BENCHMARKING:

RABBITMQ V/S KAFKA

# DIPEN KUMAR (2018CS50098)

This report describes my benchmarking project between RabbitMQ and Kafka. The project was taken and completed as part of the course on Cloud Computing Technology Fundamentals, COL733, instructed by professor Abhilash Jindal, 2nd Semester, 2021-2022, Indian Institute of Technology Delhi. This project was not done in group and nor is this associated with any other project.

This report explains the example architecture of different microservices I used for benchmarking RabbitMQ and Kafka. This report will guide you through my project to independently reproduce my findings. The report gives instructions on how to set up the system, where to get the code, a walkthrough of the code, how to run the experiments, and how to interpret the results.

# Introduction

In this new age of evolving architectures, events have always been consistent. We use events in one form or another while processing messages. With multiple open-source platforms, it is difficult to choose one over the other. In this project, we are going to understand the trade-offs between RabbitMQ and Kafka, which can help us in choosing the right messaging platform within given event-driven architecture.

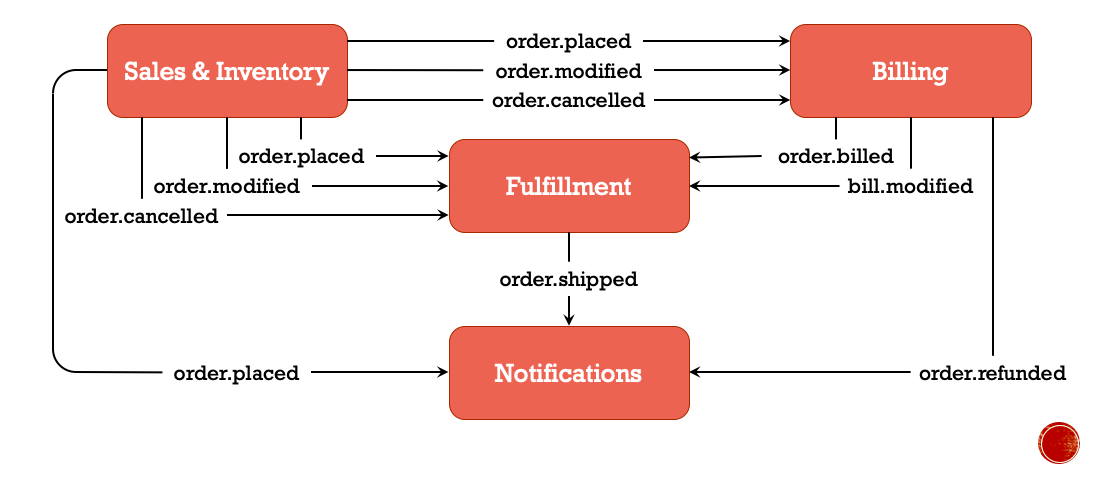
At a high level, Kafka and RabbitMQ have some common use cases. For example, both can be used as part of a microservices architecture that connects producing and consuming apps. Both can also be used as a message buffer, providing a location to temporarily store messages when consuming apps are unavailable or smoothing out spikes in messages generated by producers. Both can also handle very large amounts of messages. But because they handle those messages in different ways, each is best suited for subtly different use cases.

# Benchmarks

I implemented a simple microservices based architecture where services interact with each other in form of events, requiring a message broker between them. I used this real world example architecture to test RabbitMQ and Kafka as message broker. I used synthetic data I created for the simulation. I measured performance as messages processed per unit time, end-to-end latency for a message to traverse the pipeline from the producer through the system to the consumer, tradeoffs between latency with durability and availability, acknowledgment based message retention in RabbitMQ v/s policy based message retention in Kafka.

