UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

ESCUELA DE CIENCIAS Y SISTEMAS

SISTEMAS OPERATIVOS 1 SECCIÓN N

ING. SERGIO ARNALDO MENDEZ AGUILAR

AUX. ALVARO NORBERTO GARCÍA

AUX. SERGIO ALFONSO FERRER GARCÍA

PRIMER SEMESTRE 2025

PROYECTO 2

Tweets del Clima

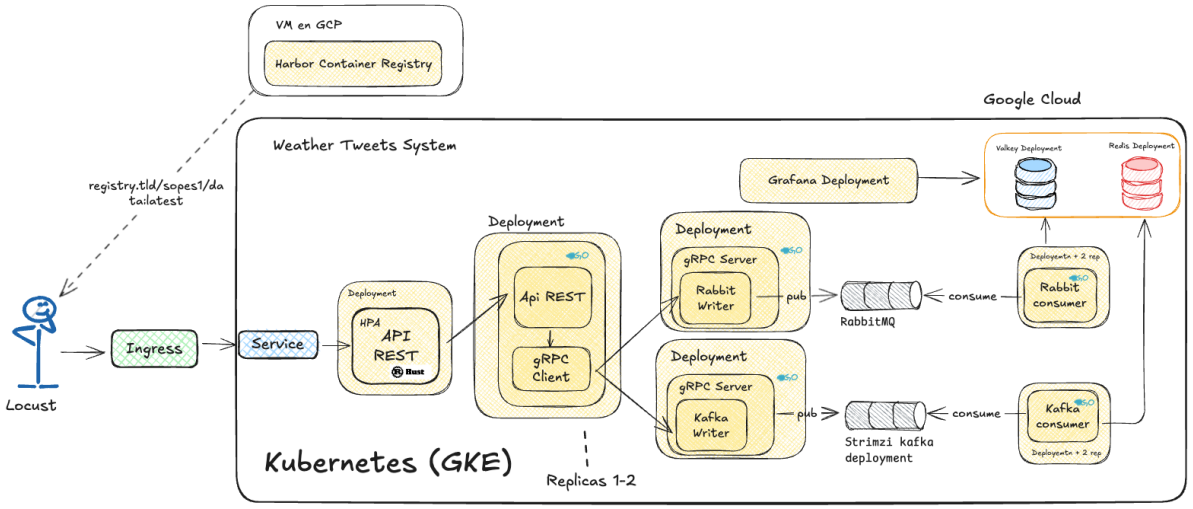
Objetivos

- Administrar una arquitectura en la nube utilizando Kubernetes en Google Cloud Platform (GCP).
- Utilizar el lenguaje de programación Golang y Rust, maximizando su concurrencia y aprovechando sus librerías.
- Crear y desplegar contenedores en un Container Registry y utilizarlo como un medio de almacenamiento.
- Entender el funcionamiento de un message broker utilizando Kafka y RabbitMQ.
- Crear un sistema de alta concurrencia de manejo de mensajes.

Introducción

Este proyecto tiene como propósito aplicar los conocimientos adquiridos en las unidades 1 y 2 mediante la implementación de una arquitectura en Google Cloud Platform (GCP) utilizando Google Kubernetes Engine (GKE). El objetivo es construir una arquitectura de sistema distribuido genérico que muestre los tuits sobre el clima mundial. Esto se procesa mediante una arquitectura conceptual escalable. Este proyecto pretende mostrar la concurrencia de tuits en el sistema.

Arquitectura



Locust

Locust es una herramienta para la generación de tráfico hecha con Python. Se instalará localmente y generará tráfico que será enviado al Ingress (puerta de acceso) de la arquitectura desplegada en Kubernetes.

