freeze > requirements.txt

Next, we need to create our classes.

**Clicky Class**

We will create a new Python class called Clicky that we will use to interact with the Clicky API. Create a new file called **clicky.py** and add the following content:

import requests

import json

class Clicky(object):

def \_\_init\_\_(self, site\_id, sitekey):

self.site\_id = site\_id

self.sitekey = sitekey

self.output = "json"

def get\_data(self, data\_type):

click\_api\_url = "https://api.clicky.com/api/stats/4"

payload = {"site\_id": self.site\_id,

"sitekey": self.sitekey,

"type": data\_type,

"output": self.output}

response = requests.get(click\_api\_url, params=payload)

raw\_stats = response.text

return raw\_stats

def get\_pages\_data(self):

data = self.get\_data("pages")

return json.loads(data)

Save the file and exit.

In order to get our metrics, we need to send an HTTP GET request to the Clicky API URL which is

https://api.clicky.com/api/stats/4

We also need to include several parameters:

**• site\_id**: This is the Site ID number that we got earlier.

**• sitekey**: This is the Site key number that also got earlier.

**• type**: To get our top pages we set the type to 'pages.'

**• output**: We set this to "json" so that the API will return JSON data.

Finally, we call the request Python module to perform an HTTP GET to our API URL with the parameters we specified. In the get\_pages\_data method, we return a dict that represents our JSON data. Next, we will code our Kafka class implementation.

**MyKafka Class**

This class will interact with our Kafka cluster and push website metrics to our topic for us. Create a new file called **mykafka.py** and add the following content:

from kafka import KafkaProducer

import json

class MyKafka(object):

def \_\_init\_\_(self, kafka\_brokers):

self.producer = KafkaProducer(

value\_serializer=lambda v: json.dumps(v).encode('utf-8'),

bootstrap\_servers=kafka\_brokers

)

def send\_page\_data(self, json\_data):

self.producer.send('admintome-pages', json\_data)

First, we import the kafka-python library, specifically the [KafkaProducer](https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html) class, that will let us code a Kafka producer and publish messages to our Kafka Topic.

from kafka import KafkaProducer

We now define our MyKafka class and create the constructor for it:

class MyKafka(object):

def \_\_init\_\_(self, kafka\_brokers):

This takes an argument that represents the Kafka brokers that will be used to connect to our Kafka cluster. This an array of strings in the form of:

[ "broker:ip", "broker:ip" ]

We will use only one broker where is the one we created in the last article: mslave1.admintome.lab:31000:

[ "mslave1.admintome.lab:31000" ]

We next instantiate a new KafkaProducer object named producer. Since we will be sending data to Kafka in the form of JSON we tell the KafkaProducer to use the JSON decoder dumps to parse the data using the value\_serializer parameter. We also tell it to use our brokers with thebootstrap\_servers parameter.

self.producer = KafkaProducer(

value\_serializer=lambda v: json.dumps(v).encode('utf-8'),

bootstrap\_servers=kafka\_brokers

)

Finally, we create a new method that we will use to send the messages to our admintome-pages topic:

def send\_page\_data(self, json\_data):

self.producer.send('admintome-pages', json\_data)

That's all there is to it. Now we will write our Main class that will control everything.

**Main Class**

Create a new file called **main.py** and add the following content:

from clicky import Clicky

from mykafka import MyKafka

import logging

import time

import os

from logging.config import dictConfig

class Main(object):

def \_\_init\_\_(self):

if 'KAFKA\_BROKERS' in os.environ:

kafka\_brokers = os.environ['KAFKA\_BROKERS'].split(',')

else:

raise ValueError('KAFKA\_BROKERS environment variable not set')

if 'SITE\_ID' in os.environ:

self.site\_id = os.environ['SITE\_ID']

else:

raise ValueError('SITE\_ID environment variable not set')

if 'SITEKEY' in os.environ:

self.sitekey = os.environ['SITEKEY']

else:

raise ValueError('SITEKEY environment variable not set')

logging\_config = dict(

version=1,

formatters={

'f': {'format':

'%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}

},

handlers={

'h': {'class': 'logging.StreamHandler',

'formatter': 'f',

'level': logging.DEBUG}

},

root={

'handlers': ['h'],

'level': logging.DEBUG,

},

)

self.logger = logging.getLogger()

dictConfig(logging\_config)

self.logger.info("Initializing Kafka Producer")

self.logger.info("KAFKA\_BROKERS={0}".format(kafka\_brokers))

self.mykafka = MyKafka(kafka\_brokers)

def init\_clicky(self):

self.clicky = Clicky(self.site\_id, self.sitekey)

self.logger.info("Clicky Stats Polling Initialized")

def run(self):

self.init\_clicky()

starttime = time.time()

while True:

data = self.clicky.get\_pages\_data()

self.logger.info("Successfully polled Clicky pages data")

self.mykafka.send\_page\_data(data)

self.logger.info("Published page data to Kafka")

time.sleep(300.0 - ((time.time() - starttime) % 300.0))

if \_\_name\_\_ == "\_\_main\_\_":

logging.info("Initializing Clicky Stats Polling")

main = Main()

main.run()

The end state of this example is to build a Docker container that we will then run on Marathon. With that in mind, we don't want to hardcode some of our sensitive information (like our clicky site id and site key) in our code. We want to be able to pull those from environment variables. If they are not set then we through an exception and exit out.

if 'KAFKA\_BROKERS' in os.environ:

kafka\_brokers = os.environ['KAFKA\_BROKERS'].split(',')

else:

raise ValueError('KAFKA\_BROKERS environment variable not set')

if 'SITE\_ID' in os.environ:

self.site\_id = os.environ['SITE\_ID']

else:

raise ValueError('SITE\_ID environment variable not set')

if 'SITEKEY' in os.environ:

self.sitekey = os.environ['SITEKEY']

else:

raise ValueError('SITEKEY environment variable not set')

We also configure logging so that we can see what is going on with our application. I have coded an infinite loop in our code that will poll clicky and push the metrics to our Kafka topic every five minutes.

def run(self):

self.init\_clicky()

starttime = time.time()

while True:

data = self.clicky.get\_pages\_data()

self.logger.info("Successfully polled Clicky pages data")

self.mykafka.send\_page\_data(data)

self.logger.info("Published page data to Kafka")

time.sleep(300.0 - ((time.time() - starttime) % 300.0))

Save the file and exit.

