Federator.ai® Drastically Improves Cost and Performance of Kafka Running on Kubernetes

Introduction

A Kafka Stream Application reads from a topic, performs some calculations, transformations to finally write the result back to another topic. To read from a topic, it creates a consumer. Kafka has become a standard tool to manage a highly loaded streaming system. However, it does not provide mechanisms for dynamic cluster capacity planning and scaling strategies. Scaling the application is about running more of the consumers. Generally, to optimize the consumer processing rate, users may assign the same number of consumers as the number of partitions for related topics. However, using such an Over-Provision policy to allocate consumers may result in unnecessary waste of resources. Hence, there have been works trying to leverage Kubernetes Horizontal Pod Autoscaler (HPA)[1] to manage consumer groups in a Kafka cluster[2] [3] and hope that the autoscaler can handle the allocation of the right amount of resources at the right time to achieve better utilization of the resources. There are some disadvantages to Kubernetes HPA:

a) Kubernetes HPA combines recommendations (calculating the desired replicas) and executions (adjusting the number of replicas) to set the number of replicas by the HPA controller. However, we have seen more and more operator-based applications in a Kubernetes cluster. Kubernetes HPA is not suitable to auto-scale operator-based applications. And users may need only recommendations and run customized executions separately.

b) If metrics are not chosen appropriately to calculate desired replicas, adverse effects on performance might happen. Users need to take extra care to find a proper metric by trial and error.

In this article, we would like to show that the Native Kubernetes HPA algorithm (K8sHPA mechanism) results in modest saving and much larger lags (latency). Federator.ai from ProphetStor uses Machine Learning technologies to predict and analyze the Kafka workload, and then Federator.ai recommender recommends the number of consumers, considering the benefit and cost with the adjustment. We can achieve much better performance (reduced latency) and use much fewer resources (reduced consumers in Kafka), all without changing the K8sHPA mechanism or a line of code of Kafka. Therefore, users can exploit Federator.ai's

recommendations and customize their executions more flexibly.

Current Issues

Kubernetes HPA uses the HPA controller to periodically (every 15 seconds) adjust the number of pods in a deployment based on the k8sHPA mechanism: the ratio between desired metric value and current metric value [1].

$$\text{desiredReplicas} = \left\lceil currentReplicas \times \frac{currentMetricValue}{desiredMetricValue} \right\rceil, \quad (1)$$

where desiredMetricValue can be a targetAverageValue or targetAverageUtilization. Kubernetes 1.6 adds support for scaling applications based on multiple metrics and custom metrics. Take Kafka as an example, targetAverageValue can be 1,000 lags in a topic for a consumer group. The targetAverageValue is based on *users' experience*. However, according to the scale of Kafka in Netflix, it should be able to manage about 4,000 brokers and process 700 billion unique events per day [4]. Manual configuration by *users' experience* is not feasible, to say the least, to manage such a Kafka cluster. We propose to use the AI-based management tool to handle the complexity to relieve the pain points of the user.

In general, a *topic* accumulates lags at the beginning stage when producers send messages. The consumers then reduce the lags in a *topic*. Kubernetes HPA *periodically* checks the value of lags and determines the number of consumers in a monitored consumer group according to the equation (1). It may scale up or down the number of consumers *drastically* based ONLY on the observed currentMetricValue. However, when adding or deleting consumers in the group, the cluster will start to rebalance and re-assign the topic's partitions for consumers [5]. During a rebalance, consumers cannot consume messages, and some partitions may be moved from one consumer to another.

After the rebalance, each consumer may be assigned a new set of partitions. If the committed offset in the new partitions is *smaller* than the offset of the latest messages that the client processed, the messages between the last processed offset and the committed offset will be processed twice. If the committed offset in the new partitions is *larger* than the offset of the latest messages that the client processed, all messages between the last processed offset and the committed offset will be missing. Consumers need to take additional time to handle the above issues. The additional time and resources needed are called *auto-scaling cost*.

The Kubernetes HPA controller determines the number of consumers based on the *current value* of lags without considering the auto-scaling cost. It may increase many consumers at

the next time interval, but these new consumers may only be created and come to be effective 30 seconds later, due to the rebalance. The fluctuation in creating/deleting consumers might not be effective and will result in added lags (queue length), which is not desirable for the operation of the Kafka application.

