3rd Lab - Cloud Computing and Distributed Systems

Authors

Oulis Evangelos	<cs151051@uniwa.gr>

1.Horizontal Pod Autoscaling (HPA)

1.1 How does the Horizontal Pod Autoscaler work?

The **HPA** is implemented as a control loop, with a period controlled by the controller manager's **--horizontal-pod-autoscaler-sync-period** flag. The default value of this flag is (15 seconds).

During each **sync-period** controller queries the resource utilization against the metric specified in each HorizontalPodAutoscaler definition. The controller manager obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).

The **HPA** normally fetches metrics from series of aggregated API's (**metrics.k8s.io**, **custom.metrics.k8s.io**, and **external.metrics.k8s.io**). The **metrics.k8s.io** API is usually provided by metric-server, whitch needs to be launched separately.

Metrics-Server is a cluster-wide wide aggregator of resource usage data. It is deployed by default in clusters created by **kube-up.sh** script as Deployment object. Metrics server collects metrics from the Summary API, exposed by **Kubelet** on each node.

The autoscaler accessed corresponding scalable controllers (such as replication controllers, deployments and replicas sets) by using the scale sub-resource. Interface of scaling allows to dynamically set the number of replicas and examine each of their current states.

1.2 Algorithm Details

The main idea of **HPA** is an operation on the radio between desired metric value and current metric value.

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue)]
```

The number of replicas after scaling is depends on the division of current metric value and desired value. So, the desired number of replicas is the current metric value each time divided by desired

metric value.

Although, the below calculation is done for each metric, if any of these metrics cannot be converted into desired replica count and a scale down is suggested by the metrics which can be fetched, scaling is skipped.

This means that **HPA** is still capable of scaling up if one or more metrics give a **desiredReplicas** greater that the current value.

Finally, a configuration variable **--horizontal-pod-autoscaler-downscale-stabilization** flag can be used to configure the graduallity of scaling in order to smooth out the impact of rapidly fluctuation metric values.

1.3 Browse Horizontal Pod Autoscaler via kubectl

Like every API, **HPA** is supported in a standard way by **kubectl**. Using **kubectl create** command we can create a new **autoscaler**.

1. Using **kubectl get hpa** we can list all the autoscalings that have been configured on a Cluster, and
2. to get more details about description we can use **kubectl describe hpa**.
3. Finally we can **delete** an autoscaler using **kubectl delete hpa**.

1.4 An Easy Way to Configure Autoscaler

In addition, using **kubectl autoscale** we can create an **HPA** using a simple command.

```
kubectl autoscale rs <resourse> --min=<min replicas> -- max=<max replicas> --cpu
-percent=<cpu unilization>
```

1.5 Autoscaling During Rolling Update

Because of the **HPA** is bound to the deployment object, **HPA** does not work with rolling update using direct manipulation of replication controllers. The reason is that in rolling update creates a new replication controller, and **HPA** will not be vound to the new replication controller.

1.6 Support for Cooldown/Delay

When we are using **HPA**, it is possible that the number of replicas keeps fluctuating more often due to the dynamic nature of metrics evaluation. This sometimes called as thrashing, because of more often changes of stabilization.

A flag **--horizontal-pod-autoscaler-downscale-stabilization** can be initialize as a duration that

specifies how long the autoscaler has to wait before another downscale operation can be performed after the current one has completed.

By default **-horizontal-pod-autoscaler-downscale-stabilization** is **5** minutes (5m0s).

When tuning these parameter values, a cluster operator should be aware of the possible consequesnce. If this value is set too long, there could be complaints that the **HPA** in not responsive to workload changes. On the other hand, if the delay value is set too short, the scale of the replicas set may keep trashing.

1.7 Support for Configurable Scaling Behavior

1.7.1 Scaling Policies

One or more scaling policies can be specified in the **behavior** section of the spec. Multiple policies are specified the policy which allows the highest amount of change is the policy which is selected by default.

As an example:

```
behavior:
  scaleDown:
    policies:
    - type: Pods
      value: 4
      periodSeconds: 60
    - type: Percent
      value: 10
      periodSeconds: 60
```

periodSeconds indicates the length of time in the past for which the policy must hold true.

1.8 Stabilization Window

The stabilization window is used by the autoscaling algorithm to consider the computed desired state from the past to prevent scaling.