**Running Our Application**

To test that everything works you can try running the application after you set your environment variables:

(intake) $ export KAFKA\_BROKERS="mslave1.admintome.lab:31000"

(intake) $ export SITE\_ID="{your site id}"

(intake) $ export SITEKEY="{your sitekey}"

(intake) $ python main.py

2018-06-25 15:34:32,259 root INFO Initializing Kafka Producer

2018-06-25 15:34:32,259 root INFO KAFKA\_BROKERS=['mslave1.admintome.lab:31000']

2018-06-25 15:34:32,374 root INFO Clicky Stats Polling Initialized

2018-06-25 15:34:32,754 root INFO Successfully polled Clicky pages data

2018-06-25 15:34:32,755 root INFO Published page data to Kafka

We are now sending messages to our Kafka Topic! We will build our Docker container next and deploy it to Marathon. Finally, we will wrap up by writing a test consumer that will get our messages from our topic.

I have created a GitHub repository for all the code used in this article: <https://github.com/admintome/clicky-state-intake>

Now that we have our application code written, we can create a Docker container so that we can deploy it to Marathon. Create a Dockerfile file in your application directory with the following contents:

FROM python:3

WORKDIR /usr/src/app

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD [ "python", "./main.py" ]

Build the container

$ docker build -t {your docker hub username}site-stats-intake .

After the Docker build is completed, you will want to push it to your Docker repository that your Mesos Slaves have access to. For me, this is Docker Hub:

$ docker push -t admintome/site-stats-intake

Then log in to each of your Mesos slaves and pull the image down:

$ docker pull admintome/site-stats-intake

We are now ready to create a Marathon application deployment for our application.

Go to your Marathon GUI.

http://mesos1.admintome.lab:8080

Click on the Create Application Button. Then click the JSON mode button:

Paste in the following JSON code:

{

"id": "site-stats-intake",

"cmd": null,

"cpus": 1,

"mem": 128,

"disk": 0,

"instances": 1,

"container": {

"docker": {

"image": "admintome/site-stats-intake"

},

"type": "DOCKER"

},

"networks": [

{

"mode": "host"

}

],

"env": {

"KAFKA\_BROKERS": "192.168.1.x:port",

"SITE\_ID": "{your site\_id}",

"SITEKEY": "{your sitekey}"

}

}

Be sure to substitute the correct values for KAFKA\_BROKERS, SITE\_ID, and SITEKEY in the **env** section for your environment.

Finally, click on the Create Application button to deploy the application. After a few seconds, you should see the application is Running.

To see the logs click on the site-stats-intake application then click on the stderr link to download a text file containing the logs.

Now that we have our application deployed to Marathon we will write a short consumer that we will run on our development system to show us what messages have been received.

This will be a simple Kafka consumer that will check out the topic and display all messages on the topic. Not really useful at this point but it lets us know that our little polling application is working correctly.

Create a new file called **consumer.py** and add the following contents:

import sys

from kafka import KafkaConsumer

consumer = KafkaConsumer('admintome-pages', bootstrap\_servers="mslave1.admintome.lab:31000",

auto\_offset\_reset='earliest')

try:

for message in consumer:

print(message.value)

except KeyboardInterrupt:

sys.exit()

Save and exit the file. This has the Kafka broker hardcoded because we simply are using it to test everything. Make sure to update the bootstrap-servers parameter with your broker name and port.

Now run the command and you should see a ton of JSON that represents your most visited pages:

(intake) $ python consumer.py

b'[{"type": "pages", "dates": [{"date": "2018-06-25", "items": [{"value": "145", "value\_percent": "43.2", "title": "Kafka Tutorial for Fast Data Architecture - AdminTome Blog", "stats\_url": "http://clicky.com/stats/visitors?site\_id=101045340&date=2018-06-25&href=%2Fblog%2Fkafka-tutorial-for-fast-data-architecture%2F", "url": "http://www.admintome.com/blog/kafka-tutorial-for-fast-data-architecture/"},...

We now have a data pipeline that has some data that we can use. The next step will be to use that data and analyze it. In the article, we will install and configure the next part of our SMACK stack which is Apache Spark. We will also configure it analyze our data and give us something meaningful.

============================================

**Kafka Python and Google Analytics**

Learn to use Kafka Python to pull Google Analytics data and push it to Kafka, and analyze this data later using Spark to derive meaningful data.

Learn how to use Kafka Python to pull Google Analytics metrics and push them to your Kafka Topic. This will allow us to analyze this data later using Spark to give us meaningful business data.

Google Analytics is a very powerful platform for monitoring your web site’s metrics including top pages, visitors, bounce rate, etc. As more and more businesses start using Big Data processes, the need to compile as much data as possible becomes more advantageous. The more data you have available, the more options you have to analyze that data and produce some very interesting results that can help you shape your business.

This article assumes that you already have a running Kafka cluster. If you don’t, then please follow my article [Kafka Tutorial for Fast Data Architecture](http://www.admintome.com/blog/kafka-tutorial-for-fast-data-architecture/) to get a cluster up and running. You will also need to have a topic already created for publishing the Google Analytics metrics to. The aforementioned article covers this procedure as well. I created a topic called admintome-ga-pages since we will be collection Google Analytics (ga) metrics about my blog’s pages.

In this article, I will walk you through how to pull metrics data from Google Analytics for your site then take that data and push it to your Kafka Cluster. In a later article, we will cover how to take that data that is consumed in Kafka and analyze it to give us some meaningful business data.

This is just one single pipeline of data that we are going to analyze. You could include many, many more pipelines based off of the process that we will be covering, so keep this post handy as you continue your journey in Big Data.

The first thing that we need to do is enable the Google Analytics API.

**The Google Analytics API**

The Google Analytics API is actually many APIs that let you interact with Google Analytics. There are APIs to build applications on top of the functionality of Google Analytics all the way to real-time reporting (which honestly sounds pretty awesome and might be a future article topic). In this article, we will focus on the [Core Reporting API](https://developers.google.com/analytics/devguides/reporting/core/v4/) which will give us a list of pages for the last 7 days and how many people visited that page.

**Enabling the Core Reporting API**

The first step we need to take is enabling the API. Open up the [Service Accounts page](https://console.developers.google.com/iam-admin/serviceaccounts). Select Create a project and click on Continue.

Next, click on **Create Service Account**.

Then click on **Go to credentials**.

You will see the **Add credentials to your project** screen. We want to select **Google Analytics Reporting API** for ‘Which API are you using.’ For, ‘Where will you be calling the API from’ select **Other non-UI** because we are going to write a Python Daemon that will use the API to pull the metrics. Select **Application Data** for ‘What data will you be accessing.’ Lastly, answer **No** for the ‘Are you planning to use this API with App Engine or Compute Engine’ since our Python application will run on our own cluster. Click **Continue** to move on to the next section.

