Manual Tecnico

P R OYECTO 2

Sistemas Operativos 1

Fabio Andre Sanchez Chavez
201709075

Descripcion del proyecto

Crear una arquitectura de sistema distribuida genérica que muestre estadísticas en tiempo real mediante Kubernetes y tecnologías nativas en la nube. Y proporcionar un despliegue blue/green, Canary o división de tráfico. Este proyecto será aplicado para llevar el control sobre el porcentaje de personas vacunadas contra el de COVID-19 en todo en Guatemala.

Tecnologías Utilizadas

- Kubernetes
- Docker
- Golang
- Python
- Locust
- Linkerd
- React
- JavaScript
- Redis
- Grpc
- MongoDB

Comenzando

Prerrequisitos

1. Tener conocimiento en Docker y construcción de imagenes
2. Tener cuenta en google cloud
3. Tener instalado gcloud, kubectl, linkerd, helm, locust

Instalando Docker

Comandos para instalar Docker

```
sudo apt-get update
sudo apt-get install docker.io
```

Crear imagen con un archivo Dockerfile

Ubicarse en el mismo nivel del archivo Dockerfile

```
docker build -t [usuario-dockerhub/][nombre-tag] .
```

Esto creara la imagen y la etiquetara con el tag que se desee, si se desea subir la imagen a dockerhub es necesario agregar el nombre de usuario seguido de la diagonal(/) antes del nombre que se le dara a la imagen.

Subir imagen al dockerhub

Si al momento de crear la imagen se le agrego el nombre de usuario de dockerhub en el tag solamente es necesario correr el siguiente comando.

```
docker push [usuario-dockerhub/][nombre-tag]
```

Creando el Cluster

- **Iniciar GCloud**

```
gcloud init
```

- **Crear el clúster**

```
gcloud container clusters create k8s-demo --num-nodes=1 --tags=allin,allout --
enable-legacy-authorization  --issue-client-certificate --machine-type=n1-standard-2
```

Instalar linkerd en el cluster

```
linkerd install | kubectl apply -f -
```

Creación de Ingress Nginx

- **Crear namespace para ingress**

```
kubectl create ns nginx-ingress
```

- **Actualizar repositorios de helm**

```
helm repo update
```

- **Instalar el ingress de nginx**

```
helm install nginx-ingress ingress-nginx/ingress-nginx -n nginx-ingress
```

Inyección del Ingress con Linkerd

- Inyectar los pods al ingress para linkerd

```
kubectl get deployment nginx-ingress-ingress-nginx-controller -n nginx-ingress -o
yaml | linkerd inject --ingress - | kubectl apply -f -
```

Creación de Deployments, Service, Ingresses y Function Split

- Para ello se utiliza el siguiente comando con cada archivo .yaml o se puede utilizar una archivo .yaml que contenga todas las configuraciones.

```
kubectl create -f [file.yaml]
```

Namespace

Se crea el namespace 'project' en el cual se agrupan las configuraciones.

Deployments

Se crean los siguentes despliegues:

- dummy: Despliegue de la API dummy la cual funciona como punto de division del tráfico.
- grpc-client-deployment: Despliegue blue del cliente de grpc.
- grpc-server-deployment: Despliegue para el servidor de grpc.
- redis-pub-deployment: Despliegue green del publish para redis.
- redis-sub-deployment: Despliegue para el subscribe de redis.

Services

Se crean los siguientes servicios:

- dummy: Servicio de la API dummy de división de tráfico.
- grpc-client-service: Servicio del despliegue blue para el cliente de grpc.
- grpc-server-service: Servicio del server para grpc.
- redis-pub-service: Servicio del despliegue green del publish para redis.
- redis-sub-service: Servicio para el subscrib de redis.

Ingresses

Se crean los siguientes Ingresses

- load-ingress: Controlador de ingreso para la API dummy.

TrafficSplit

- function-split: Función para el servicio dummy que divide el tráfico hacia los despliegues blue y green.

