MANUAL TECNICO

Locust

Para la simulación de envío de tráfico se utiliza Locust, ya que con esta herramienta se puede testear la concurrencia y el envío de peticiones. En este proyecto se utiliza Locust en su versión 2.12.2

- Requisitos

  pip3 install locust

  locust -V

Hay dos maneras de poder usar locust, una de ellas es con interfaz web, y la otra es directamente con línea de comandos. Para este proyecto se utiliza sin interfaz web.

Parámetros utilizados en locust:

- Locust: este hace referencia directamente a llamar al servicio de locust
- -F: este parametro indica que se debe ingresar el nombre del archivo.py
- --host: direccion url a la que se desea realizar el envío de trafico
- --headless: indica que se utiliza solo linea de comandos, sin interfaz web
- -U: el numero de usuarios
- -t: timeout

CLI Golang

- Fatih/color
  - Color.new

    Se emplea esta libreria para Go, para dar una mejor apariencia a la linea de comandos, esta libreria tiene bastantes bondades, entre las cuales se utilizaron el de establecer colores para el menu y los diversos mensajes de validaciones que se tiene en esta linea de comandos. La sintaxis es la siguiente:

    <Nombrevar> := color.new(color.<colordeseado>).Add(color.<bold|cursive>)

  - Consola.println()

    El nombre de la variable que se da en color.new, es el mismo que se utiliza para imprimir los mensajes que se desean, de la siguiente manera:

    <Nombrevar>.println(“mensaje”).

- Reconocimiento de la linea de comandos (CLI)
  - Bufio.NewReader()

    Bufio ayuda en esta cli para la entrada del comando ingresado por el usuario, ya que hace la conversión, para guardarla en una variable, la cual fácilmente se podrá manipular la entrada ingresada por el usuario.

- Sintaxis

  En el reconocimiento de sintaxis se emplea un array en el cual en la posición cero se obtendrá cada parámetro que previamente ha sido ingresado por el usuario. Este comando se separa a traves de espacios, para poder obtener cada parametro, en este caso se llama al metodo “separarComando”, que tiene la siguiente estructura:

  ```
  func separarComando(comando string) []string {
  comandoSeparado := strings.Split(comando, " ")
  return comandoSeparado
  }
  ```

  Luego de que el comando es separado se hacen las validaciones correspondientes que se pueden observar en el codigo de Golang, y asi proseguir con la comunicación a locust.

Comunicación de Golang, Locust y python:

- Golang
  - Os/exec

    Se utiliza esta librería para poder ejecutar el script de locust en el cual se definen los parametros que este llevara, y que previamente han sido ingresados por el usuario, para ello es de gran utilidad exec.Comand() ya que a traves del sistema es posible generar la accion de llamar a locust desde golang y a la vez locust llamara a python. Para este proceso se creó la funcion ejecutarComando() la cual se puede visualizar en el codigo de golang.

Python

- **Json**

  Esta librería se utiliza para poder cargar el archivo de entrada json, el cual contiene toda la información que será enviada en la simulación de concurrencia de usuarios.
- **Locust**

  Es la librería de mayor interes ya que es la que posee los metodos de comunicación con locust, estos metodos se ejecutan mediante “tasks” y se define el tipo de peticion al host, ya sea post, get, etc.

```
@task
  def PostMessage(self):
    random_data = self.lectura.pickRandom()

    if (random_data is not None and self.environment.parsed_options.my_argument > 0):
      # data_to_send = json.dumps(random_data)
       print_debug('>> MensajeDeTrafico: Enviando datos: ' + str(random_data))

       if self.environment.parsed_options.my_argument > 1:
          self.client.post("/input", json=random_data)
```

```
            self.environment.parsed_options.my_argument -= 1
        print("VALOR DE CONTADOR: ", self.environment.parsed_options.my_argument)
    else:
        print('>> MensajeDeTrafico: No hay datos para enviar')
        #cuando no hay datos para enviar, se detiene el test
        events.request_failure.fire(request_type="PostMessage", name="PostMessage",
response_time=0, exception=Exception("No hay datos para enviar"))
```

Google cloud

Instalación de Nginx e instalación Kafka por medio de strimzi

Cuando recién se crea el clúster de GCP se realiza la instalación de estas dos tecnologías necesarias para el funcionamiento del clúster de GCP, para empezar, se instala kafka-strimzi por medio de la guía que se encuentra en su página oficial, una vez instalado se procede a instalar Nginx en el clúster por medio de helm y así obtener $NGINX_INGRESS_IP el cual será de vital importancia para el servicio Ingress de GCP. Ya obtenido la ip de ingress($NGINX_INGRESS_IP) modificamos nuestro YAML 2ingress para que se acople al clúster actual.