Now on to where we create the service account.

Under **Service account name**, give your service account a meaningful name. I used kafka-intake as the name for my service account. Under the **Role** drop-down, select Service Account then Service Account User. Next, select **JSON** as the key type. Finally, click on **continue**. The key will be downloaded to your system. Take note of the Service Account ID field. You will need this in the next step.

**Add Your Service Account to the Google Analytics Account**

Go to your Google Analytics page and sign in. Go to **Settings** and click on **User Management**. Click on the blue plus sign in the upper right corner to add a new user then click on **Add new users**. You will see the add new user dialog:

When you are done, click on the blue Add button at the top right-hand corner. You now have the Google Analytics API enabled and you are ready to securely pull metrics from it. In the next section, we will walk through writing our Python application that will pull this data and push it to Kafka.

**Kafka Python Example**

Now that we got through the boring part, it is finally time to roll up our sleeves and start some development! If you just want the code then check out my [GitHub repository](https://github.com/admintome/analytics-intake).

We need to create a Python environment for our application by running the following commands (may differ for Mac and Windows users).

$ mkdir analytics-intake

$ cd analytics-intake/

$ virtualenv ../venvs/analytics-intake

$ source ../venvs/analytics-intake/bin/activate

This assumes that you have a **venvs** directory above your application directory (analytics-intake for this example). Next, we need to install some required Python modules.

$ pip install kafka-python

$ pip install --update google-api-python-client

$ pip install --upgrade oauth2client

Next, freeze your requirements to a text file.

$ pip freeze > requirements.txt

Time to write our main class and get things started.

**AnalyticsIntake.py**

Our first file is called AnalyticsIntake.py and is our main script. The first thing we will need is our imports:

from googleapiclient.discovery import build

from oauth2client.service\_account import ServiceAccountCredentials

from mykafka import MyKafka

import logging

from logging.config import dictConfig

import time

import os

from datetime import datetime, timedelta

The Google Analytics API Python Client needs to have the first two imports. Next, we need to define our class and our member variables:

class AnalyticsIntake(object):

SCOPES = ['https://www.googleapis.com/auth/analytics.readonly']

KEY\_FILE\_LOCATION = None

VIEW\_ID = None

kafka\_brokers = None

topic = None

delay = 3600

In order to call the Google Analytics API, we need several variables set. The first one is the SCOPES variable which is the API URL that our code will be calling. Next is the KEY-FILE\_LOCATION variable which is a string representing the Credentials Key that was downloaded earlier when we enabled the Google Analytics API. The VIEW\_ID variable can be found by using the [Account Explorer](https://ga-dev-tools.appspot.com/account-explorer/) for Google Analytics. The rest of the variables we will cover when we talk about the Kafka code.

Now we need to write our constructor for the AnalyticsIntake class.

def \_\_init\_\_(self):

self.init\_logging()

if 'KAFKA\_BROKERS' in os.environ:

self.kafka\_brokers = os.environ['KAFKA\_BROKERS'].split(',')

self.logger.info(

"Set KAFKA\_BROKERS: {}".format(self.kafka\_brokers))

else:

raise ValueError('KAFKA\_BROKERS environment variable not set')

if 'KEY\_FILE' in os.environ:

self.KEY\_FILE\_LOCATION = os.environ['KEY\_FILE']

self.logger.info("Set KEY\_FILE: {}".format(self.KEY\_FILE\_LOCATION))

else:

raise ValueError('KEY\_FILE environment variable not set')

if 'VIEW\_ID' in os.environ:

self.VIEW\_ID = os.environ['VIEW\_ID']

self.logger.info("Set VIEW\_ID: {}".format(self.VIEW\_ID))

else:

raise ValueError('VIEW\_ID environment variable not set')

if 'TOPIC' in os.environ:

self.topic = os.environ['TOPIC']

self.logger.info("Set TOPIC: {}".format(self.topic))

else:

raise ValueError('TOPIC environment variable not set')

if 'DELAY' in os.environ:

self.delay = os.environ['DELAY']

self.logger.info("Set DELAY: {} s".format(self.delay))

else:

self.delay = 3600

self.logger.info(

"DELAY environment variable not set - Setting to default {} s".format(self.delay))

Here, we are pulling the values for all these variables from environment variables. This is because we eventually want to put this code into a Docker container and we don’t want to put our key file in our Docker container. Your key file should always be kept secret. If any of these environment variables are not set, then we throw a ValueError exception and exit the application. Only one of these variables has a default and that is the delay variable. This is the number of seconds to wait between polls to Google Analytics. The default is 3600 seconds, which is equivalent to one hour.

We are going to want to use logging, so lets set up a member function to initialize logging.

def init\_logging(self):

logging\_config = dict(

version=1,

formatters={

'f': {'format':

'%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}

},

handlers={

'h': {'class': 'logging.StreamHandler',

'formatter': 'f',

'level': logging.INFO}

},

root={

'handlers': ['h'],

'level': logging.INFO,

},

)

self.logger = logging.getLogger()

logging.getLogger("googleapiclient").setLevel(logging.ERROR)

dictConfig(logging\_config)

Notice that we have a line at the end to set logging for the  googleapiclient to ERROR. This is to prevent some warning from popping up in our output logs.

Now we will create a member function to initialize our Google Analytics Reporting.

def init\_reporting(self):

creds = ServiceAccountCredentials.from\_json\_keyfile\_name(

self.KEY\_FILE\_LOCATION, self.SCOPES)

analytics = build('analyticsreporting', 'v4', credentials=creds)

return analytics

This creates a new [ServiceAccountCredentials](http://oauth2client.readthedocs.io/en/latest/source/oauth2client.service_account.html) object that uses our KEY\_FILE\_LOCATION and SCOPES variables. We then use the [googleapiclient.discovery](https://google.github.io/google-api-python-client/docs/epy/googleapiclient.discovery-module.html) object’s build class to set up our Analytics API passing in our credentials that we just created.

Next, we will write a get\_reports member function that we will use to pull the metrics that we want.

def get\_reports(self, analytics):

return analytics.reports().batchGet(

body={

'reportRequests': [

{

'viewId': self.VIEW\_ID,

'dateRanges': [{'startDate': '7daysAgo', 'endDate': 'today'}],

'metrics': [{'expression': 'ga:sessions'}],

'dimensions': [{'name': 'ga:pageTitle'}]

}]

}

).execute()

There is a lot to this code so I will try to cover a lot of it. When you pull metrics from Google Analytics you need to specify the metrics that you want to pull. In this case, we are pulling ga:sessions metrics. The ga:sessions metric gives us the total number of sessions. With every metric you pull, you need to pull at least one dimension. In this case, we are pulling the ga:pageTitle metric. This will give us all the page titles for our website. There are a literal ton of combinations of metrics and dimensions. To make it easy Google provides the [Dimensions and Metrics Explorer](https://developers.google.com/analytics/devguides/reporting/core/dimsmets) which will let you pick the data that you need.