# A Simple Example Architecture

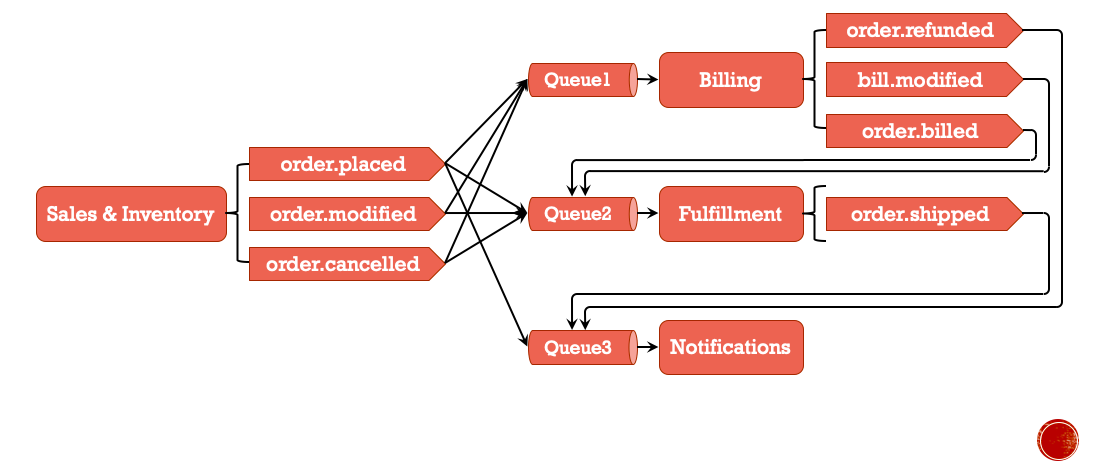
I have used a microservices based architecture taken from Jack Vanlightly [1] described below, and overlaid it with RabbitMQ and Kafka kind of use cases representing queue based system and log based system respectively, and compared their performance. The example architecture has four services that interact via events.



The Sales & Inventory service emit three events one for each order placed, order modified, and order cancelled. These events are then consumed by Billing service who then creates an entry in billing table capturing the transaction type. Once the bill has been generated, billing service then further produce three events one each for order placed, order modified, and order cancelled. Events produced by billing service is consumed by fulfillment service which then ship the order only if it has not yet been cancelled or shipped producing a corresponding event. Finally the last service called notifications service which don't produce any further event but consumes three events— order placed, order cancelled, and order shipped by three different services— Sales & Inventory, Billing, and Fulfillment respectively.

# RabbitMQ

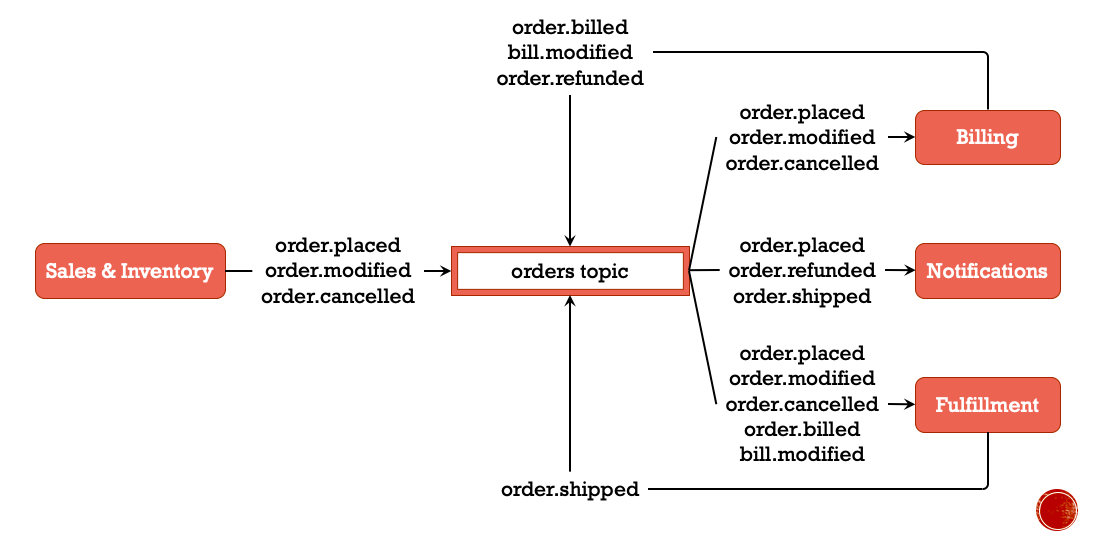
If we overlay this with a RabbitMQ kind of use case with queue, we end up with the design described below. I have used celery to create workers for billing, fulfillment, and notifications services, taking tasks from queue.



Here, order placed, order modified, order cancelled, order refunded, bill modified, order billed, order shipped are events produced by sales & inventory, billing, fulfillment which are our different microservices in the architecture including notifications, just consuming events without producing any.

# Kafka

If we overlay this with a Kafka kind of use case with log, we end up with design described below. I had spanned few concurrent threads as workers for taking task on behalf of billing, fulfillment, and notifications services.