How ProphetStor's Federator.ai Helps

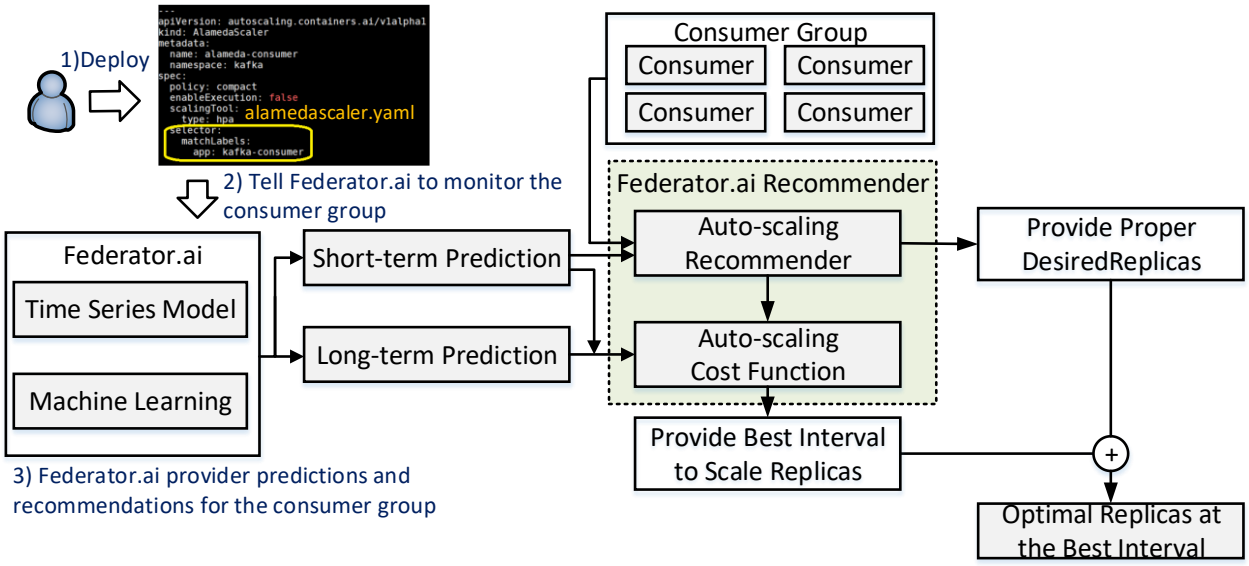

Figure 1. Federator.ai uses long-term and short-term predictions to predict and analyze application workloads, and then Federator.ai recommender provide recommendations for auto-scaling consumers and reducing replica overheads, while at the same time improved performance.

We at ProphetStor would like to bring the AI technology to the IT Operations (AIOps) in Kubernetes and think that the enterprise running Kafka on Kubernetes should have been a boring job, unlike what it is today. *Federator.ai Recommender*'s architecture for providing recommendation for Kubernetes's Autoscaler is as shown in Figure 1. ProphetStor's Federator.ai is an application-aware add-on to Kubernetes. When users deploy a custom resource definition (CRD) – alamedascaler.yaml to label a consumer group as "Kafka-consumer," Federator.ai starts to collect data about the consumer group and generate predictions. It uses time series models and other machine learning technology to analyze the collected workload data to generate long-term and short-term predictions. Federator.ai Recommender includes two components: Federator.ai Auto-scaling Recommender and Auto-scaling Cost Function. Federator.ai Auto-scaling Cost Function uses Long-term and short-term predictions to articulate the auto-scaling cost, to recommend the best auto-scaling execution time, and ultimately to optimize the operation cost. Federator.ai Auto-scaling Recommender adopts short-term predictions to predict resource utilization and can recommend the best

desiredReplicas for the scaling.

Users can use Kafka consumer API [6] or Kafka client tools to make up a consumer group by using a deployment in a Kubernetes cluster [7][8]. Each consumer in the consumer group can be connected with external services, such as MySQL or Elasticsearch, by custom Kafka connectors [9][10]. We have devised Federator.ai cost functions to recommend the best auto-scaling interval to reduce the *auto-scaling cost*. In addition, Federator.ai recommends the best number of consumers according to the total of the benefits of HPA and cost functions during the execution so that the system manager can focus on what they do best and enjoy much-reduced cost with improved performance.