Servicio de Ingress

Para la realización del servicio de ingress en GCP primero se tuvo que primero crear un deployment con un pod el cual contenía la api-rest-go-kafka la cual metía a la cola de Kafka(topic) la lista de los partidos que eran enviados por locus. Para esto se tenia que aplicar el YAML llamado 1api-kafka el cual creaba el deployment mas el service de loadbalancer.

Una vez creado este pod y expuesto un puerto por el servicio de loadbalancer (3000 en nuestro caso) se le aplico el servicio de ingress de kubernetes por medio del YAML llamado 2ingress.yaml que se encuentra en la carpeta perteneciente al lado del clúster de GCP.

Kafka server(producer):

Las librerías que se utilizaron fueron las siguientes:

"context"

"encoding/json"

"fmt"

"net/http"

"time"

/*

El paquete gorilla/muximplementa un enrutador de solicitudes y un despachador para hacer coincidir las solicitudes entrantes con su respectivo controlador

*/

"github.com/gorilla/mux"

github.com/segmentio/kafka-go" //escritura en la cola de Kafka

Estructuras:

- Obj2
  - Esta estructura posee un atributo llamado Lista el cual es un array de tipo Obj el cual contiene los partidos enviados por medio de locust CLI.
- Obj
  - Esta estructura cuenta con cuatro atributos los cuales son Team1,Team2,Score y Phase los cuales poseen los valores correspondientes de nombre del equipo 1 y 2 como la fase en que se jugara el partido y el marcador, estos atributos se declararon en el enunciado del proyecto.

Métodos:

- inicio(w http.ResponseWriter, r *http.Request)
  - Permite visualizar mediante la ruta / un mensaje por parte de la api para saber que todo está bien, despliega el texto con el número de grupo y curso.
- createUser(response http.ResponseWriter, request *http.Request)
  - mediante el puerto /input recibe el JSON con el cual se va a escribir a la cola de kafka mediante el método send_message. Este método transforma el JSON en []Byte
- send_message(datos []byte)
  - Crea las conexiones necesarias hacia la cola de Kafka para poder escribir en ella mediante el método WriteMessages de la librería segmentio/Kafka-go

Cola de Kafka + MongoDB

En este caso se creo el POD del numeral 3 del diagrama del proyecto usactar 2022, haciendo uso del YAML llamado 3pod que se encuentra en la carpeta de GCP. Este pod contiene al container de go-Kafka-worker y al container de gRPC-client y además de este deployment se creo un servicio de cluster-ip para conectar la aplicación del Kafka worker(app que se suscribe al topic de kafka) con el cliente de gRPC que envía los datos al gRPC server para su posterior almacenamiento en redis-cache de azure.

El conteiner de Kafka-worker al instante en que se ingresa los datos al topic de Kafka extrae los datos por medio de ReadMessage para su posterior preparación e ingreso a la base de datos de mongoDB. Una vez ingresado en mongoDB el “partido” se lo envía al gRPC cliente para su posterior procesamiento.

Explicación del funcionamiento del Kafka worker(extrae la info de los partidos, de la cola de kafka)

Kafka worker(consumer):

Las librerías que se utilizaron fueron las siguientes:

"bytes"

"context"

"encoding/json"

"fmt"

"io/ioutil"

"log"

"net/http"

"testing"

"time"

"github.com/segmentio/kafka-go" //escritura en la cola de kafka

"go.mongodb.org/mongo-driver/mongo" //guardar en la db mongo (cosmosDB)

"go.mongodb.org/mongo-driver/mongo/options" //servicios db mongo (cosmosDB)

Estructuras:

- Obj2
    - Esta estructura posee un atributo llamado Lista el cual es un array de tipo Obj el cual contiene los partidos enviados por medio de locust CLI.
- Obj
    - Esta estructura cuenta con cuatro atributos los cuales son Team1,Team2,Score y Phase los cuales poseen los valores correspondientes de nombre del equipo 1 y 2 como la fase en que se jugara el partido y el marcador, estos atributos se declararon en el enunciado del proyecto.

