Workload performance management and elasticity

Application scaling based on performance metrics

Basically there is one performance management solution in CaaS sub-system and it is exposed to the application via the HPA API provided by Kubernetes platform. Applications can use both core metrics and custom metrics to horizontally scale themselves. The first is based on CPU and memory usage and the other uses practically every metric that the developer provides to the API Aggregator via an HTTP server.

In the following there is a short overview of the components of Performance management and elasticity sub-system of CaaS.

- **API Aggregator**: The API Aggregator is part of the K8S (Kubernetes) API server. The aggregation layer allows Kubernetes to be extended with additional APIs, beyond what is offered by the core Kubernetes APIs. The aggregation layer enables installing additional Kubernetes-style APIs in K8S cluster. These can either be pre-built, existing 3rd party solutions or user-created APIs. The role of the API aggregator in scaling is to proxy the core, and custom metrics API requests to the core, and custom metrics API handler servers registered to serve the APIs. In CaaS these are the metrics server for core: link, and Prometheus for custom: link.
- **API Server**: The Kubernetes API server validates and configures data for the API objects which include pods, services, replication controllers, and others. The API Server services REST operations and provides the frontend to the cluster's shared state through which all other components interact.
- **cAdvisor**: cAdvisor is an open source container resource usage and performance analysis agent. It is purpose-built for containers and supports Docker containers natively. In Kubernetes, cAdvisor is integrated into the Kubelet binary. cAdvisor auto-discovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics. cAdvisor also provides the overall machine usage by analyzing the 'root' container on the machine.
- **Custom Metric adapter**: Custom Metric adapter is an element to provide connection between Prometheus and API Aggregator.
- **HPA (Horizontal Pod Autoscaler)**: The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. It automatically scales the number of pods in a replication controller, deployment or replica set based on core or custom metrics.
- **Kubelet**: The Kubelet acts as a bridge between the Kubernetes master and the nodes. It manages the pods and containers running on a machine. Kubelet translates each pod into its constituent containers and fetches individual container usage statistics from cAdvisor. It then exposes the aggregated pod resource usage statistics via a REST API.
- **Prometheus**: Prometheus is an open-source software project written in Go that is used to record real-time metrics in a time series database (allowing for high dimensionality) built using a HTTP pull model, with flexible queries and real-time alerting.

The key differences between core and custom metrics are that core metrics support scraping metrics only from CPU and memory whereas custom metrics can scrape practically every kind of metrics. In the first case Kubernetes offers the metrics out of box, but in the second case users have to implement the metrics provider HTTP server.

Note that the database behind the performance management system is not persistent but uses time-series database to store metric values in both solutions.

Check core metrics in the system

Metrics APIs provided by Kubernetes can be gotten by:

```
~]$ kubectl api-versions

...

custom.metrics.k8s.io/v1beta1

...

metrics.k8s.io/v1beta1

...
```

cAdvisor automatically scrapes metrics from every active pod in the system regarding to CPU and memory usage. The state of core metrics can be requested by the following:

```
~]$ kubectl top node

NAME            CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%

172.24.16.104   1248m        62%    5710Mi                      74%

172.24.16.105   1268m        63%    5423Mi                      71%

172.24.16.107   1215m        60%    5191Mi                      68%

172.24.16.112   253m           6%     846Mi                        11%
```

With this command you can get core metrics from nodes respectively to CPU and memory usage.

The printout shows the names of nodes actually the IP addresses of nodes, the usage of CPUs in percentage and milli standard for 2 CPUs in the example furthermore memory usage in percentage and Mi (MiB) standard.

```
~]$ kubectl top pod --namespace=kube-system | grep elasticsearch
NAME CPU(cores) MEMORY(bytes)
elasticsearch-data-0 71m 1106Mi
elasticsearch-data-1 65m 1114Mi
elasticsearch-data-2 75m 1104Mi
elasticsearch-master-0 4m 1068Mi
elasticsearch-master-1 7m 1076Mi
elasticsearch-master-2 3m 1075Mi
```

Console output shows pod names and their CPU and memory consumption in the same format.

Check custom metrics in the system

In case of the usage of Custom Metrics the developer has to provide the exposition of metrics in his application in Prometheus format. There are specific libraries that can be used for creating HTTP server and Prometheus client for this purpose in Golang, Python etc.

Basic example application in Python:

```
from prometheus_client import start_http_server, Histogram
import random
import time

function_exec = Histogram('function_exec_time',
                          'Time spent processing a function',
                          ['func_name'])

def func():
    if (random.random() < 0.02):
        time.sleep(2)
        return

time.sleep(0.2)
start_http_server(9100)

while True:
    start_time = time.time()
    func()
    function_exec.labels(func_name="func").observe(time.time() - start_time)
```

This application imports http_server and Histogram metrics from Prometheus client library and exposes metrics from the func() function. Prometheus can scrape these metrics from port 9100.

```
~]$ kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1" | jq .

{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "pods/go_memstats_heap_released_bytes",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
    {
      "name": "jobs.batch/http_requests",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    }
    ]
}
```