Here, we have created a single topic for all orders event. We have Sales & Inventory, Billing, and Fulfillment— all three services pushing messages in the same topic and another three services— Billing, Fulfillment, and Notifications consuming events from the same topic.

# Design Implementation

I have created a Django web application which listens for HTTP GET request with two arguments— path to synthetic data and message broker to use in simulation. The handler for the run-simulation-get-request is the Sales & Inventory service mentioned above and it reads orders.in and according produces three events one each for order placed, order modified, and order cancelled. If message broker specified in get request is RabbitMQ, then it pushes the celery task corresponding to event type in the RabbitMQ queue, else it pushes the message in Kafka Topic.

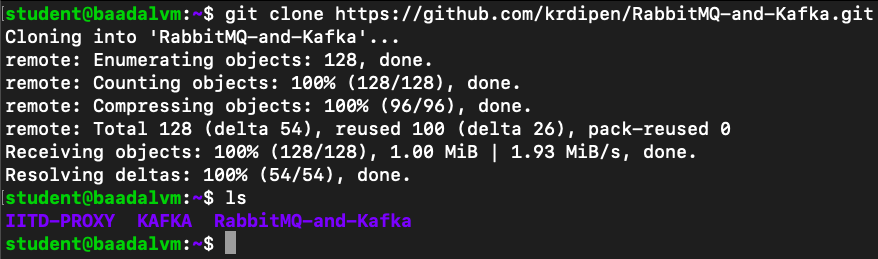
Celery workers polls tasks from the queue and according does the roll for the Billing, or the Fulfillment, or the Notifications services if our broker was RabbitMQ. In case of Kafka I have created another HTTP GET request handler which takes concurrency value as an argument and accordingly forks threads that further polls message from Kafka topic and according does the roll for the Billing, or the Fulfillment, or the Notifications services, just the same way celery worker does with RabbitMQ queue.

And the last HTTP GET request handler I created is for generating synthetic data for the simulation with path and data size as arguments. I have used PostgreSQL as Database Management Software for the project. There are three tables in the database. First table is orders table which is filled and modified by Sales & Inventory as the orders are placed, modified, or cancelled. Fulfillment service also modifies orders table if it ships any order. Second We have inventory table which is edited by Sales & Inventory again whenever an order is placed, modified, or cancelled. The last table is bills table which is edited by Billing service. Notifications service doesn't edit any table it dumps its notification in a file named "notifications.out" in data directory.

# Installation Manual

Install RabbitMQ, Apache Kafka, Postgres, Python3, and Git in your system. Installation of these three open-source software can be easily be found on internet from different sources including their official documentation.

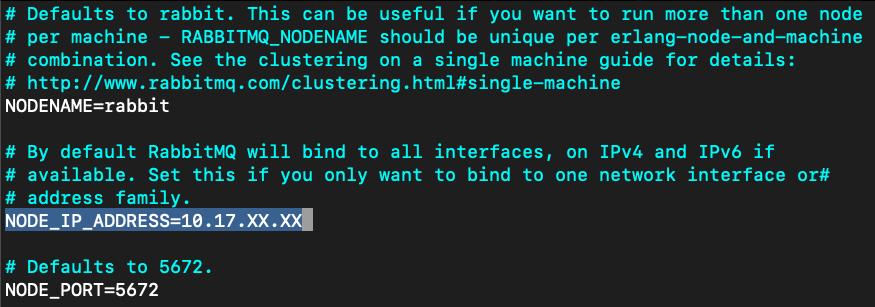
Download the project from "https://github.com/krdipen/RabbitMQ-and-Kafka.git" using git clone command on your ubuntu system.



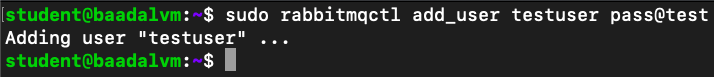
# Execution Procedure Manual

Change the configuration file of RabbitMQ for its server to listen on the public IP address of your system. The path along with filename is for the configuration file is "/etc/rabbitmq/rabbitmq-env.conf". Use sudo command with vim to edit the file. Also add a user in your RabbitMQ server to allow him/her access the server. Finally start or restart the server as is the case. The commands and the screenshot of terminal is shown below for you to follow along the set up procedure.

