UDC 004.272

S. Telenyk, V. Voinalovych, D. Smakovskyi

WEB-APPLICATION ARCHITECTURE FOR THE KUBERNETES CLUSTER AT THE GOOGLE CLOUD PLATFORM WITH HORIZONTAL AUTOMATIC SCALING

Abstract: The article is devoted to the development of the Web application architecture with the distribution of the application by components into 2 tiers connected by the messaging system, and using of load balancing by horizontal scaling in the Kubernetes cloud cluster. Queue length in the message broker is used for scaling as critirea. This approach allows to increase resource usage efficiency of the system. The relevance of the topic is due to the widespread use of various web services and web applications. When the load of web applications increases it can lead to delays or even failure of these services. Therefore, the issues of creating reliable, fault-tolerant, and scalable systems become extremely important. If the load is greater than the system or service can withstand, it may result in denial of service or termination of service. Also, the load can be unevenly distributed to services over a period of time, and therefore, even if the system has enough resources to withstand high loads, during periods of low load, these resources will not be used, resulting in problems of inefficient use of resources and overspending. The proposed system was deployed in the Google Cloud environment. The components of the server part of the Web application are grouped into 2 tiers. The microlayer components of the first layer analyze HTTP client requests and transmit messages to the components of the second layer using the Google Pub-Sub messaging system. It is proposed to make all relatively "difficult" operations on the components of the second layer. For the numerical experiment, a system was implemented using an algorithm for horizontal scaling of microservices based on the current number of messages in the queue. Load testing of the system was performed, which showed that the created system is capable of processing more than 2 times more requests for the same period of time compared to the system without scaling.

Keywords: microservices architecture, Kubernetes, horizontal scaling, Google Cloud Platform, Google Pub-Sub, Spring Boot, Java.

Introduction

The idea of cloud computing came into existence in 1961 [1], when John McCarthy suggested that someday computer computing would be done using "nationwide utilities." The ideology of cloud computing has gained popularity since 2007 due to the rapid development of communication channels and the rapidly growing needs of users. The main benefit of cloud computing – an application can use only required computer resources.

Cloud computing is usually understood as providing computer resources and capabilities in the form of an online service for some user. Thus, computing resources are

© **S. Telenyk, V. Voinalovych, D. Smakovskyi**

provided to the user in a "pure" form, and the user may not know which computers are processing his requests or running which operating system it is, etc.

The reasons for the growing popularity of cloud technologies are clear: the possibilities of their application are very diverse and allow users to save on maintenance and staff, as well as on infrastructure. Hardware can be greatly simplified when processing data and storing information in remote data centers. All these problems are almost completely translated to the service provider.

While solving the problem of increasing system load is a major concern, reducing downtime and eliminating particular failure points are just as important.

When creating reliable information systems, the most common priority is to minimize downtime and service interruptions. No matter how secure your systems and software are, there are issues that can cause programs or servers to fail.

Actuality and novelty

During high-load services design and implementation, there are two major issues involved: scalability and reliability. Most systems have an uneven load over certain intervals (Fig. 1). The system should be designed in such a way that, even during temporary load peaks, it will continue to operate reliably.

The purpose of this paper is to describe an Web-application architecture that allows automatic horizontal scaling in the Kubernetes orchestrator by using queue length information, for the efficiency increase of computing resources usage in the system.

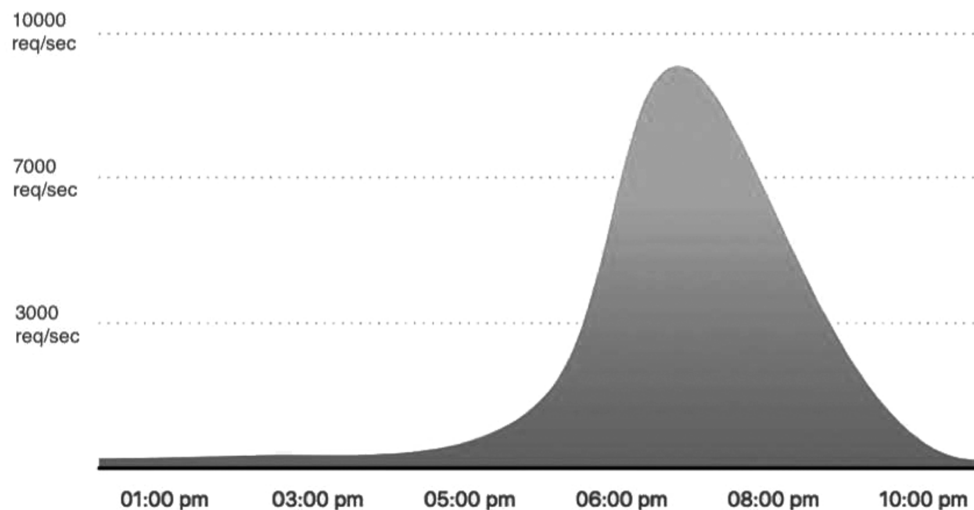

Figure 1. Request distribution based on time of day

Main part

Consider a Web-application consisting of several services that interact with HTTP. When the load on a particular component of the system is expected to increase, the system

can be scaled several times by involving the required number of instances of a particular service. However, it is not necessary to increase the instances of unloaded components of a distributed system. But if the load turned out to be higher than expected, the system may not be able to handle all requests and require additional scaling.

This system does not meet the current requirements for fault tolerance. To improve resiliency and accessibility, an approach with a queue between microservices services may be used.

This architecture makes it possible to achieve the following:

1. If one service is unavailable, the queue acts as a buffer and stores all received requests.

2. If the client generates more requests than the service can process, these requests are stored in the queue.

3. Both services are completely independent of each other.

Scaling allows fulfilling all customer requests during peak times, but there is a question of timeliness. It is impossible to constantly anticipate system load fluctuations [2], so system administrator should always follow system metrics and act on time. As a rule, for reliability, the system works with some excess resources to be able to withstand unpredictable workloads, but this approach uses the resources of virtual machines and costs extra money [3]. The usage of queue enables automation of horizontal scaling and thus optimizes costs.

The Kubernetes orchestrator and its Horizontal Pod Autoscaler (HPA) automatic horizontal scaling mechanism can be used to solve the automation scaling problem [4]. HPA changes the shape of the Kubernetes workload by automatically increasing or decreasing number of Pods when configured limit of CPU utilization or some other custom metrics is reached. Technically, HPA is a controller that is configured by HPA resource objects. HPA is implemented as a control loop that periodically checks specific metrics against configured target values and provides an API for implementing system-specific metrics (Fig. 2) [4].

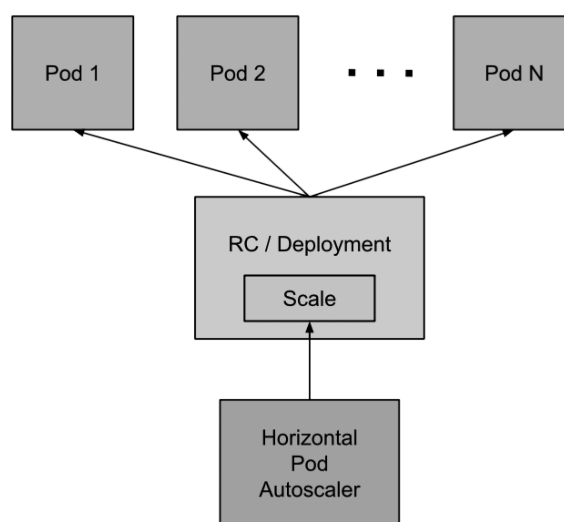

Figure 2. Horizontal scaler diagram

ISSN 1560-8956

Each iteration, the controller queries the resource utilization of the resources specified in the definition of each scaler (Fig. 3) [5]. Then, if the target utilization value is set, the controller calculates the percentage of resource utilization for each instance of the service and compares it to the target level, based on which it decides to scale up or down. If a target value is set in raw format, target value and raw resource metrics value are compared directly.