Métodos:

- TestCreate(t *testing.T, carro Obj)
    - Esta function escribe en la base de datos de cosmosDB los valores requeridos para los logs, estos son extraídos de la cola de Kafka.
- enviarJoel(buff []byte)
    - Envia un arreglo de bytes hacia el gRPC cliente para su posterior procesamiento.
- main()
    - Posee las variables necesarias para acceder a la cola (topic de kafka) para poder así leer mediante un for enterno cada vez que se insertan datos a la cola y para así poder

extraerlos y enviarlos a mongo como al gRPC cliente. Extrae del topic de Kafka mediante el método ReadMessage de la librería segmentio/Kafka-go.

Funciones:

- NuevoPartido(a string, b string, c string, d int) *Obj
  - Devuelve un Obj con los parámetros que se le envían como el nombre del equipo 1 y 2 la fase en que se jugara y el marcador. Se uso en la fase de testeo para almacenar en cosmosDB.
- Create(carro Obj) error
  - Metodo que inserta en la collection de mongoDB que en nuestro caso es cosmosDB. Recibe el contexto y el objeto a ingresar.
- CreateV(carro Obj) error
  - Metodo anterior al Create el cual se encarga de llamar a Create y si hay error notificarlo.
- GetCollection(collection string) *mongo.Collection
  - Se encarga de establecer la conexión a mongoDB(cosmosDB) como también de generar las claves necesarias para su posterior procesamiento.

GRPC client

GRPC es un sistema de llamada a procedimiento remoto desarrollado por Google basandose en RPC y que utiliza como transporte tecnología HTTP/2 y protocol buffers. Para poder funcionar primero se necesita crear protocol buffers con las instrucciones y procedimientos de conversión de datos que se esperan transmitir en la aplicación, posteriormente estos protofol buffers son configurador en un server gRPC y en un cliente gRPC. El cliente se encarga de recibir los datos via formato JSON y convertirlos a protol buffers para transmitirlos hacia el server gRPC y este último realice los procedimientos que han sido programados dentro del servidor. La respuesta que retorna gRPC es otro objeto protocol buffer que el cliente se encarga de decodificar y devolver como respuesta a la aplicación que este realizando la llamada.

Librearías utilizadas:

```
"context" //Para la obtención de contextos

"encoding/json" //Para el manejo de objetos JSON

"fmt" //Para impresión de datos

"strconv" //Para la conversión de datos string
"log" //Para la obtención de logs

"net/http" //Para el manejo de rutas http

"os" //Para utilizar funciones del SO
```

```
"time" //Para la creación de objetos time
"github.com/gorilla/mux" //Para el manejo de rutas http y del servidor
"google.golang.org/grpc" //Para el manejo de objetos GRPC
```

Funciones:

```
func conectar_server(wri http.ResponseWriter, req *http.Request)
```

- Método utilizado para conectar el server client con el server grpc

```
func main()
```

- Función principal que inicia el servidor con la librería mux. La única ruta que se encuentra configurada es "/" la ruta principal donde recibe todos los datos de los partidos en formato JSON.

```
func SetCors(wri http.ResponseWriter)
```

- Función encargada de configurar los headers de cors en los objetos http.ResponseWriter recibidos en cada llamada.

Protol Buffer:

- **Service getInfo:** Servicio que recibe un objeto RequestInfo desde el el grpc client y que devuelve un objeto ReplyInfo con el resultado del procedimiento.
- **Message RequestInfo:** Objeto message que guarda los datos enviados por parámetros desde el grpc client en formato message.
- **Message ReplyInfo:** Objeto message que devuelve el resultado de una operación.