Next, we will write a member function that will simplify pulling these metrics for us.

def get\_page\_visit\_data(self):

analytics = self.init\_reporting()

response = self.get\_reports(analytics)

return response

This will return metrics to us in the form of JSON data. Here is a sample of what I got back:

response:{

'reports':[

{

'columnHeader':{

'dimensions':[

'ga:pageTitle'

],

'metricHeader':{

'metricHeaderEntries':[

{

'name':'ga:sessions',

'type':'INTEGER'

}

]

}

},

Finally, we need to write our main member function that will run everything for us.

def main(self):

starttime = time.time()

self.logger.info('Starting Google Analytics API Intake Daemon')

while True:

self.logger.info('Pulling site data from Google Analytics API')

response = self.get\_page\_visit\_data()

self.logger.info(

'Got back data of type: {}'.format(type(response)))

self.logger.info(

'Successfully pulled site data from Google Analytics API')

now = datetime.now()

self.logger.info('Scheduling next run at {}'.format(

now + timedelta(seconds=self.delay)))

time.sleep(self.delay - ((time.time() - starttime) % self.delay))

if \_\_name\_\_ == '\_\_main\_\_':

intake = AnalyticsIntake()

intake.main()

This is simply an infinite loop that sleeps for the duration of self.delay (defaults to 3600 seconds) and repeats itself until you break out of the execution. Save the file and exit. The next section will cover pushing our JSON metrics data to our Kafka Cluster.

**Sending Metrics to Kafka**

Create a new file called **mykafka.py** and add the following contents.

from kafka import KafkaProducer

import json

class MyKafka(object):

def \_\_init\_\_(self, kafka\_brokers):

self.producer = KafkaProducer(

value\_serializer=lambda v: json.dumps(v).encode('utf-8'),

bootstrap\_servers=kafka\_brokers

)

def send\_page\_data(self, json\_data, topic):

result = self.producer.send(topic, json\_data)

print("kafka send result: {}".format(result.get()))

In our constructor we setup our Kafka producer. We tell it that we want to send JSON data to our topic using the value\_serializer parameter. We also give it the Kafka broker list that we want to publish to. This is a list that contains the hostname and port of our brokers:

[ “mslave1.admintome.lab:31000” ]

This class only has one member function that we will utilize to publish messages to our Kafka topic. This simply uses the [kafka.producer.send](https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html) method to send data to our topic.

We capture the result of the send method and log out the results by using result.get(). There wasn’t much documentation on how to do this so take note. The result turns out to be of type kafka.producer.future.FutureRecordMetadata. Which I had a hell of a time finding any Python information for. [So when in doubt, go to the code](https://github.com/dpkp/kafka-python/blob/master/kafka/producer/future.py). This showed a  get() member function that I could use to get the status and sure enough that is what we get:

kafka send result: RecordMetadata(topic='admintome-ga-pages', partition=0, topic\_partition=TopicPartition(topic='admintome-ga-pages', partition=0), offset=18, timestamp=1530244755191, checksum=None, serialized\_key\_size=-1, serialized\_value\_size=21583)

This was the result of me pushing to my admintome-qa-pages topic in Kafka.

Save and Exit the file.

We need to update our AnalyticsIntake.py file to make use of our MyKafka class.

Add a new member function to our AnalyticsIntake class called publish\_metrics:

def publish\_metrics(self, logger, response):

kafka\_brokers = ['mslave1.admintome.lab:31000']

logger.info(

'Publishing site data to Kafka Broker {}'.format(kafka\_brokers))

mykafka = MyKafka(kafka\_brokers)

mykafka.send\_page\_data(response, self.topic)

logger.info(

'Successfully published site data to Kafka Broker {}'.format(kafka\_brokers))

This simply instantiates our MyKafka class and calls the send\_page\_data member function.

Next, Update the main member function of the AnalyticsIntake class:

def main(self):

starttime = time.time()

self.logger.info('Starting Google Analytics API Intake Daemon')

while True:

self.logger.info('Pulling site data from Google Analytics API')

response = self.get\_page\_visit\_data()

self.logger.info(

'Got back data of type: {}'.format(type(response)))

self.logger.info(

'Successfully pulled site data from Google Analytics API')

self.publish\_metrics(self.logger, response)

now = datetime.now()

self.logger.info('Scheduling next run at {}'.format(

now + timedelta(seconds=self.delay)))

time.sleep(self.delay - ((time.time() - starttime) % self.delay))

Save and Exit the file.

**Running Our Kafka Python Example**

Now we have everything ready to run our application except our environment variables.

$ export KAFKA\_BROKERS="broker.example.com:31000"

$ export KEY\_FILE="keyfile name"

$ export VIEW\_ID="00000000"

$ export TOPIC="admintome-ga-pages"

Substitute those values for the actual values from the beginning of this article.

Finally, run the application:

$ python AnalyticsIntake.py

2018-06-28 23:12:07,851 root INFO Set KAFKA\_BROKERS: ['mslave1.admintome.lab:31000']

2018-06-28 23:12:07,851 root INFO Set KEY\_FILE: .json

2018-06-28 23:12:07,851 root INFO Set VIEW\_ID:

2018-06-28 23:12:07,851 root INFO Set TOPIC: admintome-ga-pages

2018-06-28 23:12:07,851 root INFO DELAY environment variable not set - Setting to default 3600 s

2018-06-28 23:12:07,851 root INFO Starting Google Analytics API Intake Daemon

2018-06-28 23:12:07,851 root INFO Pulling site data from Google Analytics API

2018-06-28 23:12:08,900 root INFO Successfully pulled site data from Google Analytics API

2018-06-28 23:12:08,900 root INFO Publishing site data to Kafka Broker ['mslave1.admintome.lab:31000']

kafka send result: RecordMetadata(topic='admintome-ga-pages', partition=0, topic\_partition=TopicPartition(topic='admintome-ga-pages', partition=0), offset=19, timestamp=1530245529016, checksum=None, serialized\_key\_size=-1, serialized\_value\_size=21583)

2018-06-28 23:12:09,030 root INFO Successfully published site data to Kafka Broker ['mslave1.admintome.lab:31000']

2018-06-28 23:12:09,030 root INFO Scheduling next run at 2018-06-29 00:12:09.030824

The application will continue to run until you hit [CTRL]-[C].

