SOFE 4790U: Distributed Systems

Date: October 9, 2022

Lab 2: Deploying a request splitting ambassador and a load balancer with Kubernetes

Group 14
Members:

Preet Patel	**100708239**
Zachary Carman	**100697844**
Philip Jasionowski	**100751888**
Rodaba Ebadi	**100708585**

Links:

Github: https://github.com/preetpatel87/Distributed-Systems-Group-14

Video Demonstrations:

- **Part 2 Deployment:** **https://drive.google.com/file/d/1-Ld1-DKLCDwTssZtVCJRTexqSH-AiiXC/view?usp=sharing**
- **Part 3 Deployment:** **https://drive.google.com/file/d/1Iak-DUOdGwDu-O-mm2fHFHhle1Iw55iD/view?usp=sharing**
- **Autoscaler Deployment:** **https://drive.google.com/file/d/1v9bdqxVTgTOraFt_4HbQgBWjDCsuJSB0/view?usp=sharing**

Learning Objectives:

In this lab, we learned how to configure and run a request splitter using Nginx and gained familiarity with the kubernetes ConfigMap tool. We gained familiarity and learned how to configure load balancer services and autoscale pods.

Discussion

Summarize the problem, the solution, and the requirements for the pattern given in part 1. Which of these requirements can be achieved by the procedures shown in parts 2 and 3?

The Gateway Routing Pattern is used to route requests to multiple different services or multiple service instances using a singular endpoint. It is useful in exposing multiple services on a single endpoint which requires routing to appropriate services based on request type, exposing multiple service instances on a single endpoint for load balancing and availability, and exposing different versions of a service on an endpoint which requires routing across versions.

The Gateway Routing Pattern was created to address the problem of the client needing to consume multiple services, instances or different service versions. The client needs to be updated when the services are added or removed. In the scenario of multiple disparate services, each service has a different API. Client needs to know about each endpoint to function, so if an API changes the client needs to be updated. Code has to be updated for both client and service. In the scenario of multiple services instances, each time an instance is added or removed, the client has to be updated. Lastly, in the scenario of the service having multiple versions, the client needs to be updated whenever there are changes to the percentage of traffic being routed to different versions.

To solve this problem of updating the client, we can place a gateway in front of the applications, services, or deployments. We have to use the application layer routing to route requests to appropriate service instances. The client only needs to know about a single endpoint and the gateway will solve the rest of the problems of any changes to the applications, services, or deployments. In the scenario of multiple disparate services, if a service is removed for a reason, the client can still continue to make requests to the gateway. Only the routing to the service changes which is handled by the gateway. Changes can be made in the backend services behind the gateway and the client requests will be routed to the appropriate service to handle the request without the need to update the client. The client does not have to be aware of any increase or decrease to the number of services. In the scenario of services having multiple versions, routing can be used to control what version of the service is used by clients. This gives flexibility to use various release strategies, whether incremental, parallel, or complete rollouts of updates. If any issues are discovered the new service can be reverted by making a configuration change at the gateway, without affecting clients.

Requirements for this pattern:

- Client needs to consume multiple services which can be accessed behind a gateway and the application can be simplified to a single endpoint.
- Requests need to be routed from externally addressable endpoints to internal virtual endpoints, such as exposing ports on a VM to cluster virtual IP addresses.
- Client needs to consume a variable number of services running in multiple or same regions for latency or availability benefits.
- Clients need to access multiple versions of the service at the same time.

For part 2 of the lab, we had multiple (two) instances of the service running separately. We needed to route the incoming requests to either one of the services for processing. We solved this requirement by adding another service, which will route the requests to the other two services depending on a configured weight for routing. Clients can consume multiple variable number of service instances using this method and the only change has to be on the route splitting service configuration. For part 3, we introduced a load balancer between 3 service instances to balance the load of each of the services. This also addresses the requirement of the client needing to consume a variable number of services for latency and availability benefits. The client does not need information about the number of services or does not need to be updated when a new service instance spins up or a service instance spins down. The client requests can be routed to running instances by the load balancer without having the need to update the client of any changes to any of the operating services.

Design

Autoscaling is another pattern that can be implemented by GKE. Your task is to configure a Yaml file that autoscaling the deployment of a given Pod.

- **Autoscaler Demonstration:** **https://drive.google.com/file/d/1v9bdqxVTgTOraFt_4HbQgBWjDCsuJSB0/view?usp=sharing**

- **YAML File:**
  Github:
https://github.com/preetpatel87/Distributed-Systems-Group-14/tree/main/Lab%202/Final

Why autoscaling is usually used?
Auto Scaling is used to gain the following benefits in an application:

- Better fault tolerance. Auto Scaling can detect if any instances failed and can launch a new instance to replace it.
- Better availability. Auto Scaling helps ensure that your application always has the right amount of capacity to handle the current traffic demand.

- Better cost management. Auto Scaling can dynamically increase and decrease capacity as needed. In cloud pay for use policy, it is one way for businesses to reduce costs.
- Automation. Auto scaling can add resources when needed.
- Reliable performance levels. Autoscaling can ensure that desired performance levels are achieved and maintained.

How autoscaling is implemented?

Autoscaling provides users an automated approach to increase or decrease computer resources depending on the current traffic demand. Without autoscaling, the resources are locked at a particular configuration and would not be able to handle spikes and operations will cost the same even if demand decreases. Autoscaling is implemented with the concept of virtualization. Users define a virtual instance with a specific capacity and predefined performance attributes, called a launch configuration. During traffic spikes, more resources are provided to the virtual application instance resource pool by the cloud from the large pools of resources of the provider. When demand drops, the extra resources are released by the virtual instance of the system that is running the application and is returned to the cloud’s resource pool automatically.

How autoscaling is different than load balancing and request splitter?

Load balancer distributes traffic evenly across all the service instances. In case of a service failure, a load balancer will simply stop routing requests to the failed instance but will not restart or remove the failed instance. It simply keeps track of the health of each running instance. On the other hand an autoscaler removes the failed instance and adds a new working instance to replace the failed service instance. A request splitter distributes requests across service instances based on a predefined configuration of the weight of each instance. It is not dynamic like the load balancer which ensures traffic is always evenly distributed. In case of service failures, the request splitter will still send requests to the service. It does not need to monitor the health of the instance, it simply distributes requests regardless. It will not try to restart the service like the auto scaler, it will need manual efforts to restart the service to make it function appropriately.