UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESCUELA DE CIENCIAS Y SISTEMAS
SISTEMAS OPERATIVOS 1
SEGUNDO SEMESTRE 2022
ING. SERGIO ARNALDO MENDEZ AGUILAR
AUX. CARLOS DAVID RAMIREZ ALTAN

PROYECTO FASE 3

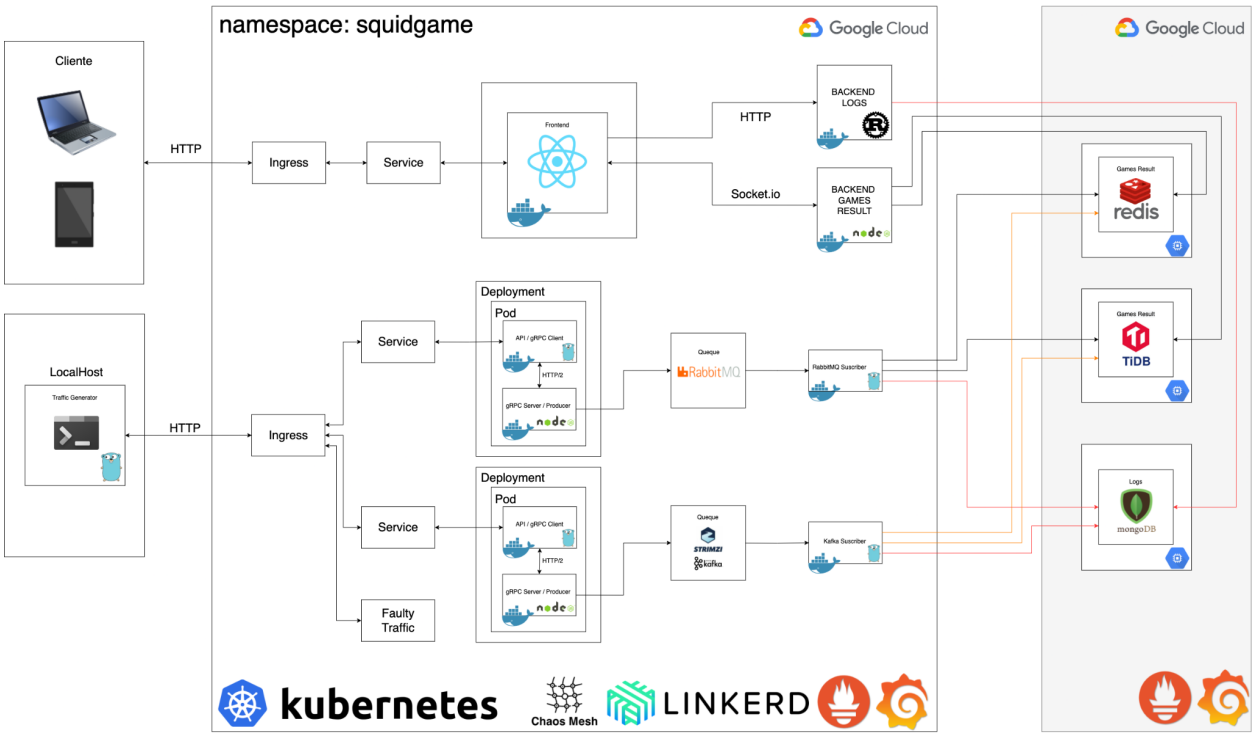
Se solicita construir un sistema genérico de arquitectura distribuida que muestre estadísticas en tiempo real utilizando Kubernetes y service mesh como Linkerd y otras tecnologías Cloud Native. En la última parte se utilizará una service mesh para dividir el tráfico. Adicionalmente, se añadirá Chaos Mesh para implementar Chaos Engineering.

Este proyecto se aplicará a la visualización de los resultados de juegos implementados por los estudiantes.

OBJETIVOS

- Comprender la teoría de la concurrencia y paralelismo para desarrollar sistemas distribuidos.
- Experimentar y probar tecnologías Cloud Native que ayudan a desarrollar sistemas distribuidos modernos.
- Diseñar estrategias de sistemas distribuidos para mejorar la respuesta de alta concurrencia.
- Monitorear el procesamiento distribuido utilizando tecnologías asociadas a la observabilidad y la telemetría.
- Implementar contenedores y orquestadores en sistemas distribuidos.
- Medir la confiabilidad y el rendimiento en un sistema de alta disponibilidad.
- Implementar Chaos Engineering.