La información recibida por GRPC client se configura en un struct con los siguientes datos:

```
type Informacion struct {
    Team1 string `json:"team1"`
    Team2 string `json:"team2"`
    Score string `json:"score"`
    Phase int    `json:"phase"`
}
```

La conexión con el servidor de GRCP se realiza por medio de la función dial de la librería "google.golang.org/grpc":

```
conn, err := grpc.Dial(host+":50051", grpc.WithInsecure(), grpc.WithBlock())
if err != nil {
    json.NewEncoder(wri).Encode("No se puede conectar con el server de grpc")
    log.Fatalf("No se pudo conectar con el server :c (%v)", err)
}
```

Cloud Run

Esta aplicación(front-end) fue programada en React y se implementó mediante Google Cloud Run y llama a otra API programada en Go que se encuentra en el clúster de Azure. Esta API lee directamente la información de Redis Cache y CosmosDB usando MongoDB.

Frontend ReactJS

Componentes:

- **GraficaLive:** Componente encargado de la construcción de las gráficas basadas en los resultados de los partidos y la información seleccionada en el componente SelectorPartido. Las gráficas son construidas con la librería ChartJS.
- **MongoDash:** Componente encargado de la construcción de la tabla de datos con los últimos 10 registros recuperados desde la base de datos de mongo.
- **SelectorPartido:** Componente encargado de la construcción de dropDownBox con la información de las fases y los partidos de cada fase
- **Navigation:** Componente encargado de la construcción de la barra de navegación con las rutas “/home”, “/live” y “/logs”

Linkerd

Monitoreo (Likerd-viz y grafana): Para la parte de monitoreo en GCP se instaló linkerD, linkerd-viz y grafana para poder ejecutar un dashboard el cual indica el tráfico del namespace inyectado en el clúster de GCP.

Multi-cluster: Librería de linkerd que permite conectar servicios de Kubernetes entre diferentes clusters independientes de la topología de la red. Esto se logra con la función "mirroring" en la cual se refleja la información de servicios entre los clusters.

Configuración:

1) Instalación del linkerd en la consola actual:

curl --proto '=https' --tlsv1.2 -sSfL https://run.linkerd.io/install | sh

2) Agregar el cli de linkerd al archivo .barchrc

export PATH=$PATH:/home/#ejemplo/.linkerd2/bin

3) Instalar librería step para la creación de certificados

wget https://dl.step.sm/gh-release/cli/docs-cli-install/v0.21.0/step-cli_0.21.0_amd64.deb

sudo dpkg -i step-cli_0.21.0_amd64.deb

4) Crear certificados para la conexión entre clusters

step certificate create root.linkerd.cluster.local root.crt root.key --profile root-ca --no-password --insecure

step certificate create identity.linkerd.cluster.local issuer.crt issuer.key --profile intermediate-ca --not-after 8760h --no-password --insecure --ca root.crt --ca-key root.key

5) Instalar linkerd CRDs en ambos clusters

linkerd install --crds | kubectl --context=google apply -f -

linkerd install --crds | kubectl --context=azure apply -f -

6) Instalar los certificados creados con step en cada cluster

linkerd install \

--identity-trust-anchors-file root.crt \

--identity-issuer-certificate-file issuer.crt \

--identity-issuer-key-file issuer.key \

| kubectl --context=google apply -f -

linkerd install \

--identity-trust-anchors-file root.crt \

--identity-issuer-certificate-file issuer.crt \

--identity-issuer-key-file issuer.key \

| kubectl --context=azure apply -f -

7) Instalar linkerd-viz en ambos clusters

linkerd --context=google viz install | kubectl --context=google apply -f -

linkerd --context=azure viz install | kubectl --context=azure apply -f -

8) Instalar linkerd-multicluster en ambos clusters

linkerd --context=google multicluster install | kubectl --context=google apply -f -

linkerd --context=azure multicluster install | kubectl --context=azure apply -f -

9) Verificar que todas las dependencias de linkerd se encuentre correctamente instaladas en ambos clusters

linkerd check --context google

linkerd check --context azure

10) Conectar los clusters via linkerd-multicluster

linkerd --context=google multicluster link --cluster-name google | kubectl --context=azure apply -f -

linkerd --context=azure multicluster link --cluster-name azure | kubectl --context=google apply -f -