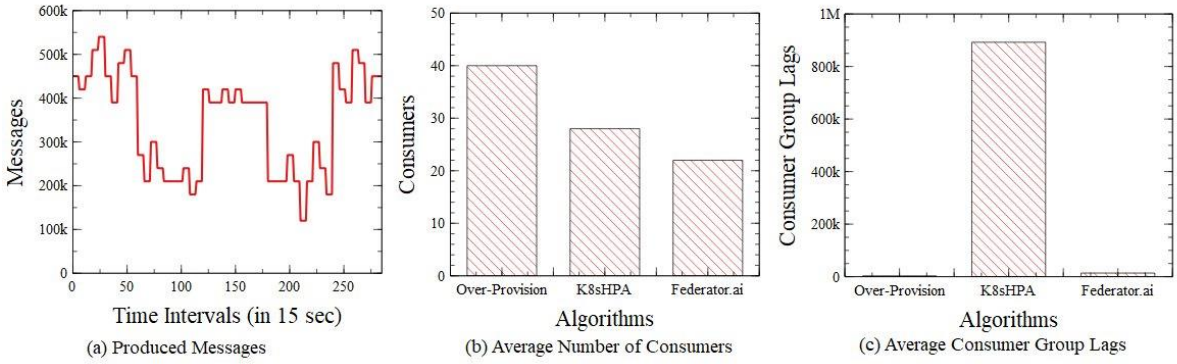

Figure 2. (a) Produced Messages, (b) the average number of consumers, and (c) average consumer group lags.

Figure 2(a) is a real workload pattern shared by Alibaba [11]. Figures 2(b) and 2(c) show the benchmark results in a Kafka cluster on Kubernetes. A producer produces messages to a topic with 40 partitions, as shown in Figure 2(a). *Over-Provision* is the scenario that users allocate a consumer group with 40 consumers to access messages for the topic. *K8sHPA* is the scenario that users dynamically assign consumers by the *K8sHPA* mechanism, and the targetAverageValue is 10,000 lags. *Federator.ai* indicates the scenario that Federator.ai Recommender exploits the prediction with the consideration of cost function to recommend the optimal desiredReplicas at the best intervals. From Figures 2(b) and 2(c), K8sHPA results in nearly 900,000 lags in related topics per interval, and each message must wait for an average of 117 ms in related topics, while Federator.ai only results in 14,000 lags and takes about 2 ms for each message. Figure 2(b) shows that Federator.ai's average number of consumers (cost of execution) being improved by 45% when compared to that of the *Over-Provision*. This result shows Federator.ai's average consumer group lags (latency) being reduced by 98.46% when compared to that of the *K8sHPA* mechanism.

Summary

Federator.ai turns monitoring into analytics. The insight gained can be used to creates the recommendation for scaling, considering the net gain of the actions. Since the recommendation is based on the future, rather than past, workloads, the result is a just-in-time fitted resource usage that can optimize the performance with minimized cost. Currently, ProphetStor is offering Federator.ai for OpenShift® and Federator.ai for NKS®, and they are fully integrated with respective platforms. They can be easily deployed with one click to individual clusters where Kafka is running. We can expect that the Kafka being accelerated and cost being reduced significantly with Federator.ai's recommendation.

References

[1]. "Horizontal Pod Autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[2]. "Autoscaling Kafka Streams applications with Kubernetes," https://blog.softwaremill.com/autoscaling-kafka-streams-applications-with-kubernetes-9aed2e37d3a0

[3]. "Kubernetes HPA Autoscaling with Kafka metrics," https://medium.com/google-cloud/kubernetes-hpa-autoscaling-with-kafka-metrics-88a671497f07

[4]. "Kafka At Scale in the Cloud," https://www.slideshare.net/ConfluentInc/kafka-at-scale-in-the-cloud

[5]. "Kafka: The Definitive Guide," https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html

[6]. "Documentation," https://kafka.apache.org/documentation/

[7]. "Apache Kafka Helm Chart," https://github.com/helm/charts/tree/master/incubator/kafka

[8]. "Strimi, Run Apache Kafka on Kubernetes and OpenShift," https://github.com/strimzi/strimzi-kafka-operator

[9]. "Setup Kafka with Debezium using Strimzi in Kubernetes," https://medium.com/@sincysebastian/setup-kafka-with-debezium-using-strimzi-in-kubernetes-efd494642585

[10]. "Apache Kafka® on Kubernetes®," https://blog.kubernauts.io/apache-kafka-on-kubernetes-4425e18daba5

[11]. "Alibaba Trace," https://github.com/alibaba/clusterdata, 2017.