**Building a Docker Container**

We eventually want to deploy this application to Marathon. The best way to do this is to build a Docker container. If you clone the [admintome/analytics-intake](https://github.com/admintome/analytics-intake) repository you will find a Dockerfile that you can use to build a Docker container and push it to your Docker repository.

**Deploying the Application to Marathon**

After you have built your Docker container and pushed to a Docker repository (like Docker Hub), you will need to do a Docker pull on all your Mesos slaves. Then create a new Marathon application using this JSON.

{

"id": "analytics-intake",

"cmd": null,

"cpus": 1,

"mem": 128,

"disk": 0,

"instances": 1,

"container": {

"docker": {

"image": "[docker repo username]/analytics-intake"

},

"type": "DOCKER"

},

"networks": [

{

"mode": "host"

}

],

"env": {

"KAFKA\_BROKERS": "mslave1.admintome.lab:31000",

"KEY\_FILE": "something.json",

"VIEW\_ID": "00000000",

"TOPIC": "admintome-ga-pages"

}

}

Make sure to substitute your correct values including your Docker repo username in the docker:image section.

**What’s Next?**

You now have a Python application deployed to Marathon that can pull metrics from Google Analytics and push that data in the form of JSON to your Kafka cluster.

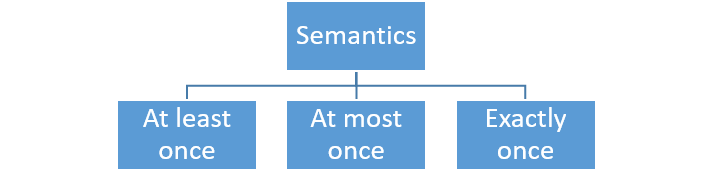
The next step is to take that raw data and analyze it into some meaningful and actionable data. We will do this using Apache Spark in a future article. Thanks for sticking with me through this article. If you enjoyed then please do me a solid and throw up a link to it.

**Kafka Producer Delivery Semantics**

This article is a continuation of [Part 1, Kafka Technical Overview](https://dzone.com/articles/kafka-technical-overview) and [Part 2, Kafka Producer Overview](https://dzone.com/articles/kafka-producer-overview) articles. Let's look into different delivery semantics and how to achieve them using producer and broker properties.

**Delivery Semantics**

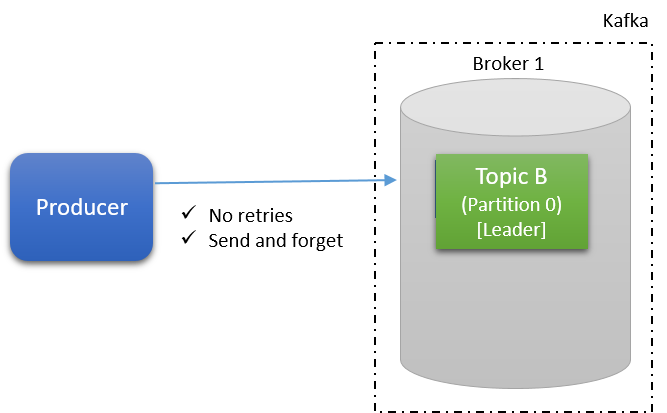
Based on broker and producer configuration, all three delivery semantics— “at most once”, “at least once” and “exactly once” — are supported.



*Different delivery semantics*

**At Most Once**

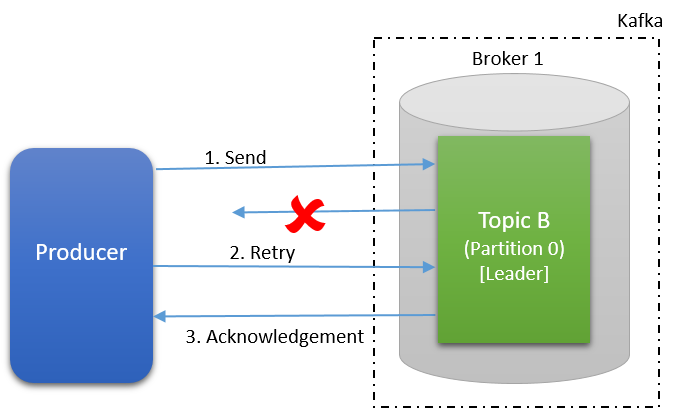
In an 'at most once' delivery semantics, a message should be delivered maximum only once. It's acceptable to lose a message rather than delivering a message twice in this semantic. A few use cases of at most once includes metrics collection, log collection, and so on. Applications adopting at most semantics can easily achieve higher throughput and low latency.



*At most delivery semantic*

**At Least Once**

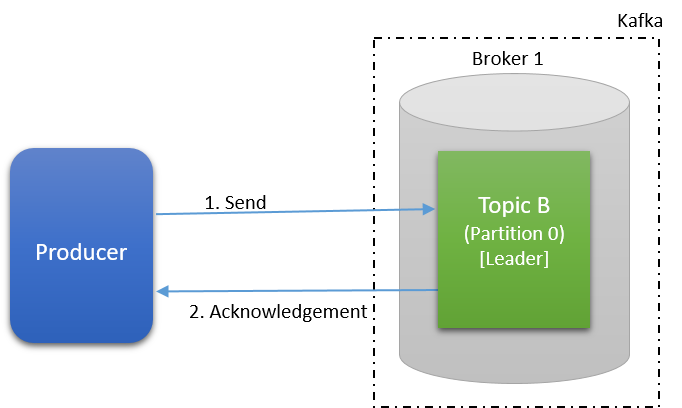
In 'at least once' delivery semantics it is acceptable to deliver a message more than once but no message should be lost. The producer ensures that all messages are delivered for sure, even though it may result in message duplication. This is the most preferred semantics system out of them all. Applications adopting at least once semantics may have moderate throughput and moderate latency.



*At least once semantic retry*

**Exactly Once**

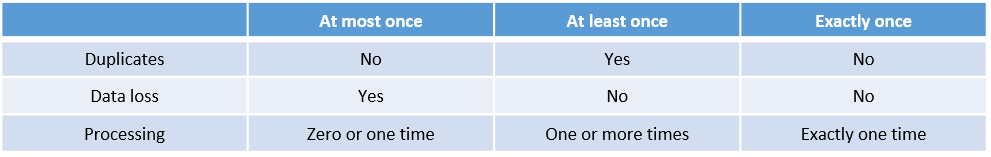
In 'exactly one' delivery semantics, a message must be delivered only once and no message should be lost. This is the most difficult delivery semantic of all. Applications adopting exactly once semantics may have lower throughput and higher latency than the other two semantic systems we've looked at.