11) Inyectar linkerd en el cluster deseado para crear un reflejo

kubectl get deploy #nombreDeploy -o yaml | linkerd inject - | kubectl apply --context #nombreContexto -f -

12) Etiquetar el servicio del reflejo creado

kubectl --context=#nombreContexto label svc #nombreServicio mirror.linkerd.io/exported=true

13) Inyectar linkerd en el pod del cluster que va a utilizar el servicio del reflejo

kubectl get deploy #nombreDeploy -o yaml | linkerd inject - | kubectl apply --context #nombreOtroContexto -f -

AZURE

gRPC server

Es el servidor de sistema de llamada a procedimiento remoto utilizado para guardar información en la base de datos de Redis enviada desde el client gRCP montado en el cluster de google y conectado por medio de linkerd multicluster.

Librearías utilizadas:

```
"context" //Para la obtención de contextos
"encoding/json" //Para el manejo de objetos JSON
"fmt" //Para impresión de datos
"strconv" //Para la conversión de datos string
"log" //Para la obtención de logs
"net/http" //Para el manejo de rutas http
"os" //Para utilizar funciones del SO
"time" //Para la creación de objetos time
"github.com/gorilla/mux" //Para el manejo de rutas http y del servidor
"google.golang.org/grpc" //Para el manejo de objetos GRPC
```

Funciones:

```
func almacenar_datos(t1 string, t2 string, field string, phase string)
```

- Función que guarda los datos de un partido en la base de datos de redis.

```
func main()
```

- Función principal que inicia el servidor de GRCP con la función net.Listen() y que crear el servidor con la función grpc.NewServer()

```
func (s *server) ReturnInfo(ctx context.Context, in *pb.RequestInfo)
(*pb.ReplyInfo, error)
```

- Función que recibe un objeto tipo RequestInfo (configurado en el protoc buffer) y que lanza la función almacenar_datos() para guardar datos en la base de datos redis.

Redis-cache Se implemento una base de datos en Azure por medio del servicio de redis-cache(servicio NO perteneciente a la capa gratuita) para así poder utilizar para el proyecto de usactar los servicios de una base de datos no SQL en el apartado referente a los partidos a los cuales se les puede apreciar el porcentaje de votos(pronósticos/ quiniela) en la pestaña de index del front-end realizado por REACT.

CosmosDB Se implemento una base de datos en Azure por medio del servicio de CosmosDB(servicio perteneciente a la capa gratuita) para así poder utilizar para el proyecto de usactar los servicios de una base de datos no SQL en la pestaña de logs del front-end realizado por REACT. Aquí se mostrarán los últimos 10 partidos que se han ingresado en mongoDB.

GoApi

Librearias utilizadas:

```
    "github.com/go-redis/redis/v9" //Libreria para manejar redis

    "github.com/gorilla/mux" //Libreria para la gestión del servidor web

    "github.com/joho/godotenv" //Librearia para la gestión de variables de
entorno

    "go.mongodb.org/mongo-driver/mongo" //Libreria para manejar mongo

    "go.mongodb.org/mongo-driver/mongo/options" //Libreria para manejar mongo

    "context" //Para la obtención de contextos

    "encoding/json" //Para el manejo de objetos JSON

    "fmt" //Para impresión de datos

    "strconv" //Para la conversión de datos string
    "log" //Para la obtención de logs

    "net/http" //Para el manejo de rutas http

    "os" //Para utilizar funciones del SO

    "time" //Para la creación de objetos time

    "github.com/gorilla/mux" //Para el manejo de rutas http y del servidor
```

Funciones:

```
func main()
```

- Función principal que inicializa las conexiones con redis, mongo y el servidor web. Las rutas utilizadas son:
    - “/getLogsMongo”: Retorna los últimos 10 registros desde mongo.
    - “/getTotalMongo” Retorna el total de registros en mongo.
    - “/getDataFases” Retorna la información de partidos por fases.
    - “/getDataPartidos” Retorna estadísticas de un partido seleccionado.

```
func GetLogsMongo(client *mongo.Client) http.HandlerFunc
```