Notas:

- La estructura JSON que se debe de manejar para enviar como *body* en las peticiones *HTTP* es la siguiente:
    - *Description* describe como esta el clima (“Esta lloviendo”)
    - *Country* corresponde al país (“GT”)
    - *Weather* corresponde al tipo de clima reportado y puede ser
        - *Lluvioso*
        - *Nubloso*
        - *Soleado*

```
[
    {
        "description":"Its raining",
        "country":"USA",
        "weather":"rainy"
    },
    {
        "description":"Cloudy",
        "country":"Guatemala",
        "weather":"cloudy"
    }
]
```

- El dominio expuesto por el Ingress Controller (NGINX) será : *http://IP.nip.io/input*

Sugerencias:

- Como mínimo manejar arreglos de 10,000 objetos para tener una mejor visualización.
- Locust debe al menos utilizar 10 usuarios concurrentes para hacer un total de 10,000 peticiones.

Deployments de Rust

El tráfico desde el Ingress se redirige a una API REST desarrollada en Rust. Esta API REST recibe las peticiones HTTP de Locust y las envía a un Deployment de Go. El Deployment de Rust debe ser capaz de soportar alta carga y escalará horizontalmente usando HPA.

Sugerencias:

- Utilizar HPA (Horizontal Pod Autoscaler), con un minimo de 1 réplica y máximo 3.
- Configurar el HPA para activarse cuando el uso de CPU sobrepase el 30% (ajustable según la carga).

Deployments de Go

Go Deployment 1: API REST y gRPC Client (Entrada de Tráfico): Este deployment es la puerta de entrada desde el Deployment de Rust. Recibe las peticiones HTTP desde la API REST en Rust a través de un servicio. Internamente, este deployment actúa como un client gRPC. Por cada petición recibida, el servidor gRPC invoca dos funciones: una para publicar un mensaje en una cola de RabbitMQ y otra para publicar en una cola de Kafka.

Go Deployments (2 y 3): gRPC Server y Writers (Publicación a Message Brokers): Estos deployments actúan como un cliente gRPC. Se conecta al servidor gRPC del Go Deployment 1. Estos deployments contienen funciones principales (con diferente implementación): una se encarga de publicar mensajes en RabbitMQ y la otra en Kafka. Ambas funciones reciben las peticiones del client gRPC y las envían al message broker correspondiente.

Deberá realizar pruebas con una 1 y 2 réplicas ambos deployments, para analizr el rendimiento y obtener conclusiones.

Sugerencias:

- En el Go Deployment 1, manejar dos contenedores por pod: uno para la API REST y otro para el gRPC Server.
- En el Go Deployment 2 y 3, usar un único contenedor por pod, conteniendo las funciones mencionadas.
- Si se usan 2 réplicas por Deployment, asegurar que las peticiones no se dupliquen en el procesamiento (controlar la lógica de publicación).
- Utilizar el mismo topic o cola para las pruebas, por ejemplo, "message".

Deployments de kafka y RabbitMQ

Apache Kafka es una plataforma de streaming distribuida de código abierto, diseñada para gestionar flujos de datos en tiempo real de forma rápida. Kafka es un message broker con características importantes como Publisher y Subscriber, esenciales para esta arquitectura. RabbitMQ es similar a Kafka pero con diferentes características, también un message broker.

Estos message brokers (Kafka y RabbitMQ) almacenarán y distribuirán los mensajes publicados por el Deployment de Go 2. El objetivo es comparar el rendimiento de Kafka y RabbitMQ para esta carga de trabajo, identificando cuál maneja mejor el volumen de mensajes y es más eficiente.

Deployments de consumidores

Se requieren dos deployments de consumidores:

Kafka Consumer Deployment: Consumirá los mensajes de Kafka. Almacenará los valores extraídos de los mensajes en Redis.

RabbitMQ Consumer Deployment: Consumirá los mensajes de RabbitMQ. Almacenará los valores extraídos de los mensajes en Valkey.

Valkey y Redis son bases de datos en memoria que se utilizarán para almacenar los datos consumidos. Analizar y explicar cuál es mejor o más adecuado para este caso, y si se debe usar HPA o 2 réplicas para los deployments de Redis y Valkey.