```
scaleDown:
  stabilizationWindowSeconds: 300
```

When the metrics indicate that the target should be scaled down the algorithm looks into previously computed desired states and uses the highest value from specified interval.

For example above we initialize a **5** minutes previously computed desired states window.

1.9 Specify a CPU Request and a CPU Limit

To specify a CPU **request** for a container, include the **resources:requests** field in the Container resource manifest, and also to specify a CPU **limit**, include **resource:limits**.

For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo            #Cluster name
  namespace: cpu-example    #Namespace Name
spec:
  containers:
  - name: cpu-demo-ctr
    image: vish/stress
    resources:
      limits:
        cpu: "1"    #CPU limit
      requests:
        cpu: "0.5"  #CPU request
    args:
    - -cpus
    - "2"           #CPU amount at start
```

1.10 Specify a Memory Request and a Memory Limit

To specify a memory **request** for a Container, include the **resources:request** field in the Container's resource manifest, and also to specify a memory **limit** include **resources:limits**.

For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example    #Namespace name
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    resources:
      limits:
        memory: "200Mi"     #Memory Limit MiB
      requests:
        memory: "100Mi"     #Memory Request MiB
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]     #Memory allocation
on start
```

2. Work on an Example

2.1 Requirements

To work this lab exercise we have to configure **kubectl** to a **cluster** in order to create a communication between kubectl and the cluster.

In my case I will use **Google Cloud Console** Platform that provides **Kubernetes** infrastructure. So, using **Google's** web interface I created a cluster and I bridge it with my local **kubectl** using the below command.

```
gcloud container clusters get-credentials goodye-world --zone us-central1-a --project
bustling-opus-262223
```

To check the version of **kubectl** enter:

```
kubectl version
```

Each node in the cluster must be at least 1 CPU. To determine that **metrics-server** is enabled we have to enter the below command,

```
kubectl get apiservices

# or
kubectl get services | grep "v1beta1.metrics.k8s.io"
```

To ensure that is enabled, a label **v1beta1.metrics.k8s.io** will be displayed.

2.2 Create Docker Image

Now we are going to create the **docker image**, that is defined by an overloaded application to highlight the performance of the cloud cluster.

Main Process

Below is presented the main process written in **Node.js** and implements a **web server**.

```
'use script';
const express = require('express');

const PORT = '8080';
const HOST = '0.0.0.0';

const app = express();

app.get("/high-load", (req, res) => {
    var os = require('os');

    let i;
    let sum = 0;

    for (i=0; i < 5000000; i++) {
        sum += Math.pow(Math.random()*Math.floor(500000), 2);
    }

    res.send('Goodbye World. Pod with id ' + os.hostname() + ' answered. The result of
random\'s 2nd powers is: ' + sum + ' .');
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

The above is a process that implements a computation logic with high load in order to run an overloaded process into my cluster. Also, in the output (**response**) includes the hostname to separate each pod's response.

Dockerfile

Below is presented the **dockerfile** in order to build the image that will run the above process on Cloud.

```
#From the official Node.js runtime as a pattern Image
FROM node:10

#Set the working directory to /app
WORKDIR /app

#Copy the current directory contents into the container at /app
COPY . /app

# Set instructions on build.
ONBUILD ADD package.json /app/
ONBUILD RUN npm install
ONBUILD ADD . /app

RUN npm install

#Make port 80 available to the world outside this container
EXPOSE 80

#Define enviroment variable
ENV NAME World

#Run Node.js when the conatiner launches
CMD ["npm", "start"]
```

The above **Dockerfile** builds the **docker image** that implements the process that presented at **9.1**.

2.3 Package.json

Below we present the congigure file that manages some labels and the command with which our application will boot.

```
{
    "name": "test1",
    "version": "1.0.1",
    "description": "Node.js Overload on Docker",
    "author": "Oulis Evangelos <cs151051@uniwa.gr",
    "main": "server.js",
    "script": {
        "start": "node server.js"
    },
    "dependencies": {
        "express": "^4.16.1"
    }
}
```

2.4 Build Docker Image

Using the below command we can configure some environment variables and **build** the docker image into my local repository.

```
export HOSTNAME=gcr.io
export PROJECT_ID=bustling-opus-262223
export IMAGE=goodbye-app
export TAG=v2

docker build -t ${HOSTNAME}/${PROJECT_ID}/${IMAGE}:${TAG} .
```

Also, we can try it running in the local system:

```
docker run -p 4000:8080 ${HOSTNAME}/${PROJECT_ID}/${IMAGE}:${TAG}
```

2.5 Push Docker Image on the Repository

After we have to push the **docker image** from the local repository to **Google** repository. Using the below command we succeed to push the docker image on the cloud, where run cluster have access.

```
docker push ${HOSTNAME}/${PROJECT_ID}/${IMAGE}:${TAG}
```

2.6 Create a Namespace

Create a **namespace** so that the resources I create in this exercise are isolated from the rest of my cluster.

Lets remind about **namespaces**.

Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. **Namespaces** are a way to divide cluster resources between multiple users.

To define a namespace, for example named **hello-cpu** we have to enter:

```
# kubectl create namespace <namespace - name>
export NAMESPACE=goodbye-cpu

kubectl create namespace ${NAMESPACE}
```

2.7 Deploy the Application and Create a Kubernete Cube

Using **kubectl** we are going to deploy the image into the namespace we created and start defined miscroservices. The configuration **yaml** file that describe the **deployment** is shown below.

Deploymnet.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: goodbye-web
  name: goodbye-web
  namespace: goodbye-cpu
spec:
  replicas: 1
  selector:
    matchLabels:
      app: goodbye-web
  template:
    metadata:
      labels:
        app: goodbye-web
    spec:
      containers:
      - name: goodbye-app
        image: gcr.io/bustling-opus-262223/goodbye-app:v2
        resources:
          requests:
            cpu: 100m
          limits:
            cpu: 500m
```

To start the deployment and apply the above **yaml** configuration we enter the below command.

```
export DEPLOYMENT=goodbye-web

kubectl create -f deployment.yaml --namespace=${NAMESPACE}
```

As we can see above configuration file, it is clear that we defined 100 (milli-cores) as the least requirement for each pod. Shortly, each **Pod** will created only when leftover resources meet the needs of the requested **CPU** resources for each **Pod**.

Secondly, there is a **limit** attribute. This attribute defines that the maximum load for each pod that created and this value on our case is 500 (milli-cores).

In order to configure **HPA** to our deployment, we have to add a defined **resources** value in order to let the **HPA** algorithm, (**1.2** chapter), make the necessary computations. Algorithm binds the value **resources.requests.cpu** from deployment as a **desired** metric value for the **HPA** algorithm.

At the above **yaml** file I mentioned these values to **resources** point.

2.8 Expose the Deployment as a Service

Using **kubectl** we are going to expose the deployment as a microservice into the namespace we created above.

```
kubectl expose deployment ${DEPLOYMENT} --type=LoadBalancer --port 80 --target-port
8080 --namespace=${NAMESPACE}
```

Now the microservice is available on the cloud and is accessed via Internet. The microservice provides a function that returns a summarization of an amount of random numbers. This function runs in /**high-load** path parameter.

2.9 Autoscaling Using Horizontal Pod Autoscaler in kubectl

We described this section widely in **1.3** and **1.4** chapters. So the commands that we will use are shown below. In this example we are going to create an Autoscaler declaratively using a **yaml** configuration file.

Autoscaling YAML Configureation

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: goodbye-web
  namespace: goodbye-cpu
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: goodbye-web
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

The above configuration I created a Horizontal Pod Autoscaler that maintains between **1** and **10** replicas as pods controlled by our **goodbye-web** deployment. We described the algorithm in **1.5** chapter. Shortly, **HPA** will increase and decrease the number of replicas through the same

deployment, to maintain an average CPU utilization across all Pods of **50%**.

To create, browse and delete an **HPA** configuration we can enter each of below commands.

```
kubectl create -f autoscaling.yaml --namespace=${NAMESPACE}

kubectl get hpa     #listing of HPA's

kubectl delete hpa goodbye-web --namespace=${NAMESPACE}     #To delete the HPA.
```

2.10 Check the Performance of HPA

To check the performance of this deployed cloud application I developed a multi-threaded application using **Java**. Shortly, this application runs a loop creating **threads**. Each thread runs a nested loop that send **HTTP GET** requests to our service.

Main Loop

```
for(int i=0; i<2000; i++) {
    new MyHttpRequest("http://34.66.26.200/high-load").start();
}
```

Nested Threaded Loop