- Función que retorna los últimos 10 registros desde mongo. Utilizada en la ruta “/getLogsMongo”

```
func GetTotalMongo(client *mongo.Client) http.HandlerFunc
```

- Función que retorna el total de registros en mongo. Utilizada en la ruta "/getTotalMongo"

```
func SetCors(wri http.ResponseWriter)
```

- Función utilizada para configurar los headers de cors en el objeto http.ResponseWriter enviado.

```
func GetAllMatches(client *redis.Client, ctx context.Context) http.HandlerFunc
```

- Función utilizada para retornar todos los partidos registrados en redis.

```
func GetCounters(client *redis.Client, ctx context.Context) http.HandlerFunc
```

- Función utilizada para retornar todos los datos del partido seleccionado y registrado en redis. Utilizada en la ruta "/getDataPartidos".

```
func GetFase(fase string, client *redis.Client, ctx context.Context) []string
```

- Función utilizada para retornar toda la información de todos los partidos registrados en redis. Utilizada en la ruta "/getDataFases".

Linkerd

- **Monitoreo (Likerd-viz y grafana):** Para la parte de monitoreo en GCP se instaló linkerD, linkerd-viz y grafana para poder ejecutar un dashboard el cual indica el tráfico del namespace inyectado en el clúster de GCP.

CREACIÓN DE DEPLOYS EN GCP

1)creacion de cluster

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

2)conectarnos al cluster

kubectl get ns // para probar q si estas conectado en el cluster xd

3) instalar Kafka por medio de strimzi

//no hacer lo de minikube(1er comando) y Send and receive messages, despues del wait de 300s ya esta instalado kafka en el cluster (quickstart strimzi-kafka)

https://strimzi.io/quickstarts/

4)chequear que se halla instalado kafka en el cluster

kubectl get ns

kubectl get pods -n kafka

kubectl get deployments -n kafka

kubectl get services -n kafka

kubectl get nodes

5)instalacion de nginx

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

helm repo update

helm install nginx-ingress ingress-nginx/ingress-nginx

kubectl get deployment nginx-ingress-ingress-nginx-controller

kubectl get service nginx-ingress-ingress-nginx-controller

kubectl get service nginx-ingress-ingress-nginx-controller

export NGINX_INGRESS_IP=$(kubectl get service nginx-ingress-ingress-nginx-controller -ojson | jq -r '.status.loadBalancer.ingress[].ip')

echo $NGINX_INGRESS_IP

//guardar la ip de $NGINX_INGRESS_IP sera de utilidad al realizar el servicio de ingress

6)aplicar yaml 1api-kafka.yaml para instalar el kafka_server(escribe en kafka)

7)ya ingresado el kafka_server y su loadbalancer le agregamos el servicio de ingress por medio del comando

kubectl apply -f 2ingress.yaml

8)instalar lo de multi-cluster (realizar instalaciones de azure y aplicar yamls)

9)aplicar yaml (3pod)

10)linkerD+grafana (MONITOREO)

curl -sL http://run.linkerd.io/install | sh

export PATH=$PATH:/home/g201700644_2s_2022/.linkerd2/bin

linkerd version

linkerd check --pre //todo ok! sino F :(

linkerd install --crds | kubectl apply -f -

linkerd install | kubectl apply -f -

linkerd check //todo ok sino F :(

linkerd viz install | kubectl apply -f -

linkerd check //todo ok sino F :(

11)ahora instalar grafana

helm repo add grafana https://grafana.github.io/helm-charts

helm install grafana -n grafana --create-namespace grafana/grafana \

-f https://raw.githubusercontent.com/linkerd/linkerd2/main/grafana/values.yaml

linkerd viz install --set grafana.url=grafana.grafana:3000 \

| kubectl apply -f -

linkerd check

kubectl get ns //debe de aparecer el ns grafana

12)ahora toca inyectar

kubectl -n kafka get deploy -o yaml | linkerd inject - | kubectl apply -f -

13)Conectarse al dashboard de linkerD

linkerd viz dashboard & //ahora si abrir pag web

linkerd dashboard --port 8781 //si falla el de arriba usar este