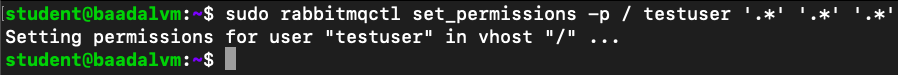
$ sudo vim /etc/rabbitmq/rabbitmq-env.conf



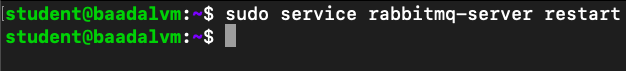
$ sudo rabbitmqctl add\_user <username> <password>



$ sudo rabbitmqctl set\_permissions -p / <username> '.\*' '.\*' '.\*'

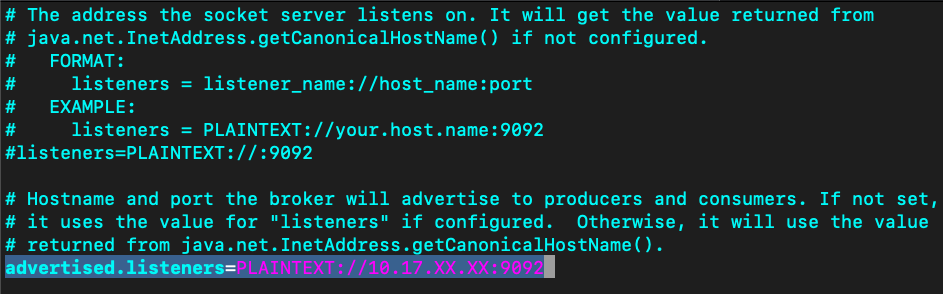


$ sudo service rabbitmq-server restart

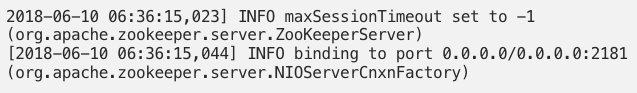


This completes for RabbitMQ work. Our RabbitMQ is now running. Next let's move to Kafka and start the server but before that we will edit the configuration file to allow it listen on public IP address and start zookeeper server. Inside the Kafka directory we downloaded we find configuration file at "config/server.properties". Edit the file with your own IP address. The screenshot of terminal is attached below.

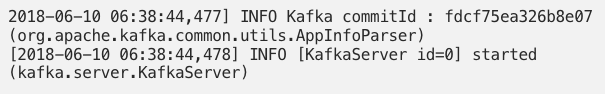
$ vim config/server.properties



$ bin/zookeeper-server-start.sh config/zookeeper.properties

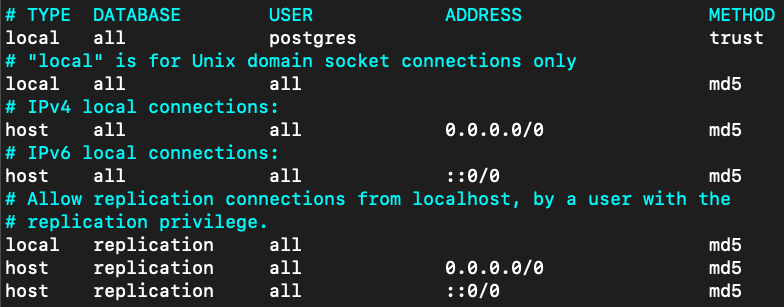


$ bin/kafka-server-start.sh config/server.properties

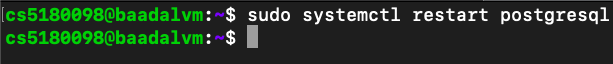


Now our Kafka server is also running. Yay! Now let keep moving further. We will now start our postgres server and create a database in there. But again before that lets edit our postgres configuration file at "/etc/postgresql/12/main/pg\_hba.conf" to allow it listen on public IP address and also change the settings for it to allow password based authentication. We will also create a new postgres user and set its password and allow him the access for the database we just created for our application by granting them all the privileges of operations on the database. We will use out master postgres user which doesn't require password for login to create other user. The screenshots of the terminal is shown below.