ARQUITECTURA



Explicación:

Ingress recibe el tráfico y se redirige a una API que escribe en una cola. Antes de eso, recibe el número de jugadores y elige aleatoriamente un juego de algoritmo, para elegir el ganador del juego actual, luego escribe los datos necesarios en la cola. El Worker de Go Queue leerá estos datos primero, luego se escribirán en las bases de datos, Redis para los datos en tiempo real en los paneles y MongoDB para los registros de transacciones.

Juegos:

Los juegos serán algoritmos sencillos implementados por los estudiantes estos algoritmos deben estar implementados en el servidor de gRCP de manera que al recibir los datos del juego en este servidor se seleccionara al azar uno de los juegos implementados por el estudiante de manera que al finalizar el juego y tener un ganador se procede a mandar los resultados del juego a la cola (Queue) correspondiente.

Por ejemplo, un juego tendrá las siguientes reglas:

Genera 10 números

Elija al azar un número como ganador y el otro perderá el juego.

Nota: **Se deberán implementar como mínimo 5 juegos diferentes** sin embargo si el estudiante desea implementar más de 5 está bien.

PRIMERA PARTE (GENERADOR DE TRÁFICO CON GO)

Esta parte consiste en la creación de una herramienta que genera tráfico utilizando Go como lenguaje de programación empleando Goroutines y canales. El tráfico será recibido por un balanceador de carga, se utilizará el sitio nip.io para generar una URL pública, por ejemplo:

193.60.11.13.nip.io, este dominio se expone usando un ingress controller y su balanceador de carga.

GENERADOR GO

Esta aplicación está escrita en Go, usando Goroutines y canales. La sintaxis del CLI será:

rungame --gamename "1 | Game1 ; 2 | Game2" --players 30 --rungames 30000 --concurrence 10 --timeout 3m

Esta es la funcionalidad:

gamename: Tiene la descripción de los juegos, usando el formato GAME_NUMBER | GAME_NAME

players: Número de jugadores de cada juego

rungames: Número de veces para ejecutar los juegos

concurrence: Solicitud simultánea a la API para ejecutar los juegos

timeout Si el tiempo restante es mayor que este valor, el comando se detendrá

Ejemplo:

La funcionalidad del generador de tráfico es generar endpoints durante el tiempo que este esté ejecutando definido por rungames o en su defecto por el timeout por ejemplo al correr el siguiente comando en el CLI del generador de

tráfico **rungame --gamename "1 Random | 2 Ruleta | 5 Last " --players 30 --rungames 30000 --concurrence 10 --timeout 3m** pasaría lo siguiente:

--gamename "1 Random | 2 Ruleta | 5 Last" esta parte del comando quiere decir que voy a usar el juego 1, 2 y 5 que tengo implementados en mi servidor de gRPC.

--players 30 esta parte del comando quiere decir cual es el máximo de jugadores que puedo tener por juego por ejemplo puedo tener un ramdom hasta 30 de manera que para un juego puede salir que solo van a participar 6 jugadores y en el siguiente 25 y así sucesivamente.

--rungames 30000 esto quiere decir que se van a generar como máximo 30000 endpoints para que se ejecuten los juegos ya que si se llega al timeout puede que no se llegue a este máximo.

--concurrence 10 mediante las go rutinas se va a tener simultáneamente 10 llamadas a las apis con los endpoints que se generen.

--timeout 3m el generador de tráfico funcionará durante 3 minutos y luego de estos 3 minutos, detendrá su funcionamiento.