```
@Override
    public void run() {
        try {
            for (int i=0; i<50; i++) {
                URL url_site = new URL(this.url);
                URLConnection yc = url_site.openConnection();
                BufferedReader in = new BufferedReader(
                                        new InputStreamReader(
                                        yc.getInputStream()));

                in.close();
            }
        } catch (MalformedURLException ex) {
            Logger.getLogger(MyHttpRequest.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (IOException ex) {
            Logger.getLogger(MyHttpRequest.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
```

As we can see, the above java process creates **2000** threads, each thread creates **50 HTTP** statements. So, the simultaneous **hits** against the service that we created above are **2000**.

Create Some Cases

In this section we will see how autoscaler reacts on increased load. Container already runs and using the above process we will send an infinite loop of queries to the NodeJS service.

The process' code is placed into **httprequests** folder and also a **Makefile** configuration to run the process.

Makefile

```
PACKAGE = httprequests
CLASS = HttpRequests

all:
    javac $(PACKAGE)/*.java
    java $(PACKAGE).$(CLASS)
```

Process the Result

Before we start the "Brute force" attack to our service I checked the snapshot of **HPA**, which is presented below.

```
kubectl get hpa --namespace=${NAMESPACE}

NAME          REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
goodbye-web   Deployment/goodbye-web   0%/50%    1         10        1          17h
```

Also, I get a shapshot for the deployment.

```
kubectl get deployment ${DEPLOYMENT} --namesapce=${NAMESPACE}

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
goodbye-web   10/10   10           10          19h
```

The snapshot of **HPA** at the moment of this **attack** is shown below.

```
kubectl get hpa --namespace=${NAMESPACE}


NAME          REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
goodbye-web   Deployment/goodbye-web   285%/50%   1         10        10         17h
```

And the state of each Pod, that one which is created from the **deployment** and was waiting for a request and these Pods which created from **HPA** is shown below.

```
kubect get pods --namespace=${NAMESPACE}


goodbye-web-7c9fb57c77-2v4vq   1/1     Running   0          8m2s
goodbye-web-7c9fb57c77-7dl5w   1/1     Running   0          7m16s
goodbye-web-7c9fb57c77-9wgtf   1/1     Running   0          8m2s
goodbye-web-7c9fb57c77-bnld7   1/1     Running   0          8m16s
goodbye-web-7c9fb57c77-jnl2p   1/1     Running   0          8m2s
goodbye-web-7c9fb57c77-lqjhc   1/1     Running   0          8m2s
goodbye-web-7c9fb57c77-ngccl   1/1     Running   0          8m16s
goodbye-web-7c9fb57c77-qrlnm   1/1     Running   0          8m16s
goodbye-web-7c9fb57c77-rlmnc   1/1     Running   0          7m46s
goodbye-web-7c9fb57c77-s4d6h   1/1     Running   0          17h
```

Now we can see how the load that created from the **HTTP** client affected the **CPU** usage and the state of the cluster. It is clear that the **HPA** divided this load by creating more **PODS** and controlled the load for each **POD** in order to have **CPU processing** under the **MAX** requested of each pod. This **MAX** value is defined into the **deployment** configuration file.

Some Screenshots

Google console, supports also some monitoring figures. In order to evaluate the cluster's resources I attach below some figures of **CPU**, **Network** as **packets** and as **bytes traffic**, and **Disk** usage as **bytes** and as **operations**.

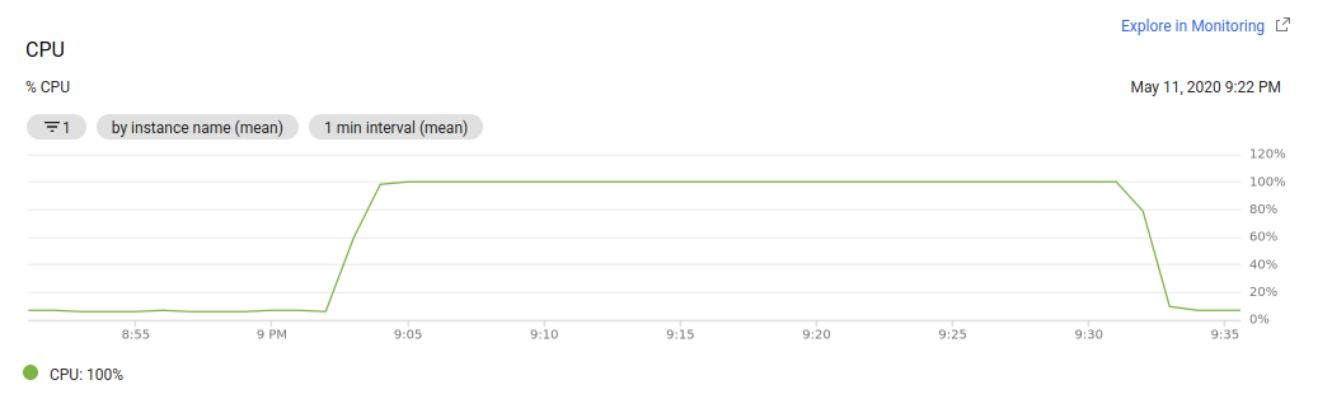

Figure 1. CPU usage.

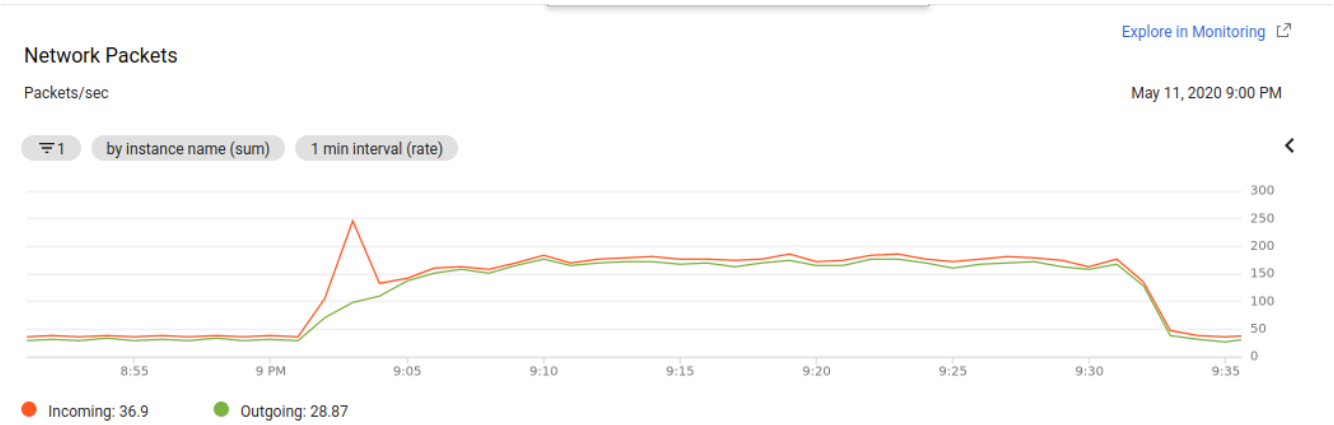

Figure 2. Network monitoring as Packet traffic (upstream & downstream).

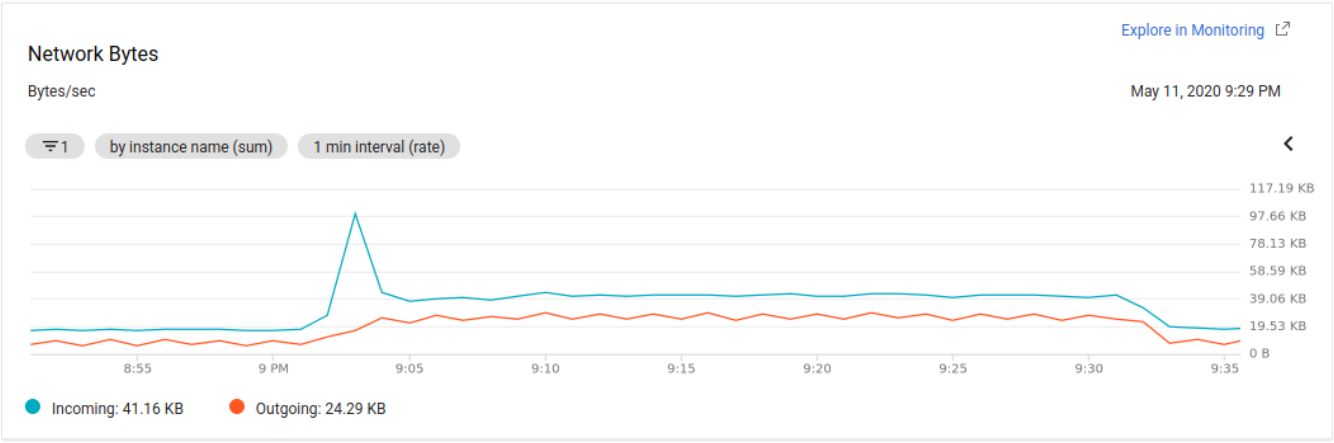

Figure 3. Network monitoring as Bytes traffic (upstream & downstream).

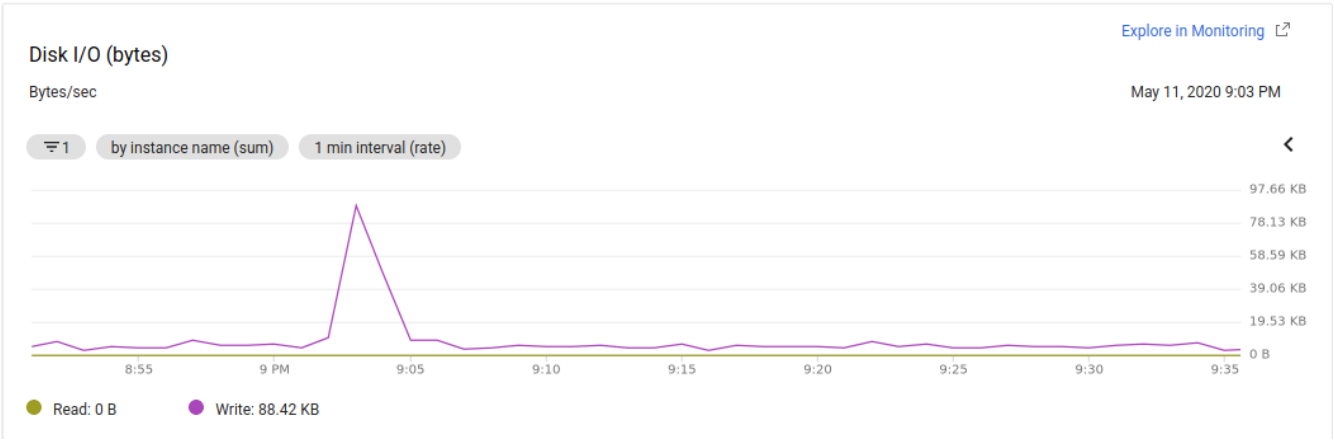

Figure 4. Disk monitoring as Bytes (read & write).

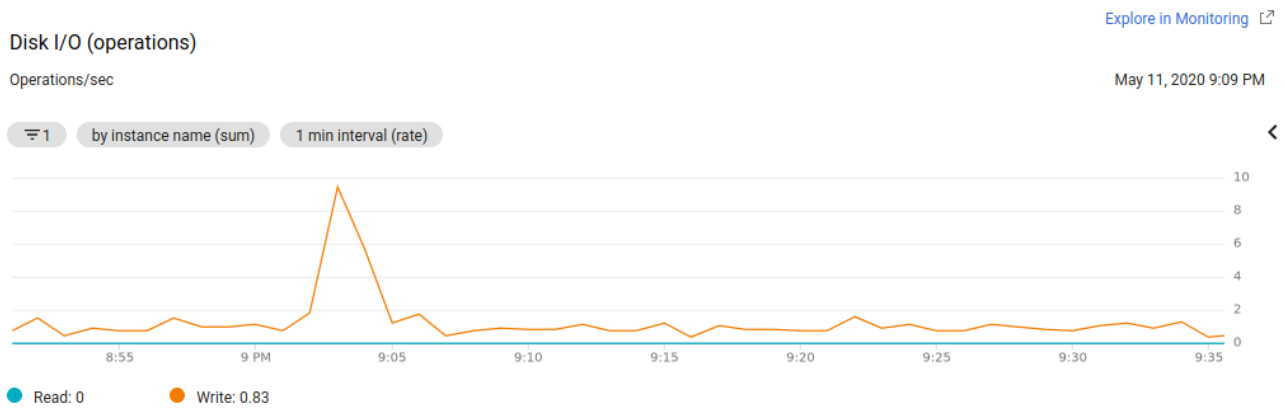

Figure 5. Disk monitoring as Operations (read & write).

Attached Files on GitHub

Finally, I attached some **code** and **configuration** files on which this report refered on as a repository on GitHub that is accessable on (repo click here).

References

1. Namespaces
2. Autoscaling
3. Google Cloud Platform Autoscaling