Inyección de los Despliegues

- Se deben inyectar los despliegues lo cual permite que cada uno tenga 2 pods.

```
kubectl get -n project deploy -o yaml \
| linkerd inject - \
| kubectl apply -f -
```

Blue Deployment

Grpc Client

Funciones en Go para Grpc Client

func homePage(http.ResponseWriter,* http.Request)

- Esta función es llamada al inicio por el manejador de mux para recibir una petición. ReadAll() obtiene el cuerpo del request, este se almacena como string en data y se envía a la función connect_with_grpc_server().

  reqBody, _ := ioutil.ReadAll(r.Body) data := string(reqBody) connect_with_grpc_server(data)

func connect_with_grpc_server(string)

- Primero se obtiene la dirección del servidor de grpc.

  address := fmt.Sprintf("%s:8081", os.Getenv("GRPC_SERVER_ADDRESS"))

- Se realiza la conexión utilizando la función Dial() utilizando la dirección del servidor de grpc.

  conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())

- Se crea un nuevo cliente utilizando la función NewGreeterClient() con la conexión que se acaba de crear como parámetro.

  c := pb.NewGreeterClient(conn)

- La funcion WithTimeout() se utiliza para cancelar la ejecución cuando la operación existente se haya completado.

  ctx, cancel := context.WithTimeout(context.Background(), time.Second)

- Por ultimo se hace la llamada a la función SayHello() la cual se encuentra en el servidor de grpc.

  r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})

func main()

- En esta función se crea el router con la función NewRouter().StrictSlash(true) y se lanza el manejador con la función HandleFunc() que recibe la dirección y la función a ejecutar. Por ultimo con la función ListenAndServe se permite al router escuchar en el puerto elegido.

```
router := mux.NewRouter().StrictSlash(true) router.HandleFunc("/", homePage)
log.Fatal(http.ListenAndServe(":8080", router))
```

Grpc Server

Funciones en Go para Grpc Server

```
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply, error)
```

- Esta es la función que los clientes ejecutan a través gRPC client, donde obtenemos la información enviada por los clientes gracias al parametro in en su método GetName().

```
jsonString := in.GetName()
```

- Hacemos uso de la libreria de mongo go.mongodb.org/mongo-driver/mongo para poder conectarnos al servidor de mongo que fue creado con la herramienta Mongo Atlas.

```
client, err :=
mongo.NewClient(options.Client().ApplyURI("mongodb+srv://admin:admin123@cluster0.4d
9ky.mongodb.net/testdb?retryWrites=true&w=majority"))
```

- Una vez conectados con Mongo debemos acceder a la base de de datos y a la collection deseada.

```
collection := client.Database("testdb").Collection("users")
```

- Convertimos el string a un struct de Golang.

```
var req Request json.Unmarshal([]byte(jsonString), &req)
```

- Y realizamos la inserción de la información de mongo.

```
insertResult, err := collection.InsertOne(context.TODO(), req)
```

- Como debemo tambien almancenar la información de redis, realizamos la conexión hacia nuestro servidor de redis con la ayuda de la libreria github.com/gomodule/redigo/redis

c, err := redis.Dial("tcp", "35.188.216.162:6379")

- Finalmente se realiza la inserción de la data en redis, con validación de errores

if _, err := c.Do("HMSET", redis.Args{}.Add(id).AddFlat(&req)...); err != nil { fmt.Println("Error insertando objeto en redis desde GRPC: ",err)
}

func main()

- La función main es la encargada de inicializar el servidor de gRPC

lis, err := net.Listen("tcp", port) if err != nil {
log.Fatalf("failed to listen: %v", err)
}
s := grpc.NewServer() pb.RegisterGreeterServer(s, &server{}) if err := s.Serve(lis); err != nil {
log.Fatalf("failed to serve: %v", err)
}

Green Deployment

Redis Publish

- Servicio de redis que publica a un canal y agrega los datos a la base de datos de Redis.