Generación de endpoints:

Para generar los endpoints se utiliza la información del comando que se ejecutó en el cli para este caso tenemos que vamos a usar los juegos 1, 2 y 5 y un máximo de 30 jugadores por lo que se podrían generar combinaciones como las siguientes:

193.60.11.13.nip.io/game/1/gamename/Random/players/12

193.60.11.13.nip.io/game/1/gamename/Random/players/25

193.60.11.13.nip.io/game/2/gamename/Ruleta/players/14

193.60.11.13.nip.io/game/1/gamename/Random/players/4

193.60.11.13.nip.io/game/5/gamename/Last/players/30

Una vez que se generó un endpoint se realiza el request al servidor y una vez que la api recibe la información en la api haciendo uso gRCP se pasa esta información del juego al Servidor de gRCP para que ejecute el algoritmo del juego y se encuentre al ganador y una vez que se finaliza el juego se pasa la información a la cola ya sea Kafka, RabbitMQ o pubSub para luego ser consumido estos resultados por el Worker de Go el cual será el encargado de guardar en las bases de datos dicha información.

SEGUNDA PARTE (DOCKER, KUBERNETES Y BALANCEADORES DE CARGA)

Esta parte contiene el uso de Git, Docker, la instalación del clúster de Kubernetes y la configuración de los balanceadores de carga.

GIT:

Será la herramienta para gestionar las versiones del código del proyecto. Se utilizará como herramienta para que los estudiantes desarrollen colaborativamente el proyecto. Los estudiantes deben tener un repositorio remoto donde se presentará el proyecto, se recomienda utilizar Github o Gitlab.

DOCKER:

Se utilizará para empaquetar la aplicación en contenedores, donde se sugiere utilizar técnicas distroless para crear imágenes pequeñas si es posible. Docker será la herramienta para crear un entorno local de pruebas antes de desplegar las imágenes en el clúster de Kubernetes.

KUBERNETES:

Kubernetes será el encargado de la orquestación de los contenedores. Antes de que el cliente genere el tráfico, el proyecto implementará un clúster de Kubernetes que se utilizará para desplegar distintos objetos:

- Ingress controllers: para exponer distintas partes de la aplicación.
- Deployments y services: para desplegar y comunicar distintas secciones de la aplicación.
- Pods: si es necesario, pero es común utilizar otros objetos con un nivel más alto de aplicación como los deployments. Se sugiere utilizar un

namespace separado llamado project, ya que es una buena práctica para organizar toda la aplicación.

NAMESPACE:

Es una buena práctica para organizar toda la aplicación. En nuestro caso se debe tener el namespace **squidgame**.

BALANCEADORES DE CARGA:

Se debe configurar un balanceador de carga de capa 7 (como Kubernetes Ingress) en el clúster de Kubernetes utilizando Helm o Kubectl. Este balanceador expondrá la aplicación al mundo exterior. Para este proyecto se utilizará nginx-ingress.

INGRESO:

El objetivo es comparar el tiempo de respuesta y el desempeño de las distintas rutas, la primera utilizando Kafka Strimzi como broker, la segunda usando RabbitMQ y la tercera Google Google PubSub. Toda la información de entrada pasará a través del ingress controller.

PRIMERA RUTA (RabbitMQ):

- Generador de Trafico
- Load Balancer
- API / gRPC Client (Go)
- grPC Server (NodeJS) / Producer RabbitMQ Queue
- RabbitMQ
- RabbitMQ Subscriber
- Escribir en las bases de datos NoSQL (Redis,Tidis y MongoDB)

SEGUNDA RUTA (kafka):

- Generador de Trafico
- Load balancer
- API / gRPC Client (NodeJS)
- grPC Server (Go) / Producer Kafka Queue
- Kafka
- Kafka Subscriber

- Escribir en las bases de datos NoSQL (Redis,Tidis y MongoDB)

Nota: Se desea implementar el escalado automático vertical y horizontal, corrutinas y subprocesos de acuerdo con la naturaleza del servicio a implementar. Esta implementación es abierta, pero debe justificarse en el contexto de las mejores prácticas.

TRAFFIC SPLITTING:

Para implementar Traffic Splitting, el proyecto utilizará Linkerd para implementar esta función con la idea de que el tráfico divide el 33% del tráfico en la primera ruta y el otro 33% en la segunda ruta y así sucesivamente. Para implementar esto, Linkerd usa un servicio Dummy que puede ser la copia del servicio de una de las rutas, y para esta funcionalidad debe usarse NGINX. En este momento es la opción estable y probada para este proyecto.