The command result lists the custom metrics in the system, each metrics can be requested one by one for more details:

```
kubectl get –raw "/apis/custom.metrics.k8s.io/v1beta1/namespaces/kube-system/pods/*/http_requests" | jq .

{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/kube-system/pods/%2A/http_requests"
  },
  "items": [
    {
      "describedObject": {
        "kind": "Pod",
        "namespace": "kube-system",
        "name": "podinfo-bd494c88d-lmt2j",
        "apiVersion": "/v1"
      },
      "metricName": "http_requests",
      "timestamp": "2019-02-14T10:21:19Z",
      "value": "898m"
    },
    {
      "describedObject": {
        "kind": "Pod",
        "namespace": "kube-system",
        "name": "podinfo-bd494c88d-lxng7",
        "apiVersion": "/v1"
      },
      "metricName": "http_requests",
      "timestamp": "2019-02-14T10:21:19Z",
      "value": "898m"
    }
  ]
}

~]$ curl http://$(kubectl get service podinfo --namespace=kube-system -o jsonpath='{ .spec.clusterIP }'):9898
/metrics
…
http_request_duration_seconds_bucket{method="GET",path="healthz",status="200",le="0.005"} 2040
http_request_duration_seconds_bucket{method="GET",path="healthz",status="200",le="0.01"} 2040
http_request_duration_seconds_bucket{method="GET",path="healthz",status="200",le="0.025"} 2040
http_request_duration_seconds_bucket{method="GET",path="healthz",status="200",le="0.05"} 2072
http_request_duration_seconds_bucket{method="GET",path="healthz",status="200",le="0.1"} 2072
http_request_duration_seconds_bucket{method="GET",path="healthz",status="200",le="0.25"} 2072
…

# HELP http_requests_total The total number of HTTP requests.
# TYPE http_requests_total counter
http_requests_total{status="200"} 5593
…
```

This is a HTTP request with cURL, it shows the custom metrics exposed by an HTTP server of an application running in a Kubernetes pod.

Core metrics and Custom metrics examples

HPA manifest with core metrics scraping:

```
php-apache-hpa.yml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache-hpa
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: php-apache-deployment
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 50
```

In this example HPA scrapes metrics from CPU consumption of php-apache-deployment. The initial pod number is one and the maximum replica counts are five. HPA initiates pod scaling when the CPU utilization is higher than 50%. If the utilization is less than 50% HPA starts scaling down the number of pods by one.

HPA manifest with custom metrics scraping:

```
podinfo-hpa-custom.yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo
  namespace: kube-system
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: podinfo
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Pods
    pods:
      metricName: http_requests
      targetAverageValue: 10
```

In the second example HPA uses custom metrics to manage the performance. The podinfo application contains the implementation of an HTTP server which exposes the metrics in Prometheus format. The initial number of pods are two and the maximum are ten. The custom metric is the cardinality of the http requests on the HTTP server regarding to the metrics exposed.

The starter command for custom metric based HPA manifest is:

```
~]$ kubectl create -f podinfo-hpa-custom.yaml --namespace=kube-system
```

In case of starting core metrics HPA the command is the same.

After HPA start you can get information about the state of actual performance management of the system with:

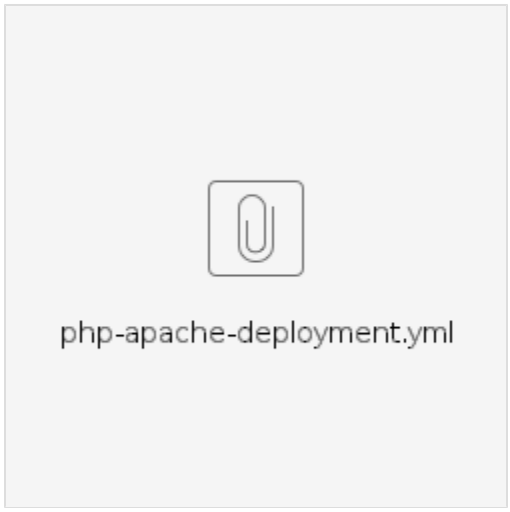
```
~]$ kubectl describe hpa podinfo --namespace=kube-system
Name:                       podinfo
Namespace:                  kube-system
Labels:                     <none>
Annotations:                <none>
CreationTimestamp:          Tue, 19 Feb 2019 10:08:21 +0100
Reference:                  Deployment/podinfo
Metrics:                    ( current / target )
  "http_requests" on pods:  901m / 10
Min replicas:               2
Max replicas:               10
Deployment pods:            2 current / 2 desired
Conditions:

  Type            Status  Reason            Message
  ----            ------  ------            -------
  AbleToScale     True    ReadyForNewScale  recommended size matches current size
  ScalingActive   True    ValidMetricFound  the HPA was able to successfully calculate a replica count from
pods metric http_requests
  ScalingLimited  True    TooFewReplicas    the desired replica count is increasing faster than the maximum
scale rate

Events:           <none>
```

Note that: HPA API supports scaling based on both core and custom metrics within the same HPA object.

Example Kubernetes manifests for testing with core metrics:



Example Kubernetes manifests for testing with custom metrics: