CS695 Assignment4

Josyula Venkata Aditya - 210050075
Ananth Krishna Kidambi - 210051002

May 3, 2024

1 Introduction

Our assignment is based on implementing FaaS(Function as a Service) using kubernetes. It is a fairly simple version of FaaS and we aim to provide the following features.

- Registration of endpoints
- Isolation of namespaces among organizations
- Registration of triggers for endpoints through a docker registry
- Autoscaling pods based on load
- Removal of inactive pods

As we go along, we will look into more of these features. Note that we are not supporting triggering function DAGs as of now, triggers can only be standalone functions without side effects, but it is possible to extend support to side effects being restricted to disk changes by using a NFS instead of a disk on each node.

2 Setup

We have setup our cluster using 3 virtual machines running on a server, one of them functioning as the control plane and the other two functioning as the worker nodes. The specs of the virtual machines are as follows:

- RAM: 4 GB (excluding swap)
- 4 CPU cores
- Operating System: Ubuntu-22.04

There is no router between the virtual machines, all of them lie on the same LAN. If you would want to reproduce the results, it is highly suggested to use physical machines/virtual machines that lie on the same LAN without any

router in between them since we have faced problems with setting up the CNI (Container Network Interface), calico in our case, in such a setup. The versions of Kubernetes and calico are as follows:

- Kubeadm version: 1.30.0
- Kubelet version: 1.30.0
- Kubectl version: 1.30.0
- Calico: Link to the yaml file we used

The node running the control plane is the main server for handling user requests in our FaaS implementation. We have not made its IP address externally available hence the FaaS service is accessible only within the private area network.

3 User API

A python process on the control plane node runs a multithreaded `Flask` server that first accepts requests from the user and then deploys containers appropriately to service the user requests. Once the request is serviced by a container, the output is returned back to the user from the `Flask` server. The format of our FaaS endpoints is `http://<IP address>:8887/<organization>/<endpoint name>`. Here, `IP address` is the IP address of the node running the `Flask` server, which is the control plane in our case.

3.1 Registering endpoints

First, we would need a way to install/manage the dependencies of the trigger function when the control plane wishes to deploy it. A docker image is a straightforward solution for such tasks and we have adopted this approach in our solution as well. An `src` folder with all the files required to run the function has to be provided, from which an image will be generated and pushed to our private docker registry running on another computer in the same private network. Note that we did not choose DockerHub for this purpose because there is a limit on the number of image pulls an IP address can make in a day, and this interferes with the process of testing our implementation. Once the image is pushed, a request has to be made at a special endpoint on the server `http://<IP-address>:8887/add_endpoint`. This request expects a json payload with the following attributes:

- `org` : This is the name of the organization. If this is a new organization, a new namespace is created for this organization and all the entities related to the organization will be put in this namespace
- `endpoint` : This is the endpoint that the organization wants to register with a FaaS trigger function.

- `image` : This is the name of the image in the docker registry which must be used to execute the trigger function
- `replicas` : This indicates the number of pod replicas to be run for this particular endpoint.

4 Implementation

We use `kubernetes` as the container orchestrator and `docker` as the container manager. In addition, a third-party CNI has to be installed to facilitate networking between the pods. After having gone through multiple failed attempts, we would suggest that if you are using `calico` for CNI, then connect the nodes using a LAN with no router in between, this makes sure that all the nodes are on the same subnet, and hence BGP used by `calico` wouldn't cause issues. `Flannel` is another choice for CNI. Also, `swap` must be disabled on the nodes and control plane for kubernetes to work correctly. We also use a private docker registry instead of DockerHub for reasons mentioned in a previous section. Also, it is suggested to have at least 4 GB of RAM and a large enough disk to avoid memory pressure and disk pressure while running the cluster.
We use python's `Flask` API to implement the central server that would first receive all the user requests. The server also uses kubernetes' python API to facilitate deployment from within the server process. We also maintain a database storing the information about the endpoints and the organizations.

- Table `organization` just stores the names of organizations
- Table `endpoints` stores the following:
  - name of the organization
  - name of the endpoint
  - IP address of the ClusterIP service
  - the image to use for deploying a container when trigger is given
  - The number of replicas of pods to use for this endpoint trigger.

Whenever an endpoint is registered, appropriate modifications are made to the tables. In addition to the files provided in the `src` folder, we add an additional file called `.interface.py` in the docker image. This runs a `Flask` server inside the container to receive the forwarded request from the central server and then calls the trigger function on the received request. It then sends the return value back to the central server in the form of a request which is then forwarded to the user.

To streamline the process of deploying at triggers, we use kubernetes deployments and services. Whenever an endpoint is registered, an associated kubernetes deployment is made with the required number of replicas, docker image url, container port(port where the flask server inside the container would be

running). We also set the default CPU limit on each deployment to `100m`, and set the default command such that it runs the `Flask` server in `.interface.py`.

Next, we create a ClusterIP kubernetes service for this deployment which will route incoming requests on the service to one of the pods associated with the deployment. Note that we aren't performing explicit load balancing in our solution and are relying on the service's capability to distribute load to all pods as a means to balance load among all pods of the deployment. Hence, whenever a request is made at a particular endpoint on the central server, the server checks the database to find the service IP corresponding to that endpoint. It then forwards the request to the service which then sends it to one of the pods. Reception of requests is similar.

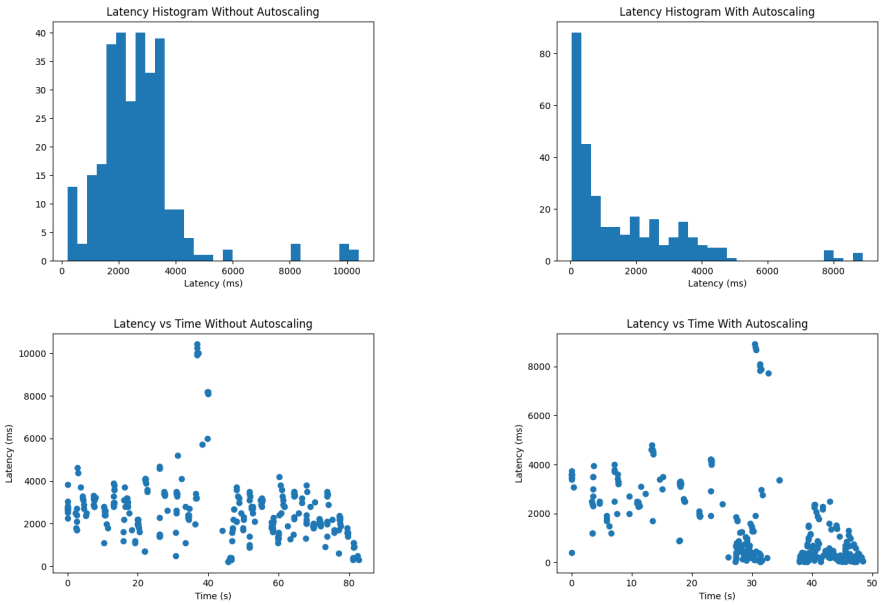
We have also implemented functionality to delete deployments that haven't been accessed for a long time to free up resources on the underlying machines. To do this, we run a separate polling thread alongside the `Flask` application which polls an in-memory dictionary which stores the last access time of a particular endpoint. If the endpoint was accessed sufficiently long ago, then we delete the deployment corresponding to that endpoint and re-deploy it when there is a next request to that endpoint.

Finally, we have incorporated horizontal autoscaling into our solution, which is a facility provided by kubernetes. For doing this, we set up the metric server. Also, note the following github issue. By default, we have set the minimum copies to 1 and maximum copies to 10. The autoscaler is deployed for each endpoint whenever it is registered.

5 Experiments

We have conducted experiments to test the latency of the system under different conditions. We have used a workload that involves numpy inversion of 100×100 matrices in the serverless function. We have generated requests using a multi-threaded python program having 10 threads. We have tested the system under the following conditions. The plots obtained for the experiments are shown below.

- Here, we run the workload for 30 iterations and 10 threads. We have tested the system with and without autoscaling. The plots obtained are shown below.



The mean and median latencies for the above experiments are as follows:

- Without autoscaling: mean = 2654.15 ms, median = 2501.53 ms
- With autoscaling: mean = 1539.94 ms, median = 768.47 ms

It can be seen that the mean and median latencies are significantly lower when autoscaling is enabled. This is because the system is able to dynamically adjust the number of pods based on the load, and hence the requests are distributed among more pods, leading to a lower latency.

- Next, we measure the cold start times - i.e. the time taken for the first request to be serviced in the worst case. This happens when the endpoint has not been accessed for time greater than the maximum idle time for the endpoint, and the deployment needs to be re-deployed. We obtained that the cold start time is around 7.71 seconds with a standard deviation of 0.477s. Here, we assume that the deployment has happened atleast once and thus the image is already present on the node. Hence, it does not account for the time taken to pull the image from the registry.

- We also measure the latencies during the beginning of a load surge and at the steady state, i.e. from a cold start and a warm start(after the autoscaler has create additional replicas). We have tested the system with 50 iterations and 10 threads.

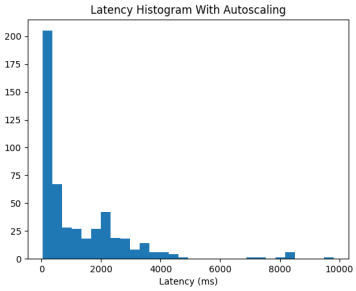


Figure 1: Latencies during the beginning of a load surge

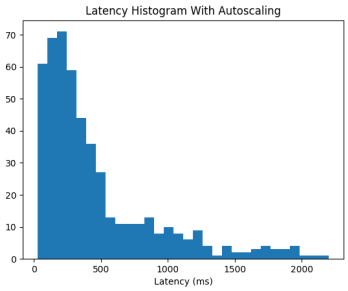


Figure 2: Latencies at the steady state

Note that once the steady state is reached, the required number of pods are already running and hence the latencies are lower compared to the beginning of the load surge. This is because the requests are distributed among more pods, leading to a lower latency.