*Exactly once delivery semantics*

**Delivery Semantics Summary**

The table below summarizes the behavior of all delivery semantics.



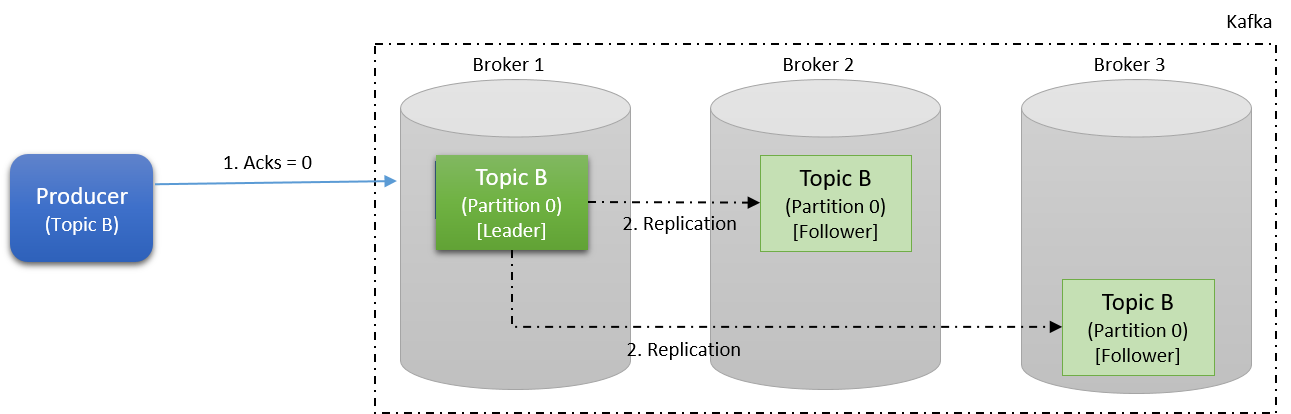
*Delivery semantics summary*

**Producer Delivery Semantics**

Different delivery semantics can be achieved in Kafka using the Acks property of the producer and the min.insync.replica property of the broker (considered only when acks = all).

**Acks = 0**

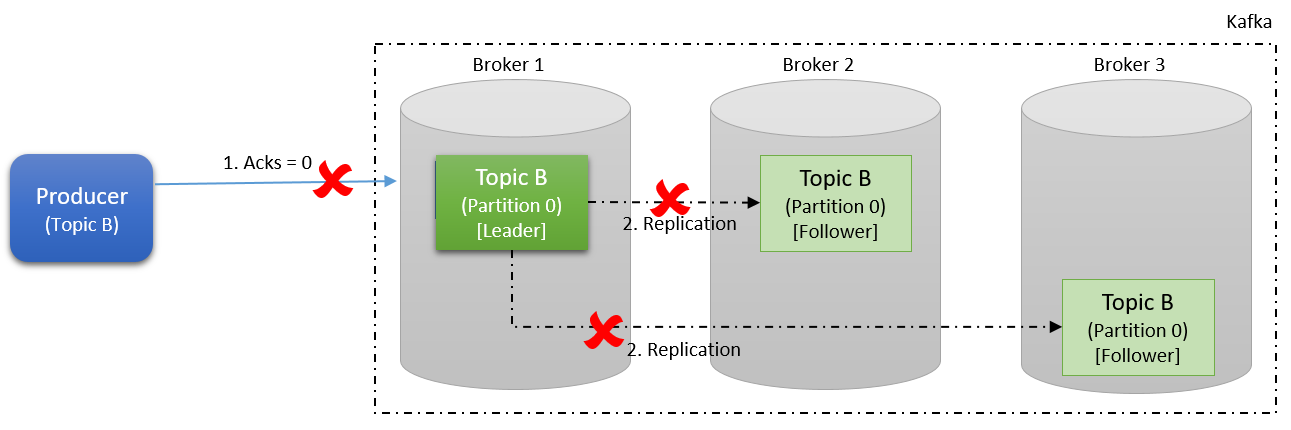
When the acks property is set to zero you get at most once delivery semantics. Kafka producer sends the record to the broker and doesn't wait for any response. Messages, once sent, will not be retried in this setting. The producer uses the “send and forget approach” with acks = 0.



*Kafka producer Acks = 0*

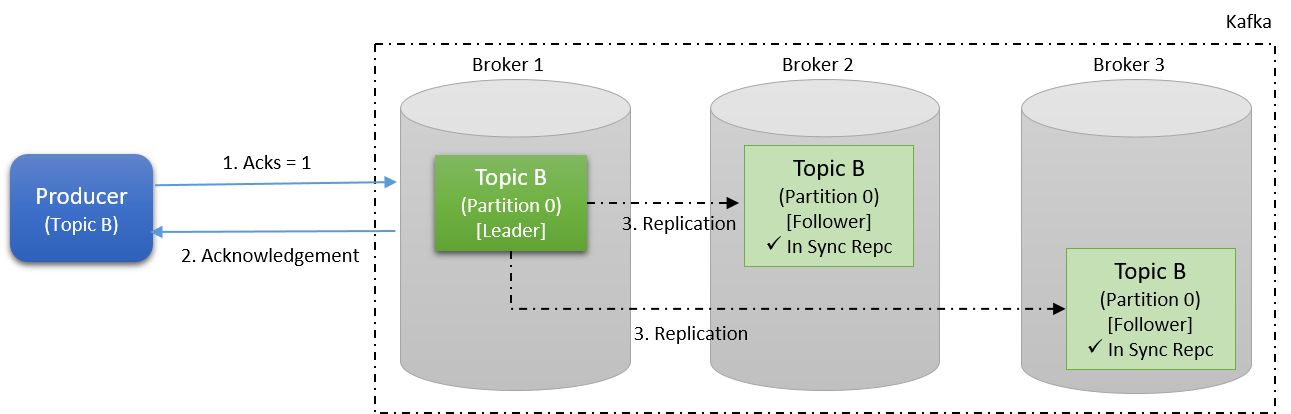
**Data Loss**

In this mode, the chances of data loss occurring are high, as the producer does not confirm the message was received by the broker. The message may not have even reached the broker or broker failure soon after message delivery can result in data loss.