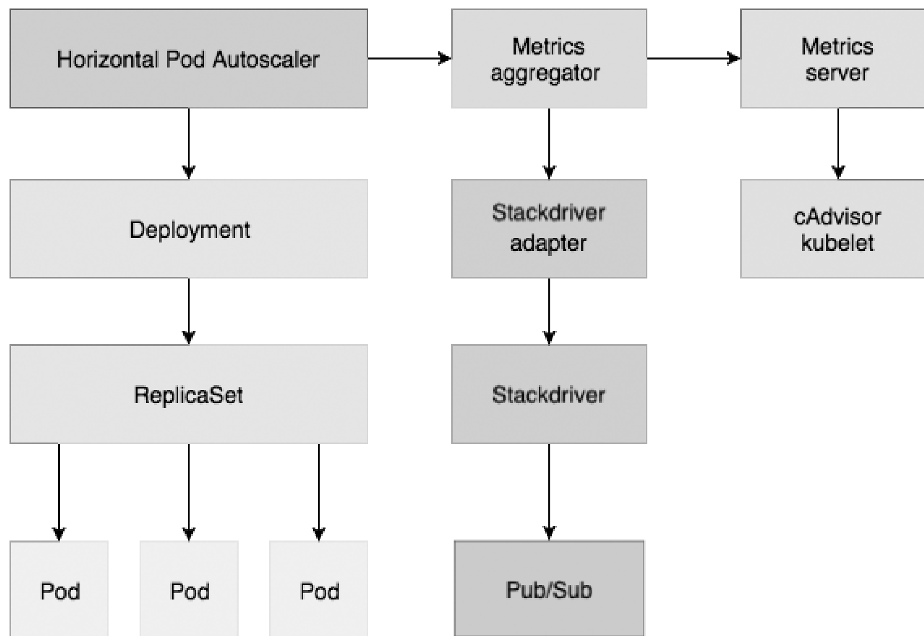

Figure 3. Diagram of component interaction for metric collection and storage

The main aim is to determine such a number of replicas that provide the closest metrics value to the target value. The autoscale algorithm is based on the next formula

$$desiredReplicas = currentReplicas \times \left(\frac{currentMetricValue}{desiredMetricValue}\right).$$

By default, Kubernetes supports automatic scaling by per-pod resources such as CPU utilization or RAM, but these metrics are not always enough. Sometimes other metrics may need to be used for greater accuracy. Providing this need in Kubernetes 1.6 was added support for using custom metrics in HPA.

If look deeper into how Kubernetes autoscaling mechanism is designed, appears that HPA is not the only part of this mechanism. HPA needs to get metrics from the outside and the component that is responsible for providing metrics to HPA is metrics registry. Metrics registry is a special part of a cluster where metrics of any kind are provided to clients such as HPA. API of the metrics registry contains three separate APIs:

- Resource Metrics API;
- Custom Metrics API;
- External Metrics API.

These APIs serve for different purposes:

- Resource Metrics API - provides per-pod metrics (CPU and memory);
- Custom Metrics API - provides custom metrics related to Kubernetes objects;
- External Metrics API - provides custom metrics not related to Kubernetes objects.

Each metric API requires a correspondent metric API server that needs to be configured to expose specific metrics through the metric API. In addition to the metric API server, a metrics collector is also required. The purpose of this collector is to collect specific metrics from sources and provide them to the metric API server (Fig. 4) [6].

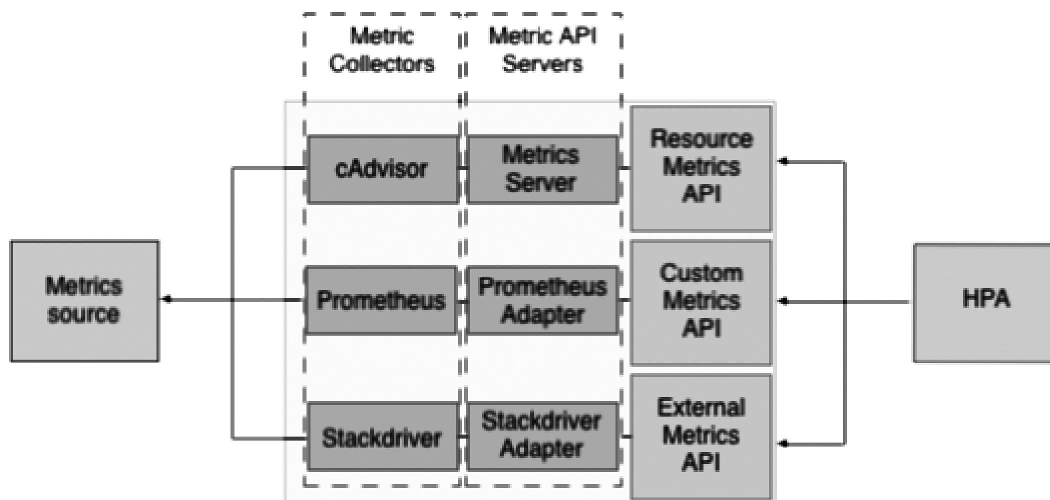

Figure 4. Diagram of metrics registry components

Different metric collectors and metric API servers can be used for different metrics API. For Resource Metric API standard configuration is cAdvisor as a metrics collector and the Metrics Server as official metric API server. Some of the choices for Custom and External Metrics API can be Prometheus or Google Cloud Stackdriver as metric collectors and their own metric API servers.