Pruebas de faulty traffic deseadas:

- Queue #1 100%
- Queue #2 100%
- Queue #1 50%, faulty traffic 50%
- Queue #2 50%, faulty traffic 50%
- Queue #1 50%, Queue #2 50%

Considere responder las siguientes preguntas:

- Cómo funcionan las métricas de oro, cómo puedes interpretar estas 5 pruebas de faulty traffic, usando como base los gráficos y métricas que muestra el tablero de Linkerd Grafana.
- Menciona al menos 3 patrones de comportamiento que hayas descubierto.

TERCERA PARTE (RPC AND BROKERS)

La principal idea es crear una manera de escribir datos en bases de datos NoSQL con alto desempeño utilizando la comunicación por RPC, message brokers y las colas de mensajería. La meta es comparar el desempeño de las rutas, Consulte el diagrama de arquitectura.

gRPC: Es un framework de RPC de alto desempeño que puede ser ejecutado en cualquier ambiente, usado principalmente para conectar servicios de backend.

Kafka: Es un sistema de colas de alta disponibilidad para la transmisión de datos en aplicación en tiempo real.

RabbitMQ: Es un modo de sistema de cola de la vieja escuela para funcionar como intermediario o procesamiento de tareas.

Debe responderse las siguientes preguntas:

- ¿Qué sistema de mensajería es más rápido?
- ¿Cuántos recursos utiliza cada sistema? (Basándose en los resultados que muestra el Dashboard de Linkerd)
- ¿Cuáles son las ventajas y desventajas de cada sistema?
- ¿Cuál es el mejor sistema?

Nota: gRCP debe ser implementado en 2 lenguajes de programación diferente go y nodejs.

CUARTA PARTE (NOSQL DATABASES)

El proyecto está basado en la estructura de la arquitectura de Instagram, debido a la naturaleza del sistema y a la ausencia de datos estructurados es mejor utilizar bases de datos NoSQL. MongoDB podría utilizarse para almacenar datos persistentes y Redis para implementar contadores y caché para mostrar datos y analíticos en tiempo real.

• MongoDB: es una base de datos NoSQL documental que almacena la información utilizando el formato de datos JSON. Un ejemplo de log seria:

```
{
    "request_number":30001, //Que numero de ejecucion es por ejemplo antes de este juego se jugaron 30000
    "game":1, //Id del juego
    "gamename":"Game1", //Nombre del juego
    "winner":"001", //Jugador que gano el juego
    "players":20 //Cantidad de jugadores que participaron en el juego
    "worker":"RabbitMQ" //El worker hizo la insercion del log en MongoDB
}
```

• Redis y Tidis: son bases de datos NoSQL de clave-valor que implementa distintos tipos de estructuras de datos como listas, conjuntos, conjuntos ordenados, etc. Estas bases de datos almacenarán los resultados de los juegos y dicha información la determinará el estudiante ya que en base a los reportes que debe generar el en frontend debe analizar qué información es necesaria almacenar.

Estas bases de datos serán instaladas en una instancia que será accesible en la VPC del clúster de Kubernetes.

Deben responderse la pregunta:

- ¿Cuál de las dos bases (Redis y Tidis) se desempeña mejor y por qué?

QUINTA PARTE (PÁGINA WEB)

Debe crearse un sitio web que muestre en tiempo real los datos insertados, de los resultados de los juegos el sitio web debe ser desarrollado utilizando React.

Backend Logs: Debe de implementar un backend para acceder a los logs almacenados en MongoDB, este debe ser desarrollado utilizando Rust como lenguaje de programación y este debe ser consumido por el frontend por medio de peticiones http.

Backend Games Results: Debe de implementar un backend para acceder a los registros en las bases de datos Redis, Tidis y este debe ser desarrollado utilizando NodeJS como lenguaje de programación y este debe ser consumido por el frontend por medio de sockets.

página principal debe mostrar los siguientes reportes:

Reportes con datos de MongoDB:

- Tabla con los logs almacenados.
- Gráfica del top 3 de juegos.
- Gráfica que compara a los 2 Subscribers de go (la cantidad de inserciones que hizo cada Subscriber).

Reportes con datos de Redis y Tidis:

- Últimos 10 juegos.
- Los 10 mejores jugadores.
- Estadísticas del jugador en tiempo real.