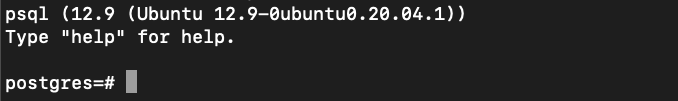
$ sudo vim /etc/postgresql/12/main/pg\_hba.conf



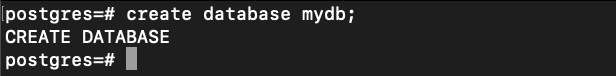
$ sudo systemctl restart postgresql



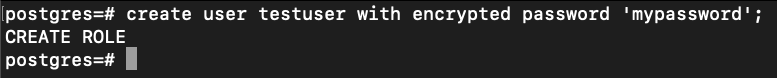
$ psql postgres -U postgres



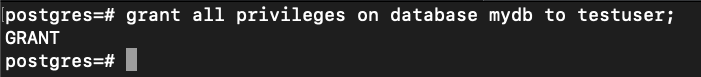
postgres=# create database mydb;



postgres=# create user testuser with encrypted password 'mypassword';

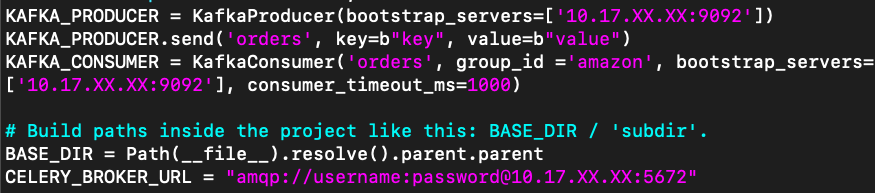


postgres=# grant all privileges on database mydb to testuser;

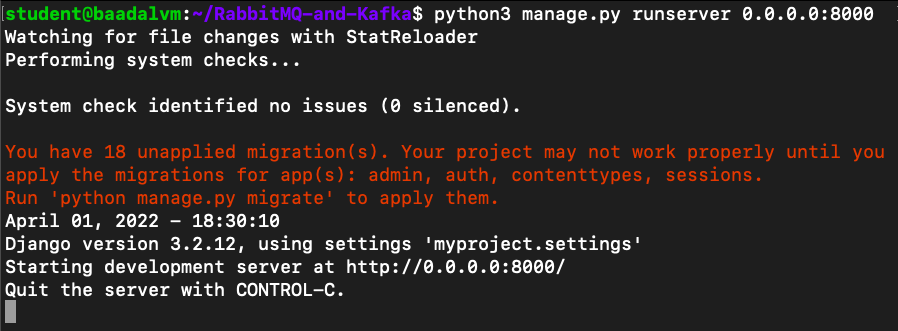


Finally, postgres server is also running. We are now ready to run our application. But there need some editing in the project in order to use above started services. In the project downloaded from GitHub, cd to the repository and edit "myproject/settings.py". Change the IP address with your own IP address and username, password for RabbitMQ with your own created in above steps.