Queue length is ideally suited to optimize automatic scaling as an indicator of scaling needs. The more unprocessed messages in the queue, the more new instances of the service you need to create. If the queue is almost empty, the number of services can be reduced again.

The Architecture for the Web-application with queue length autoscaling is shown at the Fig 5. The Client sends requests to the server via HTTP. As a queue in the solution Cloud Pub/Sub service will be used. Controller Pod is a microservice that handles user input, validates input date, sends HTTP response to client and sends the data to the Subscriber Pod for processing. So we have very lightweight Controller microservice and Subscriber microservice which is responsible for the data processing. Controller microservice and Subscriber are connected with the topic of Google Pub-Sub Message Broker Cloud Service. Any another messaging service can be used with the metrics adapter for the Horizontal Pod Autoscaler. HPA takes queue length from the Google Pub-Sub and provides autoscaling for the Subscriber Microservice Pods.

 ISSN 1560-8956

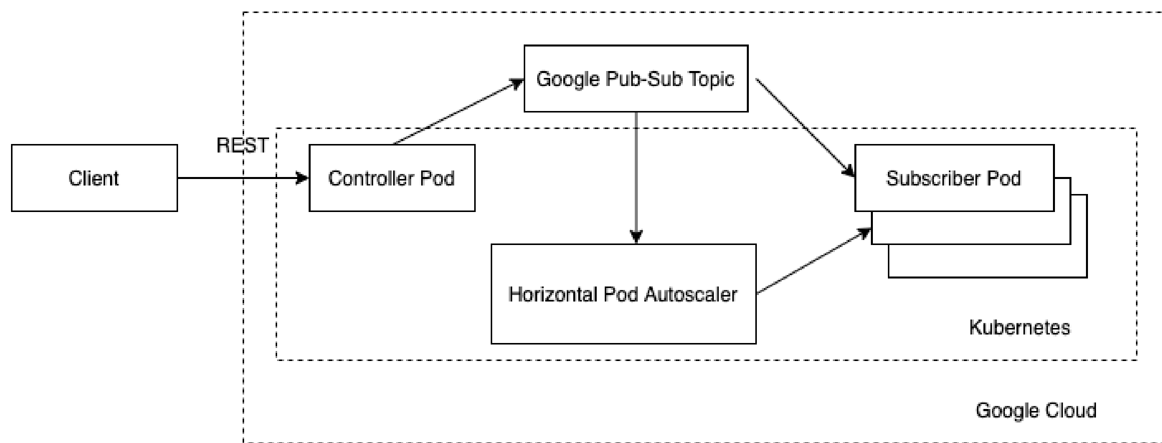

Figure 5. The Architecture for the Web-application
with queue length autoscaling

To apply horizontal scaling in Kubernetes, first of all, you need to deploy Custom Metrics Stackdriver Adapter to grant Google Kubernetes Engine access to the Stackdriver metrics. For running Custom Metrics Stackdriver Adapter you need to grant a user permissions to create required authorization roles. After that, you need to deploy the adapter to a cluster. After setting up an adapter the required metrics can be collected from Pub/Sub by metric collector provided by Google - Stackdriver, and then these metrics can be obtained by GKE and by HPA particularly through Stackdriver Adapter. For automatic scaling, the following parameters were specified:

- The minimum and maximum number of instances
- The metric name for scaling
- The target average value for the scaling metric
- The resource to scale

Figure 6 describes HPA object that is used for the solution.

