Application development report

Introduction

The following items describe the process of the development of the application which has the goal to determine the number of unique users per minute, for the given stream of log messages as an input. A personal observation is given in the final item.

Setting up the development ecosystem

For the purposes of installing and running the Zookeeper and Kafka servers, I installed the Ubuntu 20.04 LTS Linux OS on a virtual machine with 4Gb of RAM memory. It was an easier solution to run the necessary commands from the terminal on the Linux OS. There were problems regarding the occupied default port 2181 for Zookeeper and the need to kill the process that occupied it as a solution. Also cleaning the logs that were generated by Kafka server, as a solution for errors in running after unregular starting or stopping of server was needed. Configuration of the properties file for Kafka was needed so an option for deleting a registered topic could be achieved. From Kafka I could register the needed topic, view a list of topics. I could start a consumer and a producer. The producer could read the data from the stream.jsonl file or it could be manually inputted into the terminal for sending over the registered topic. The consumer was registered with the property –from-beginning so it could always start reading the messages from the topic from the begin, even if they were previously loaded.

It was useful to understand the architecture, where Kafka is a message broker, and is an event driven messaging system. The way I used Kafka in my project was a simple case where we had one Kafka server. In the real-world scenario we would have multiple Kafka servers, Kafka clusters, where we would enjoy the properties of the system such as distributed data, which result in fault tolerance. Also, a more complex user application could be scaled with adding more brokers and more consumers. Topics in my application are simple, where if we had a larger scale problem with a high data load, we would need a topic with more than one partition for the producers to subscribe to. On the consumers side, where every consumer is a part of the consumer group, we develop the logical code that processes the received data for a desired result to store or to push on forward.

So, I recognized Kafka server as some sort of a data plane, and I needed to understand the role of the Zookeeper server. It is stated that the Zookeeper is a configuration managing system. As I said previously if we had a more complex application with a Kafka cluster made up of multiple brokers, where data is distributed among them, we would need to keep the configuration and components statuses, perform synchronization of the configuration architecture and statuses. Zookeeper could be recognized as a control plane, marking up every architecture change and distributing that information.

Application development process

As suggested, I used a programming language that supported Kafka with a Kafka client as an imported library. Given that the problem I had to solve was in the area of data engineering, Python was a good choice since it is well documented, community supported, and full of user made libraries useful for data analyses. A good choice for further development. The Kafka client I used, kafka-python, builds the necessary Api for defining, implementing and running of the Kafka components I tried to ran in Kafka directly in the beginning in the testing and setting up stage.

I wanted to view and learn, recognizes the key value elements in the stream.jsonl file, see how one message looked like and find out more about timestamp format Unixtime. Understanding the substance of the messages I got from reading the stream of json objects, helped me in defining the further logic. I did some experimenting to first create a producer that would publish a loaded json file and since it was a large file, logically it clogged the ram memory, which made me think about some sort of solutions for a real-world problem. Changing the message format from .jsonl to .json and implementing the ijson python library, that wouldn't load the whole file at once but process the data as a stream, could be a good solution. Thankfully here, I could extract the stream.jsonl file from a Kafka directly generated producer, and get data as a stream, not having the file load problem.

So, getting the messages, and outputting them to the stdout, and even outputting the results as a stream of json objects was simple, a matter of implementing the Api. At first, I even tested to see the flow of same data from the producer I made in python (as previously mentioned, where now instead of loading all messages, I made a simple_stream.jsonl file that had only four messages to load), and that producer sent the data to the consumer-producer script which had both a consumer and a producer. And finally, to a final consumer, that outputs the results to stdout. So, I had a stable flow but the main part, the processing of input data, in the consumer-producer script was the key element to getting the result data to publish over the second topic, the result topic.

The main parts were choosing the right data structure and to determine the time threshold when I would output an instance of the result, since the result is given for every minute time range. I started looking up the most common in use, where at first, I remembered array and stack as the basics. But my application needed the ability for fast searching and smart storing the data, since I have a high input of data. Hash table as a solution seemed as a smart choice since I would store data depending on the hash function, wouldn't just make a complex long array. It's a fast way to search through the data structure. If a unique user, in the time range in which we are observing the inputted data, is already in the structure, don't store it. So in the end my data structure has a certain number of unique users. So, in my further research I recognized python dictionary as a structure that was the most similar to this principal. The same ability of fast searching the data structure for the inquired element is available.

Now that we have the means to store and search over a partition of the input data load, where the partition is determined by the length of the time range we are observing and the number of log instances a certain time range populates. The goal is simple, how many unique users, accompanied with their timestamps, are there in a minute of time. First, I needed to determine which time I am observing.

For a simple implementation I decided to perceive time through the Unixtime format, the time stamps that were logged, and to determine a minute passing not as a time measured time period but as the time on the clock that issued those time stamps. For an example, the very first message in the stream of data is when reading the Unitime time stamp logged at 16:39:44, and the whole minute of clock time is only 16 seconds far. So, at 16:40:00, I will make my decision and output the result, number of unique users in the observed minute. This is a way of syncing to the machine that generated the data but it's not so safe to put so much trust in it. In a better solution I would have to take into account that I have outlier data, or data that simply is taking longer time to be received, so its late to be processed in the correct minute interval. Maybe a delay in processing, of 60s range, that would buffer data and output it a little late but give a chance to get more correct results would be a good solution. Real-world sensor data isn't always received chronologically, there are bumps, but also some control movement application expects fast processing of data for a good motion response. So we are always between performance in speed and error in processing.

Thoughts about storing data and benchmarking

With benchmarking and performance measurement, we can see how big of a data load our hardware can handle and if the application solution is sufficient or there is room for improvement. There is a default benchmark tool kafka-*-perf-test for a producer/consumer in Kafka that creates a performants test for stress testing and load testing. It generates workloads, giving us the ability to test the current configuration and determine the points at which we need to reconfigure in order to handle that workload.

Since Kafka stores the stream of event messages, short-term, comparing with databases that handle long-term storing, we could combine the two and use the best of both worlds. Archiving the stream data into the database and performing data analyses, data processing and data classification, comparing the set of data that was outputted and the set that was received. Giving us the option to see the error in transmission, the number of outliers in regards to classification. All with a goal to do two things, improve the application for simultaneous data transition and processing, or to perform model training and control patterns in regulation and control.

Personal observation and comparisons

In the first overview of Kafka server, I perceived it as a message broker. Something similar to roscore node in ROS (Robot Operating System) were all the concepts of the message broker and message exchange are very similar and regarding industrial purposes it is also a good solution for robotic and automation applications. But Kafka is so much more given the abilities of storing and scaling application. I was inspired to try to create a combination of the two and overcome some limitation using just ROS.