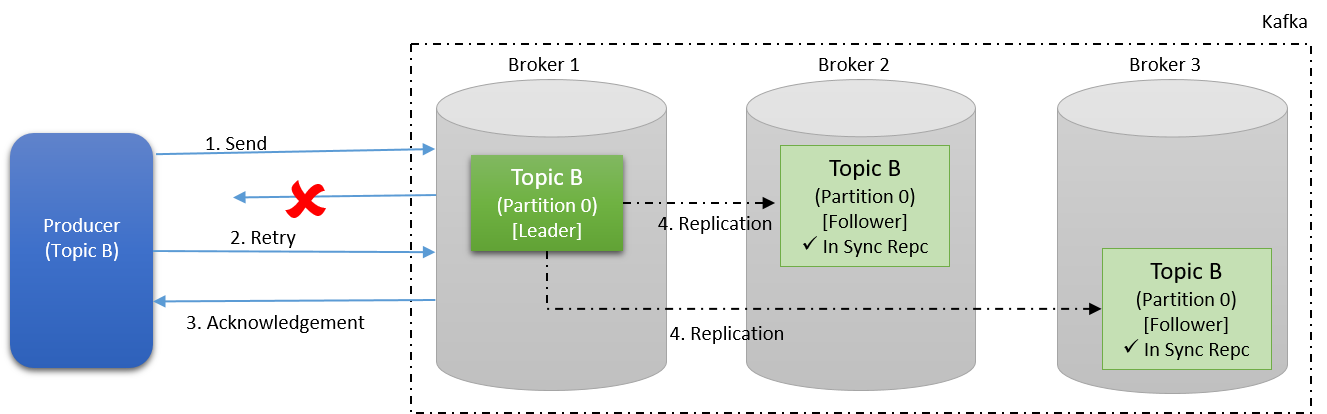
*Kafka producer Acks = 0 - data loss*

**Acks = 1**



*Kafka producer Acks = 1*

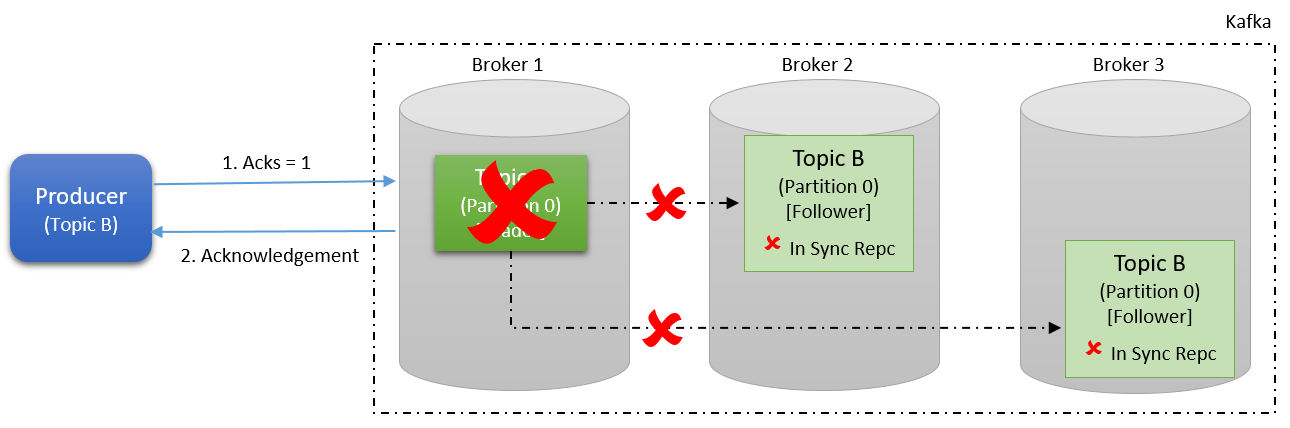
When this property is set to 1 you can achieve at least once delivery semantics. A Kafka producer sends the record to the broker and waits for a response from the broker. If no acknowledgment is received for the message sent, then the producer will retry sending the messages based on a retry configuration. The retries property, by default, is set to 0; make sure this is set to the desired number or Max.INT.



*Kafka producer Acks = 1 - retry*

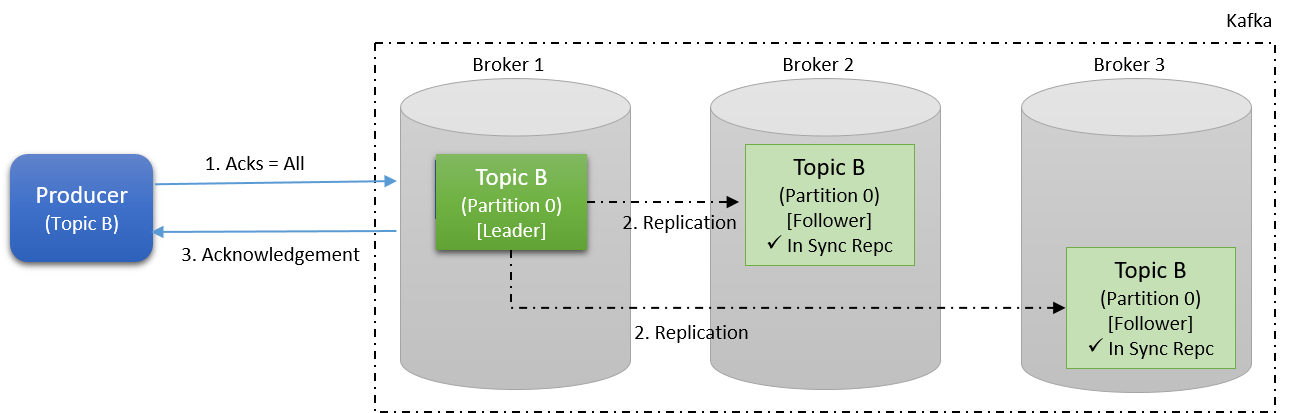
**Data Loss**

In this mode, chances for data loss are moderate as the producer confirms that the message was received by the broker (leader partition). As the replication of the follower partition happens after the acknowledgment this may still result in data loss. For example, after sending the acknowledgment and before the replication, if the broker goes down this may result in data loss as the producer will not resend the message.