$ vim myproject/settings.py



$ python3 manage.py runserver 0.0.0.0:8000

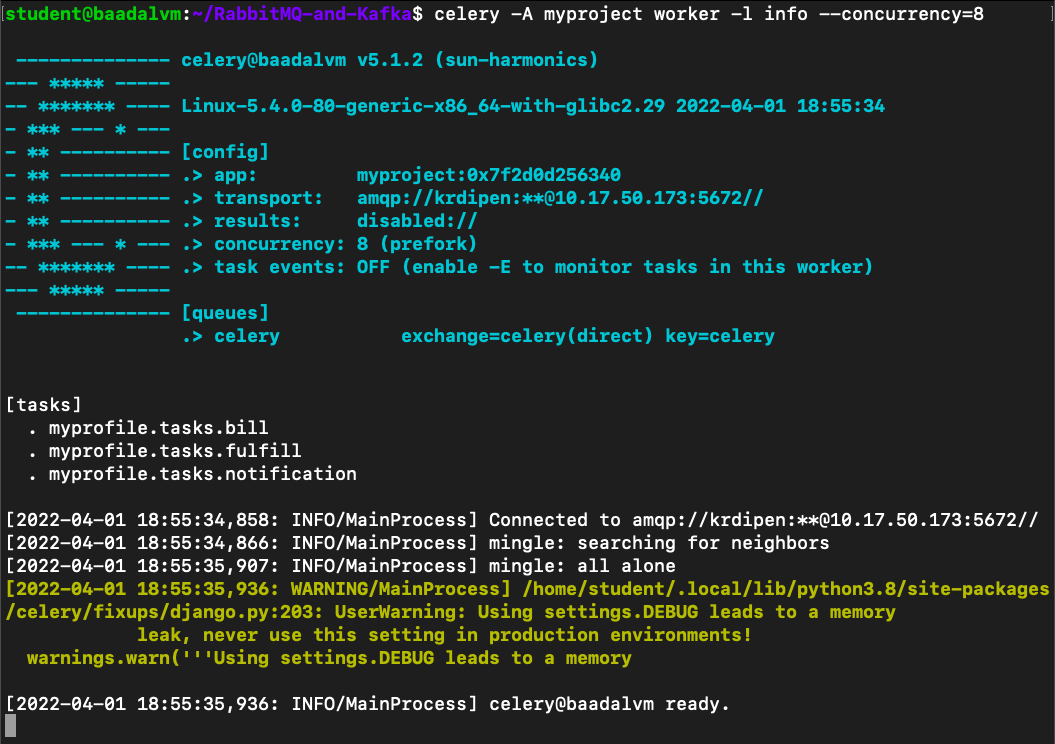


Now our application is also running and listing on port 8000. We can now start the simulation by firing a GET request on our browser mentioning the path and broker http://10.17.XX.XX:8000/runsimulation?filename=<filename>&broker=<msg-broker>

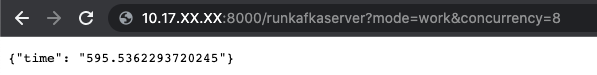
Here <filename> is "orders.in". This is the default synthetic data that comes with the project and it is in data directory inside repository. You can create more synthetic data by another get request from the browser and it will be stored in the same data directory with specified filename as in GET request and data size as mentioned in the GET request "http://10.17.XX.XX:8000/createdata?filename=<filename>&datasize=<int>" where <int> can be 500, 1000, 5000, or 10000

In the above, runsimulation get request <msg-broker> can be "rabbitmq" or "kafka". If RabbitMQ then next we need to run celery worker else we will run workers to consume from kafka topic. See below screen shots to run celery worker in case of rabbitmq and kafka workers in case of kafka respectively.

$ celery -A myproject worker -l info --concurrency=8



http://10.17.XX.XX:8000/runkafkaserver?mode=work&concurrency=8



Above GET request will start kafka server with concurrency as mentioned in request. The response of GET request will show the time it took to process the messages in the topic. In case of celery work we have to measure the time manually till we see that tasks are being received and processed on celery terminal output. Once it stops showing output we assume that jobs are done and note down the time.

Note that in repository there requirement.txt which will install all necessary python packages needed for the project.

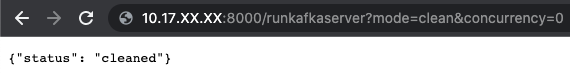
$ pip install -r requirement.txt

Congratulations we have now executed our first simulation. Important things to keep in mind before we start our next simulation. In by any case the application fails to execute completely then there is a case that our kafka topic or rabbitmq queue is dirty for the next simulation. So, as a good practice purge celery tasks and clean kafka topic before moving to next simulation. Commands are show below with screenshots.

$ celery -A myproject purge -f



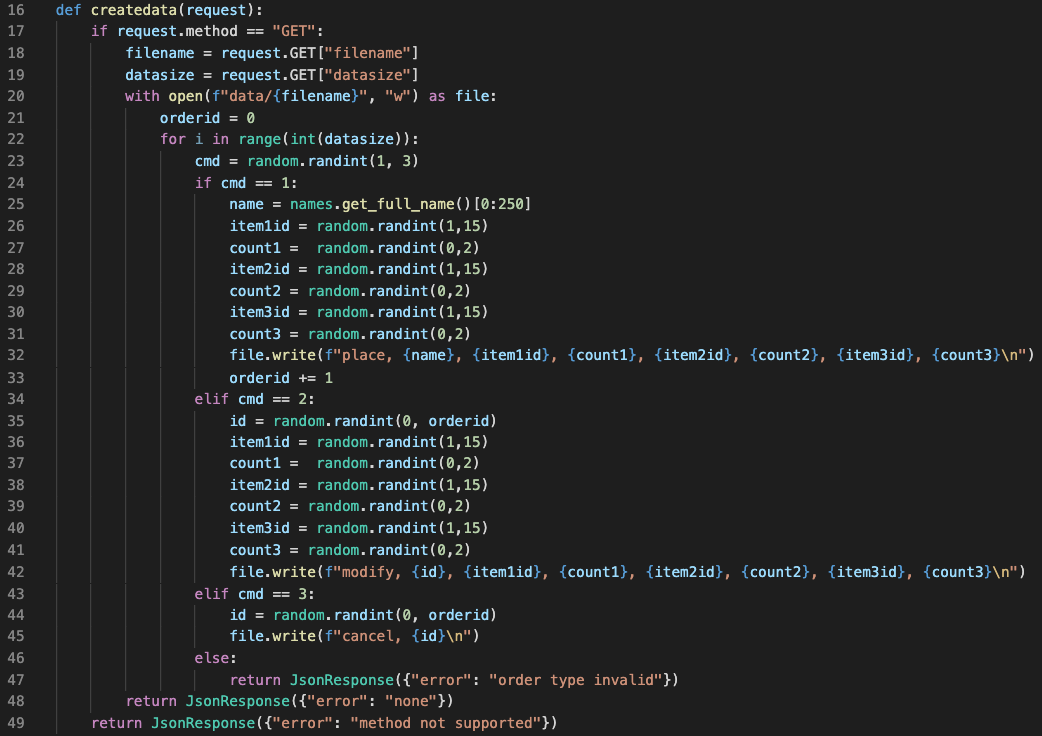
http://10.17.XX.XX:8000/runkafkaserver?mode=clean&concurrency=0



