Faculty of Engineering and Applied Science

SOFE 4790U Distributed Systems

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

Group 19 - Individual Report

William Robinson

100751756

Group GitHub Link: https://github.com/sunilt4/Distributed-Systems/tree/main/Lab%202

Procedure/Video Links:

William Robinson Video 1:
https://drive.google.com/file/d/1C2OO8CEOc9Zia7oE8sbYp6wO8zqtnJ6a/view?usp=sharing

William Robinson Video 2:
https://drive.google.com/file/d/14Eruu9RVA35-jSL68Rj0FXfaQdwVn2_M/view?usp=sharing

Objective:

Learn how to configure and run a request splitter using Nginx.
Be familiar with the ConfigMap tool in Kubernetes.
Learn how to use the curl command for requesting an HTTP method.
Learn how to configure load balancer services
Get Familiar with load balancing pattern

Part 1.

Read the following document (Gateway Routing pattern - Azure Architecture Center | Microsoft Learn). Focus on the problem being solved by the pattern, how is it solved? and the requirements needed for the solution.

The problem Gateway Routing pattern handles is when a client wants to consume multiple services and multiple service instances the client must be updated. It must be updated to reflect when services are added or removed.

This was solved by placing a gateway at the front of the applications, services and/or deployments. It utilized the application 7 layer routing to route requests to the appropriate instances. This pattern was useful because the client application only is required to know about the single endpoint and communicate to this single endpoint. If a service is replaced, the client is not required to update and can continue to make requests to the single gateway and send routing changes. Client calls will be routed to any services that need to handle the client behavior.

The requirements utilized in the environment include services, endpoints and an API.

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?

In part 2, three deployments hosting webpages were deployed onto a cluster to simulate two different servers handling requests from the combined webpage. Nginx acted as an ambassador deployment to divide and split the incoming server requests. Requests were split into a 9-1 ratio specified by the service in the yaml file. This meant 90% of the requests would be directed to the regular web deployment and the remaining 10% to the experimental deployment. The external IP of the ambassador deployment was used by the user access to send requests to the deployments to then be further split based on the set ratio. This was tested by running a command in the terminal to simulate 20 user requests which was observed to be evenly split by the 9-1 ratio.

Throughout part 3 we created a load balancing deployment which distributed the load across 3 pods. These pods were connected by a readiness service which would check the readiness of each pod 5 seconds after being deployed and every 5 seconds after. This readiness tested to see if it can be used again and if so requests can be directed to the pod. When a pod is available we can execute requests based on the routing defined. Giving “dog” as a request will return the definition when a pod is ready.

Design:

Why autoscaling is usually used?

Autoscaling is extremely useful for optimizing resource utilization and cloud spending when operating a fluctuating software application that changes in usage regularly. Without autoscaling this process has to be performed manually by an operator to manually scale the resources up or down based on the usage.

How is autoscaling implemented?

Autoscaling is implemented by deploying a service to automatically scale clusters higher or lower based on demand of resources and the set utilization % on each cluster. When operating with a Horizontal Pod Autoscaler this service will deploy more or remove pods based on load on each pod. When the utilization increases more pods will be allocated to the workload and when the workload decreases pods will be taken out and load balanced.

How is autoscaling different from load balancing and request splitter?

Autoscaling differs from load balancing and request splitting because it will add more resources to an environment or remove resources all based on utilization exceeding a set limit on each pod/cluster. Load balancing balances the load across a set environment with predetermined resources. This process only distributes the load and can not add additional resources. The request splitter will split requests/messages on the incoming channel to set pods with a predetermined split % on each pod to be processed individually.