Sugerencias:

- Utilizar goroutines en los consumidores para mejorar el rendimiento del procesamiento de mensajes.
- Al usar dos réplicas de consumidores, evitar la duplicación de información procesada (implementar lógica para evitar procesamiento doble).

Deployments Redis y Valkey

Se crearán dos deployments separados: uno para Redis y otro para Valkey. Ambos almacenarán la información procesada por los consumidores (de Kafka y RabbitMQ respectivamente). Estos datos almacenados serán consultados por el Deployment de Grafana para visualización.

Sugerencias:

- Asegurar la persistencia de datos para los dos deployments.

Deployment Grafana

Este deployment utiliza Grafana para visualizar los datos almacenados en Valkey y Redis. Grafana consultará la información en Redis y Valkey y la procesará para mostrarla en un dashboard visual.

Sugerencias:

- Utilizar tablas de tipo hash en Valkey y Redis para almacenar los contadores de los paises y el total de los mensajes.
- Para instalar grafana en el Cluster de Kubernetes se recomienda utilizar Helm.

Harbor

Al desarrollar el código, este debe ser containerizado (usando Docker) y publicado en un Container Registry. Se implementará Harbor para publicar todas las imágenes Docker de los componentes (Rust, Go, Consumidores, etc.). Por ejemplo, la imagen del Deployment de Rust se construirá con Docker, se subirá a Harbor, y Kubernetes (GKE) la descargará de Harbor para desplegar el Deployment de Rust en el clúster. **Harbor estará alojado en una VM de GCP, fuera del Cluster de Kubernetes.**

Descarga archivo de entrada desde el registry como un OCI Artifact.

Documentación

Documentación de los deployments y una breve explicación con ejemplos.

Responder a las siguientes preguntas:

- ¿Cómo funciona Kafka?
- ¿Cómo difiere Valkey de Redis?
- ¿Es mejor gRPC que HTTP?
- ¿Hubo una mejora al utilizar dos replicas en los deployments de API REST y gRPC? Justifique su respuesta.
- Para los consumidores, ¿Qué utilizó y por qué?

Sugerencias generales

Nube y sistema
Se deben crear las propias imágenes utilizando Docker y un Cluster de Kubernetes en Google.

Git + Github
Git será el versionador de código utilizado para llevar el control del código del proyecto.

Namespaces
Utilizar namespaces en Kuberenetes para mantener organizado los objetos del proyecto.

Balanceador de Carga
Utilizar NGINX Ingress Controller para exponer las APIs.

Requisitos Mínimos

- Cluster de Kubernetes en GCP.

Restricciones

- El proyecto se realizará de forma individual.
- La información que ingrese será generada por Locust.
- Utilizar GKE.
- **No habrá prorroga.**

Github

- El código fuente debe de ser gestionado por un repositorio privado de Github
- Crea una Carpeta en el repositorio llamado: Proyecto2
- Agregar a los auxiliares al repositorio de GitHub: **AlvaroG13191704** y **SergioFerrer9**

Calificación

- Al momento de la calificación se verificará la última versión publicada en el repositorio de GitHub
- Cualquier copia parcial o total tendrán nota de 0 puntos y serán reportadas al catedrático y a la Escuela de Ciencias y Sistemas.
- Si el proyecto se envía después de la fecha límite, se aplicará una penalización del 25% en la puntuación asignada cada 12 horas después de la fecha límite. Esto significa que, por cada período de 12 horas de retraso, se reducirá un 25% de los puntos totales posibles. La penalización continuará acumulándose hasta que el trabajo alcance un retraso de 48 horas (2 días). Después de este punto, si el trabajo se envía, la puntuación asignada será de 0 puntos.

Entregables

- La entrega se debe realizar antes de las 23:59 del 02 de mayo de 2025.
- La forma de entrega es mediante UEDI y Classroom subiendo el enlace del repositorio.
- Manual técnico en formato Mark Down o PDF.