# Code Description

The main working bytes of the project is written is two files— myprofile/views.py and myprofile/tasks.py where I have written three GET request handlers and three celery tasks respectively. Let's see them one by one.

http://10.17.XX.XX:8000/createdata?filename=<filename>&datasize=<int>



Randomly among order placed, order modified, or order cancelled I pick one uniformly and add entry in synthetic data. I do this till number of order request is equal to data size received in argument.

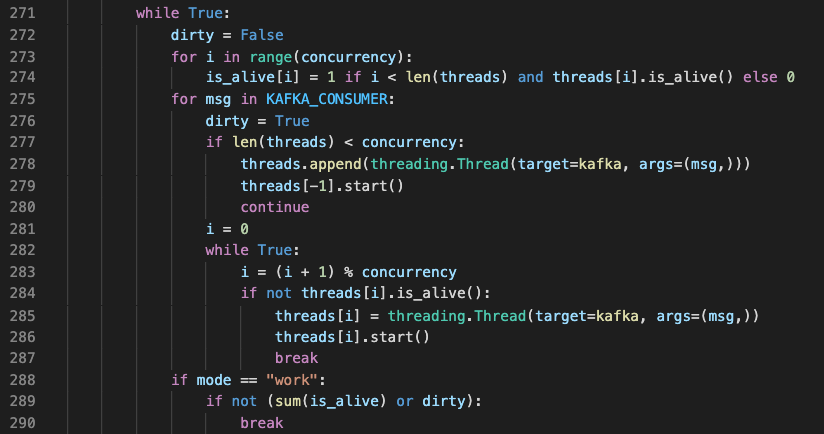
Here order place have name of customer along with products and their quantity. For simplicity in database I only allow at max three different product in a single order. For order modification only provide order id to modify with new set of items, and only order id in case of cancellation request. No modification of customer name is allowed in order modification. For simplicity only 15 different types of order exist in inventory from id=1 to id=15. Each order when placed is assigned an order id which in our case is its order of arrival i.e. 1 if it is the first order in our simulation.

Next handler is runsimulation which reads orders.in line by line and edit inventory table and orders table atomically in a transaction. It creates a new entry in order is new order is placed and update inventory accordingly. It modify existing order in orders table if modification or cancel request is received and update inventory in transaction atomically to maintaining consistency in database.



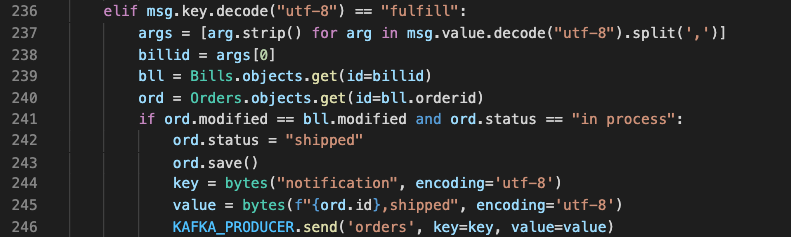
Above show is code for cancel request when received by runsimulation handler. Similar is for order placed and order modified. Once the execution is successful it will either push the event in RabbitMQ queue or Kafka Topic depending on the argument it receive as broker = "rabbitmq" or "kafka" respectively.