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: pub-sub-hpa
spec:
  minReplicas: 1
  maxReplicas: 4
  metrics:
  - external:
      metricName: pubsub.googleapis.com|subscription|num_undelivered_messages
      metricSelector:
        matchLabels:
          resource.labels.subscription_id: testSubscription
      targetAverageValue: "2"
    type: External
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sub-app
```

Figure 6. Setting to scale by metric and target value

This example specifies the minimum and the maximum number of instances of the service, as well as the average value of the “messages” metric, depending on which scaling occurs. After HPA is created, the number of instances of the service should be equal to the minimum amount that is described in the configuration file. In this case, the minimum amount is 1. To view the events that led to the scaling, the “kubectl describe hpa” command can be used. When the load of the system is increasing it is possible to observe the number of instances of the service has grown. This means that Kubernetes auto-scaling works.

Results

Within this work, the Web-application with proposed architectural approach was built. That helps to solve the described problem more efficiently. The Web-application consists of two microservices that interact through HTTP with Web-client and interacts between these microservices via the queue – Google Pub/Sub. One of the microservices acts as a publisher – it receives requests from outside the system and sends them to the specific topic in the Pub/Sub. The second microservice acts as a subscriber – it consumes messages from the specific topic in Pub/Sub and processes them. For generating load on the publisher service it was sent 500 requests with a delay of 500 ms for each test case using JMeter. In a first test case, the system doesn't have an autoscaling mechanism for adjusting the number of the subscriber service and in a second test case, the autoscaling mechanism is present.

After tests, the following results were obtained. It was compared timestamps when first and last messages were emitted by the publisher and when first and last messages were processed by subscriber service (Tab. 1) supposing that for sending and processing requests in the services relatively short time was spent.

Table 1

Results of the requests processing with and without autoscaling

	Without autoscaling		With autoscaling	
	Publisher	Subscriber	Publisher	Subscriber
First request TS (mm:ss)	00:00	00:05	00:00	00:06
Last request TS (mm:ss)	05:57	13:11	05:57	06:02

According to the table above the delay between publishing the last message and processing it by subscriber service in the system without autoscaling was 7 minutes and 14 seconds, while in the system with autoscaling this delay was 5 seconds. Also, the number of subscriber services was increased as it was configured during the load on the system (Fig. 7).

 ISSN 1560-8956

Managed pods

Revision	Name	Status	Restarts	Created on ^
1	sub-app-b9b77679-jlppv	Running	0	Mar 18, 2020, 9:28:48 PM
1	sub-app-b9b77679-j48t6	Running	0	Mar 18, 2020, 10:28:37 PM
1	sub-app-b9b77679-j9z6z	Running	0	Mar 18, 2020, 10:28:37 PM
1	sub-app-b9b77679-5q66z	Running	0	Mar 18, 2020, 10:29:39 PM

Figure 7. Increased number of Pods of the subscriber service in Kubernetes on Google Cloud Platform

Conclusions

Proposed architectural approach with Horizontal Pod Autoscaler, Message Broker and at least two tiers of microservices allows to build systems that can dynamically adapt to the actual current system load. This approach not only improves system stability but also saves resources on servers, minimizing the number of required resources when load level is minimal. The use of custom metrics improves the accuracy of auto-scaling and provides a wide range of options for configuring a distributed system. Based on this, it becomes possible to manage resources for queues regulation and this will be the topic of future publications.

REFERENCES

1. McCarthy J. Cloud computing implements the idea of utility computing, 2008. URL: https://computinginthecloud.wordpress.com /2008/09/25/utility-cloud-computingflashback-to-1961-prof-john-mccarthy/

2. Zharikov E., Telenyk S., Bidyuk P. Adaptive Workload Forecasting in Cloud Data Centers // Journal of Grid Computing, 2019. URL: https://doi.org/10.1007/s10723-019-09501-2

3. Telenyk S., Zharikov E., Rolik O. An Integrated Approach to Cloud Data Center Resource Management // Problems of Infocommunications Science and Technology. 4th International Scientific-Practical Conference. – IEEE, 2017. – pp. 211-218. DOI:10.1109/ INFOCOMMST.2017.8246382

4. Horizontal Pod Autoscaler. URL: https://kubernetes.io/docs/tasks/run-application/ horizontal-pod-autoscale

5. Prodan S. Component interaction for metric collection and storage. URL: https://stefanprodan.com/2018/kubernetes-horizontal-pod-autoscaler-prometheus-metrics

6. Weibel. D. How to autoscale apps on Kubernetes with custom metrics — 2019. URL: https://learnk8s.io/autoscaling-apps-kubernetes