NOTA:

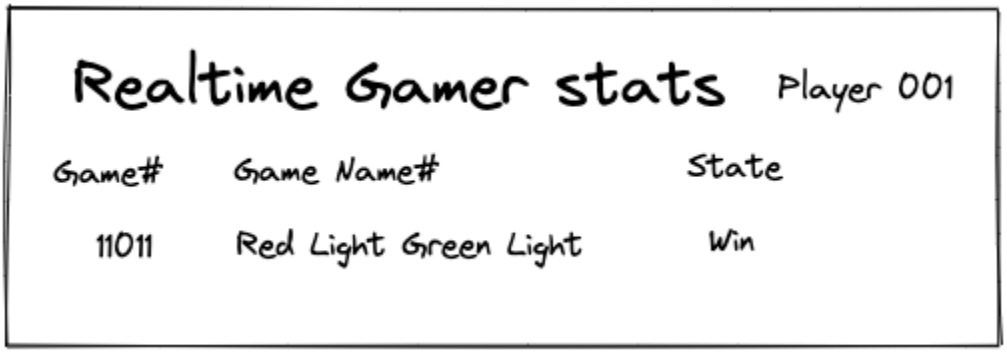
- **La forma en que se muestran los reportes queda a discreción del estudiante sin embargo debe tener por separado los reportes de Redis, los reportes de Tidis y los Reportes de MongoDB.**

- **Redis y Tidis almacenan la misma información esto para poder comparar cual base de datos se desempeña mejor.**

Página Principal:

Estadísticas en tiempo real de X jugador:

Para esta vista se debe poder elegir el jugador del que se desean ver sus estadísticas en tiempo real.



OBSERVABILITY AND MONITORING

El estudiante tiene que decidir los lugares para implementar la observabilidad y las métricas doradas usando Linkerd.

Linkerd: El proyecto tiene que implementar la observabilidad en la red y las respuestas asociadas a los diferentes pods o implementaciones implementadas en el proyecto. En este proyecto, el proyecto implementa una supervisión en tiempo real de las métricas doradas.

Prometheus: The project have to implement monitoring for the state of the services using Prometheus, for example you can use Prometheus to monitor NoSQL Databases and visualize the information using Grafana

CHAOS ENGINEERING

El alumno tiene que implementar faulty traffic al sistema y matar componentes del clúster, al mismo tiempo muestra el comportamiento del caos en el clúster.

- Linkerd: Usar Linkerd para la generación de faulty traffic.
- Chaos Mesh: Se implementará para experimentar con: Slow Network, Pod Kill, Pod Failure y Kernel Fail.

La meta es monitorear el comportamiento del sistema mientras el caos está en progreso.

Con Linkerd, prepare los siguientes experimentos:

- tráfico defectuoso según la sección de división del tráfico.

Con Chaos mesh, prepare los siguientes experimentos:

- Pod kill
- Pod failure
- Container kill
- Network Emulation (Netem) Chaos
- DNS Chaos

Deben responderse las siguientes preguntas:

- ¿Cómo cada experimento se refleja en el gráfico de Linkerd, qué sucede?
- ¿Qué diferencia tienen los experimentos?
- ¿Cuál de todos los experimentos es el más dañino?

OBSERVACIONES

- Todo debe implementarse utilizando el lenguaje o la herramienta especificada.

- El uso de kubernetes es obligatorio para desplegar todo lo solicitado.
- Deben escribir un manual técnico explicando todos los componentes del proyecto y respondiendo las preguntas de cada sección.
- Deben escribir un manual de usuario y un manual técnico.
- Las copias detectadas tendrán una nota de cero puntos y serán reportadas a la Escuela de Ciencias y Sistemas.
- No se aceptarán entregas después de la fecha y hora especificada.
- Se deberá agregar al usuario **racarlosdavid** al repositorio del proyecto.

ENTREGABLES

- Link del repositorio, debe incluir todo el código fuente con los archivos de manifiesto o cualquier archivo adicional de configuración.
- Manuales.

FECHA DE ENTREGA

4 de mayo antes de las 23:59 a través de UEDi. No hay prórrogas.