*Kafka producer Acks = 1 - data loss*

**Acks = All**

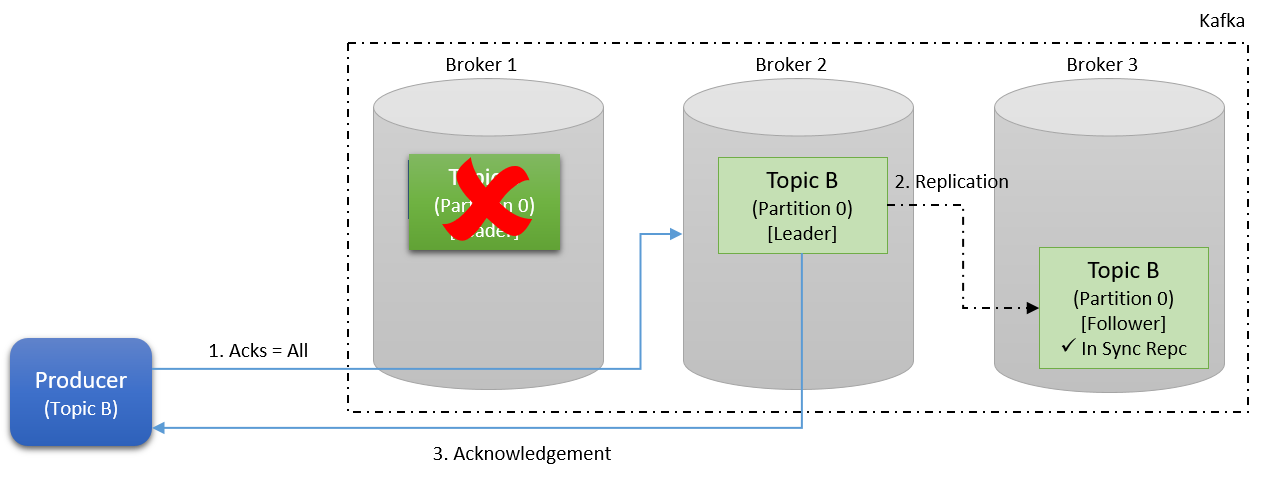


*Kafka producer Acks = all*

When the acks property is set to all, you can achieve exactly once delivery semantics. The Kafka producer sends the record to the broker and waits for a response from the broker. If no acknowledgment is received for the message sent, then the producer will retry sending the messages based on the retry config being set to *n*. The broker sends acknowledgment only after replication based on the min.insync.replica property.

For example, a topic may have a replication factor of 3 and a min.insync.replica of 2. In this case, an acknowledgment will be sent after the second replication is complete. In order to achieve exactly once delivery semantics the broker has to be idempotent. Acks = all should be used in conjunction with min.insync.replicas.

**Data Loss**

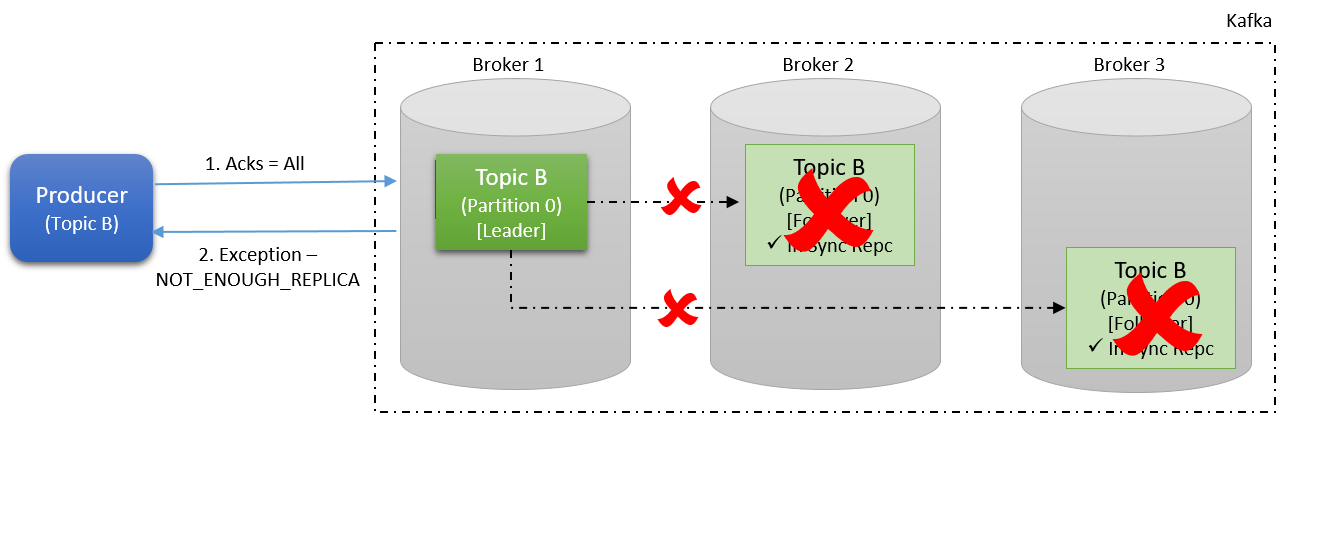


*Kafka producer Acks = all - data loss*

In this mode, the chances of data loss are low as the producer confirms that the message was received by the broker (leader and follower partition) only after replication. As the replication of the follower partition happens before the acknowledgment of data loss, the chances of actually losing data are minimal. For example, after sending the acknowledgment, and before replication, if the broker goes down, the producer will not receive the acknowledgment and will send the message again to the newly elected leader partition.

**Exception**

When there are not enough nodes to replicate as per the min.insync.replica property, then the broker will return an exception instead of an acknowledgment.



*Kafka producer Acks = all - exception*

**Safe Producer**

In order to create a safe producer that ensures minimal data loss, use the below producer properties.

**Producer Properties**

**• Acks = all** (default 1) – Ensures replication before acknowledgement.

**• Retries = MAX\_INT** (default 0) – Retry in case of exceptions.

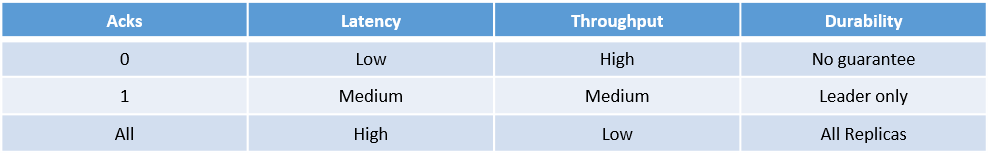
**• Max.in.flight.requests.per.connection = 5** (default) – Parallel connections to broker.

**Broker Properties**

**• Min.insync.replicas = 2** (at least 2) – Ensures a minimum In Sync Replica (ISR).

**Acks Impact**

The table below summarizes the impact of acks property on latency, throughput, and durability.



*Kafka producer Acks property impact*

**Summary**

Configure Kafka producers and brokers to achieve the desired delivery semantics based on the following properties.

• Acks

• Retries

• Max.in.flight.requests.per.connection

• Min.insync.replicas

In Part 4 of the series, we'll look into Kafka consumers, consumer groups, and how to achieve different Kafka consumer delivery semantics.