Next and the last handler for GET request we have is for kafka worker, I have implemented workers for listing to kafka topic as python threads while for rabbitmq it is celery. This handler spans multiple threads and start reading kafka topic. Reading the topic is critical section which is same as in case of celery workers because RabbitMQ queue is single threaded. Hence, for comparing kafka and rabbitmq it is on same page. multiple threads will do work parallelly. It is observed that in multiple threads the overhead is too high that execution is come way more than expected. When not using any threads and doing work in master thread it was seen that execution time was approximately 100sec but with threads and keeping concurrency as 1 it comes more than 1000sec. Hence for comparing Kafka with RabbitMQ we should see for results without any python threads library.



Above loop keeps executing till all messages are read and for performing task a new thread is forked. New thread creation overhead is way more than the actual time it takes for the job.

Time taken to do the job is same as writing one entry in database which is considerably low. Below shown is the job done by the code by fulfillment service.



Similarly it is for billing and notification. For notification it is even faster since it write to file in local computer rather than writing in database.

Next code of interest is tasks for celery worker which is the same as discussed for kafka worker i.e. to write one entry in database table while writing in file for notification task. The only difference is that it reads from RabbitMQ queue while kafka worker reads messages from kafka topic.

# Results

I have measured throughput and latency in each cases— RabbitMQ and Kafka. For throughput I have seen how fast the simulation is executed when taken RabbitMQ as message broker v/s Kafka as message broker. Since the job done by both of them is same hence the broker whose reading and writing messages in queue/topic is fast will complete the execution fast. I have tested it on 10000 orders. You can test it on other values but the idea motivation is satisfied by any given it is sufficiently large i.e. of approximately few minutes of execution of suppress overhead of latency.

In case of RabbitMQ throughput came to be around 32 order per second while in case of Kafka it was 100 orders per second. Here 1 order involves many tasks in the pipeline and also it includes the time taken to process it along it writing and reading time from queue or topic. Hence, see this numbers to get an idea what good kafka is relative to rabbitmq or how bad RabbitMQ is relative to Kafka instead of seeing these number to get idea of how good Kafka or RabbitMQ performs in absolute sense.

These reading were taken with concurrency = 1in both cases— RabbitMQ and Kafka because of the reason already discussed above in Code Description section, where I mentioned that how large is the overhead of creating new thread in python library is that when performed with master thread it takes 100sec whereas when created a new thread for every message it went as high as 1000sec, i.e. 10 times slow. Hence for comparing Kafka v/s RabbitMQ in my benchmark we are bound work on single thread.

My proposed solution for the above problem is that I should not create new thread for every message rather should at start create n number of threads as specified and never let these threads terminate and allow these thread to read directly from kafka topic with synchronization since kafka consumer function of the used library don't support concurrent writes. Hence In this case I only created n number of threads where n is < 10 whereas in existing case I am creating as many as 10000-20000 threads and terminating it.

Anyway even if we can't compare RabbitMQ v/s Kafka in concurrency > 1, we can still see how RabbitMQ performed relative to itself with varying concurrency. Also we can see how kafka performed relative to itself with varying concurrency. We can then have a rough idea of RabbitMQ v/s Kafka with varying concurrency if we scale down kafka time by 10 while using threads because we saw it increase the time by 10. This is an approximation.

Now, Let's calculate latency. Latency is defined as the time it takes to flow message from producer through pipeline to reach consumer. Here I have not calculated absolute latency for Kafka topic or RabbitMQ queue. I have noted the time it takes to write the message in the pipeline, read from the pipeline, process the message, again write to pipeline, again read, and again processes till every microservices does their part of work for the particular message in stages. Hence For RabbitMQ it took 130ms whereas in Kafka it took 70ms, again I repeat that these values don't give absolute performance of RabbitMQ or Kafka but give relative performance of these two platform relative to each other because message processing time is same for both cases as their processing piece of code is identical. I measure these value but simulating the application only on one single order being placed. I noted how fast the notifications.out file is created in both cases because notification is the last microservice the message is processed through. I used only single order in simulation to suppress the advantage of throughput any of these platform could enjoy.

I have also plotted the latency against the percentile of order having lesser or equal. This of course captures throughput in large proportional as faster will show low latency. The higher number of order we simulate the more we give throughput importance in our measurement.

10000 orders for simulation big enough for throughput to suppress latency. hence above graph is similar to what we saw in throughput. Above readings are done with concurrency = 1 for the same reason discussed above.

# References

[1] Jack Vanlightly, " Event-Driven Architectures - Queue vs Log - A Case Study", https://jack-vanlightly.com/blog/2018/5/21/event-driven-architectures-queue-vs-log-case-study, May 21, 2018