Funciones en Go para Redis Pub

func homePage(http.ResponseWriter,* http.Request)

- Esta función es llamada al inicio por el manejador de mux para recibir una petición. ReadAll() obtiene el cuerpo del request, este se almacena como string en data y se envía a la función transformAndStore().

reqBody, _ := ioutil.ReadAll(r.Body) data := string(reqBody) transformAndStore(data)

func transformAndStore(string)

- Conecta con el servidor de Redis con la función redis.Dial() que recibe el tipo de red y la dirección en la cual se encuentra alojado Redis.

c, err := redis.Dial("tcp", "35.188.216.162:6379")

- Se crea un struct mediante el string recibido el cual tiene formato json.

  var req Request json.Unmarshal([]byte(jsonString), &req)

- Si el struct no está vacío se asigna el id con la funcion Do() que recibe la acción a realizar y el id.

  value, _ := c.Do("GET", "id")

- Se obtiene un contador desde redis con la función Int()

  i, _ := redis.Int(c.Do("GET", "id"))

- Para que solo se almacenen 5 registros en la base de datos de Redis se hace la siguiente condición, si i es mayor a 5 se agrega 1 al contador, en caso contrario se reinicia el contador.

```
if i < 5 {
c.Do("INCR", "id")
} else { if i > 5 {
i = 1
}
c.Do("SET", "id", 1)
}
```

- Para insertar en la base de datos se utiliza la función Do() con la acción "HMSET" para que se almacene con un tipo de estructura Map. Add() agrega el identificador y AddFlat() agrega el struct que contiene el registro.

```
c.Do("HMSET", redis.Args{}.Add(id).AddFlat(&req)...);
```

func main()

- En esta función se crea el router con la función NewRouter().StrictSlash(true) y se lanza el manejador con la función HandleFunc() que recibe la dirección y la función a ejecutar.

  router := mux.NewRouter().StrictSlash(true) router.HandleFunc("/", homePage)

Redis Subscribe

- Servicio de redis que está suscrito a un canal y agrega los datos recibido por el mismo a la base de datos de Mongo.

Funciones en Go para Redis Sub

func main()

- Conecta con el servidor de Redis con la función redis.Dial() que recibe el tipo de red y la dirección en la cual se encuentra alojado Redis.

```
c, err := redis.Dial("tcp", "35.188.216.162:6379")
```

- Luego utiliza el método redis.PubSubConn de recepción convierte un mensaje enviado en tipos convenientes para usar en un cambio de tipo y lo asigna a una variable. Para suscribirse al canal se utiliza la funcion psc.Subscribe("canal1") dónde 'psc' es la variable a la cual se asignó con PubSubConn.

```
psc := redis.PubSubConn{Conn: c} psc.Subscribe("canal1")
```

- Se utiliza una sentencia for para verificar si hay mensajes recibidos y en caso de que existan se llama a la función insert_mongo() para instertar los registros a la base de datos de Mongo. Con la función Receive().(type) se obitene el objeto recibido que puede ser un mensaje, suscripción o un error.

```
for {


switch v := psc.Receive().(type) { case redis.Message:
fmt.Printf("%s: message: %s\n", v.Channel, v.Data) insert_mongo(string(v.Data))
case redis.Subscription:
fmt.Printf("%s: %s %d\n", v.Channel, v.Kind, v.Count) case error:
fmt.Println(v)

}
}
```

func insert_mongo(string)

- Conecta con el servidor de MongoDB con la función newClient() que devuelve el cliente con las funciones necesarias para insertar los registros.

client, err :=
mongo.NewClient(options.Client().ApplyURI("mongodb+srv://admin:admin123@cluster0.4d9ky.mongodb.net/testdb?retryWrites=true&w=majority"))

- Accede a la colección dentro de la base de datos con la función Database().Collection() y devuelve la colección.

  collection := client.Database("testdb").Collection("users")

- Se crea un struct mediante el string recibido el cual tiene formato json.

  var req Request json.Unmarshal([]byte(jsonString), &req)

- Si el struct no está vacío se inserta a la base de datos mediante la función InsertOne()

  insertResult, err := collection.InsertOne(context.TODO(), req)

Generación de tráfico con Locust

Archivo JSON de datos

- El archivo con formato json contiene un conjunto de registros formado por los siguientes atributos:

{
"name": "Duglas Francisco Avila Torres", "location": "Chimaltenango",
"age": 19,
"vaccine_type": "Sputnik V", ""n_dose":1"
}

Archivo de aplicación para Locust

- Este es un archivo de pytho que contiene el codigo utilizado por locust para ejecutar solicitudes ya sea de tipo GET o POST. Dentro de la clase QuickstartUser se encuentra delay y la función que ejecuta la soliciutdes.

funcion on_start

- En esta función se envían las solicitudes de tipo POST. Lo primero que hace es abrir el archivo json con la función open(), luego se carga los datos a una variable en este caso 'data' y elege un registro del array de forma aleatoria utilizando la función randint() para elegir una posición. Por último se envía el registro json con la función self.client.post() a la dirección que corresponde.

``` @task
def on_start(self): reg = ""
with open('traffic.json') as file: data = json.load(file)

```

value = randint(0, 51)
reg = data[value] self.client.post("/", json=reg)

Cloud Run

La aplicación de React estara en un contenedor para desplegar en Cloud Run, el servidor Websockets tambien estara desplegado en Cloud Run.

Frontend

- A continuacion se muestra el archivo Docker con el cual se construira la imagen, utilizando el comando "docker build --rm -t reactimage ."

FROM node:latest COPY . .
RUN npm install
CMD ["npm", "run", "dev-server"]

- (Opcional) Se puede probar la imagen para ver que realmente funcione, con el comando "docker run -it --rm -p 8080:8080 reactimage"
- Ahora vamos a publicar la imagen para que se sirva desde la nube de Google

docker tag <IMAGE_ID> gcr.io/g3sopes1/reactimage:v.0.1

- El <IMAGE_ID> es el ID que se obtiene con el comando "docker images"
- La imagén ya tiene un nuevo TAG y podemos subirla a la nube de Google

docker push gcr.io/g3sopes1/reactimage

- Si nos vamos a GCP, podremos ver la imagen en "Container Registry"
- Ahora en "Cloud Run" seguiremos los siguientes pasos:

1. Crear servicio.
2. Damos clic en "Configurar la primera revisión del servicio" y elegimos la imagen que ya subimos.
3. Vamos a "Opciones avanzadas" y en número maximo de instancias seleccionamos 3.
4. Permitimos todo el tráfico y luego seleccionamos "Permitir invocaciones sin autenticación".
5. Creamos el servicio.

- Por ultimo veremos la dirección donde esta activo el servicio.

Mapear servicio con dominio personalizado
Para poder mapear el servicio debemos contar con un dominio propio, en este caso es "g3sopes1.tk".

- En la pestaña de "Cloud Run" seleccionaremos en "MANAGE CUSTOMS DOMAINS".
- Daremos clic en "ADD MAPPING".
- Seleccionamos el servicio que queremos mapear y luego ingresamos nuestro dominio.
- Para poder verificar que el dominio nos pertenece, debemos agregar un registro TXT en Cloud DNS
- Una vez que el dominio este verificado, ingresaremos un subdominio, en nuestro caso "proyecto2".
- Por ultimo podremos ver el mapeo en https://proyecto2.g3sopes1.tk

Servidor
- A continuacion se muestra el archivo Docker con el cual se construira la imagen, utilizando el comando "docker build --rm -t nodejsimage ."

```
FROM node:12 WORKDIR /usr/src/app

COPY package*.json ./ RUN npm install
COPY . .

EXPOSE 3003

CMD [ "node","app.js" ]
```

- Para poder subir el servidor a Cloud debemos seguir los mismos pasos que para el ###Frontend y Listo!