Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Sistemas operativos 1
Segunde Semestre 2021
Ing. Sergio Méndez
Aux. Luis Leones Aguilar
Aux. Carlos David Ramírez

Manual técnico

PROYECTO #2

Grupo 7
Luis Danniel Ernesto Castellanos Galindo – 201902238
Benaventi Bernal Fuentes – 201021212

Introducción

Este manual describe la arquitectura y metodología utilizada para la implementación del proyecto #2 del curso de Sistemas Operativos 1.

Se darán a conocer los requerimientos, la estructura y uso de tecnologías que fueron requeridas para la construcción total del sistema, en el desarrollo de servidores y aplicaciones conectados mediante contenedores y un orquestador.

El proyectó consistió en la construcción de 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 utilizó una service mesh para dividir el tráfico. Adicionalmente, se añadió Chaos Mesh para implementar Chaos Engineering.

Requerimientos del sistema

REQUERIMIENTOS MÍNIMOS DE SOFTWARE

1. Debe poseer un navegador web actualizado para poder visualizar la página y las estadísticas correctamente.
2. Se recomienda ejecutar el generador de juegos en un entorno Linux.

REQUERIMIENTOS MÍNIMOS DE HARDWARE

1. Procesador: Core
2. Memoria RAM: mínimo 2 GB
3. Disco duro: que sea funcional
4. Tarjeta de red LAN y/o Wireless
5. Equipo
6. Equipo, teclado, monitor, cursor, dispositivo móvil

Herramientas utilizadas para el desarrollo

GO:

Go es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C, que intenta ser dinámico como Python y con el rendimiento de C o C++. Ha sido desarrollado por Google y sus diseñadores iniciales fueron Robert Griesemer, Rob Pike y Ken Thompson.

Ingress:

Kubernetes Ingress es un objeto de API que proporciona reglas de enrutamiento para administrar el acceso de usuarios externos a los servicios en un clúster de Kubernetes, generalmente a través de HTTPS / HTTP. Con Ingress, puede configurar fácilmente reglas para enrutar el tráfico sin crear un grupo de Load Balancers o exponer cada servicio en el nodo.

Docker:

Docker es un sistema operativo para contenedores. De manera similar a cómo una máquina virtual virtualiza (elimina la necesidad de administrar directamente) el hardware del servidor, los contenedores virtualizan el sistema operativo de un servidor.

Kubernetes:

Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. Kubernetes facilita la automatización y la configuración declarativa. Tiene un ecosistema grande y en rápido crecimiento. Soporta diferentes entornos para la ejecución de contenedores, incluido Docker.

gRPC:

gRPC es un sistema de llamada a procedimiento remoto de código abierto desarrollado inicialmente en Google. Utiliza como transporte HTTP/2 y Protocol Buffers como lenguaje de descripción de interfaz.

Kafka:

Apache Kafka es una plataforma distribuida de transmisión de datos que permite publicar, almacenar y procesar flujos de registros, así como suscribirse a ellos, de forma inmediata. Está diseñada para administrar los flujos de datos de varias fuentes y distribuirlos a diversos usuarios.

RabbitMQ:

RabbitMQ es un software de negociación de mensajes de código abierto que funciona como un middleware de mensajería. Implementa el estándar Advanced Message Queuing Protocol.

Pub/Sub:

Google Pub/Sub es un servicio de la plataforma nube de Google (GCP) que permite enviar mensajes asíncronos con varios remitentes y varios destinatarios. Este servicio permite una comunicación segura con mensajes duraderos y de baja latencia.

MongoDB:

Es un sistema de base de datos NoSQL, orientado a documentos y de código abierto.

En lugar de guardar los datos en tablas, tal y como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

Redis:

Redis es un motor de base de datos en memoria, basado en el almacenamiento en tablas de hashes pero que opcionalmente puede ser usada como una base de datos durable o persistente.

NodeJS:

Es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

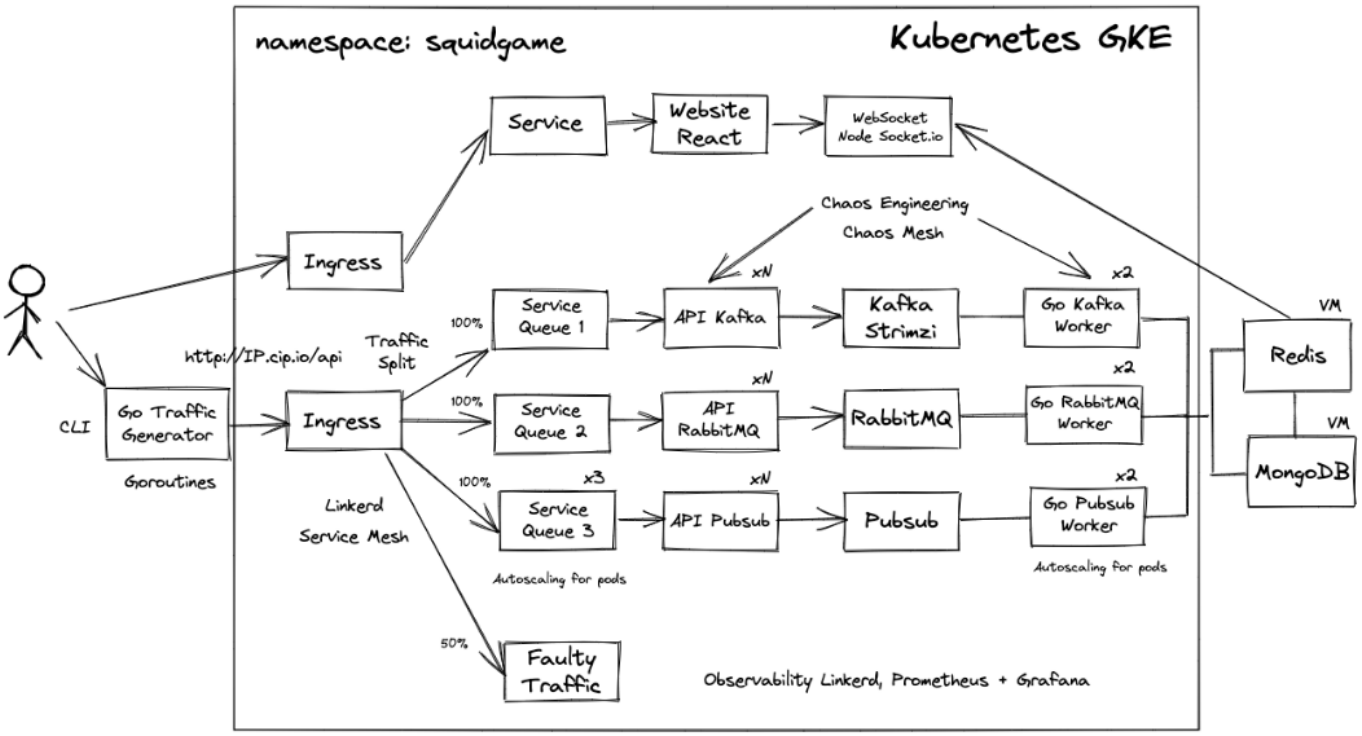
ReactJS:

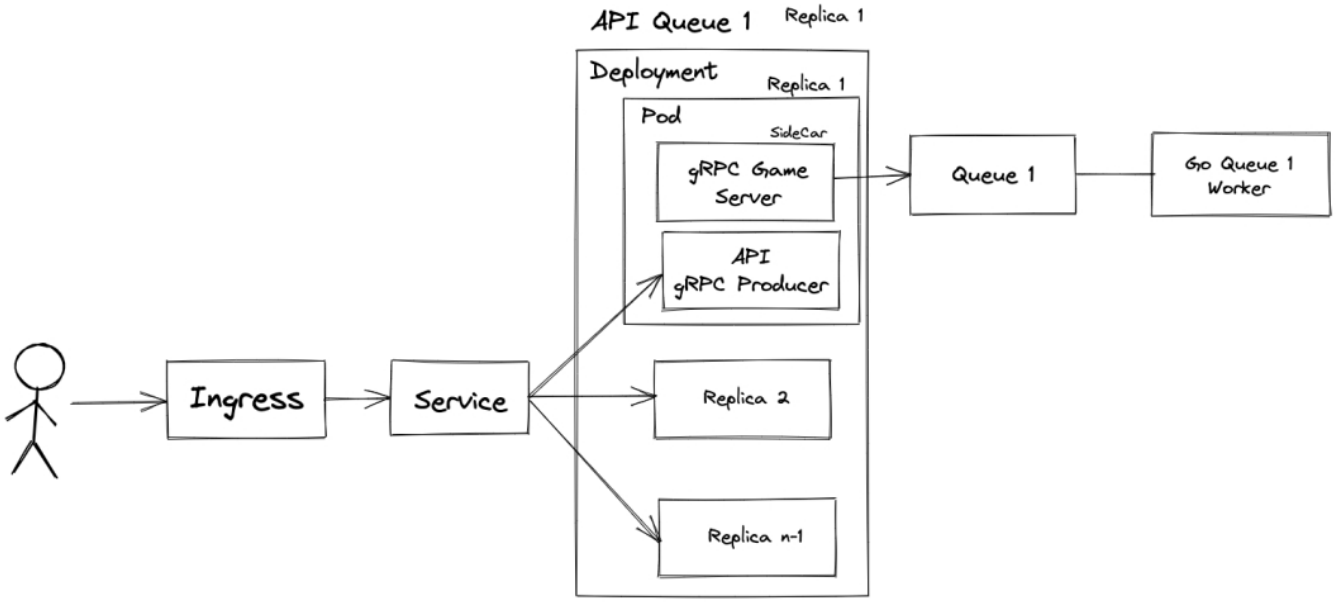
React es una biblioteca Javascript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones en una sola página.

Web sockets:

Socket.IO es una biblioteca de JavaScript para aplicaciones web en tiempo real. Permite la comunicación bidireccional en tiempo real entre clientes y servidores web.

Arquitectura del sistema utilizada:





Flujo de la aplicación:

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 son tres algoritmos sencillos implementados en el servidor de gRPC 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.

1. **MaxPlayer**:
   Gana el jugador con el número más alto ingresado
2. **MinPlayer**:
   Ganará siempre el primer jugador, el 1 es el número privilegiado dentro del juego.
3. **RandomPlayer**:
   Ganará un jugador seleccionado aleatoriamente.

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.

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

Generación del cuerpo de la petición:

La estructura del cuerpo en formato JSON que será enviado como el cuerpo de la petición, fue:

```go
type Request struct {
    Gameid   int    `json:"gameid"`
    Gamename string `json:"gamename"`
    Players  int    `json:"players"`
}
```

Una vez que se generó un el cuerpo, se realiza el POST para llamar al Ingress controller y poder enviar el tráfico a cada servicio.

Luego de que el tráfico llegue a cada API, haciendo uso de gRPC se pasa esta información del juego al Servidor de gRPC 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.

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.

DOCKER:
Se utilizó para empaquetar la aplicación en contenedores, donde se usaron técnicas distroless para crear imágenes pequeñas en lo posible.
Docker fue la herramienta para crear un entorno local de pruebas antes de desplegar las imágenes en el clúster de Kubernetes.

KUBERNETES:
Kubernetes es el encargado de la orquestación de los contenedores. Antes de que el cliente genere el tráfico, el proyecto implementa un clúster de Kubernetes que se utiliza 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 utilizó el namespace squidgame.

BALANCEADORES DE CARGA:
Se configuró un balanceador de carga de capa 7 (Kubernetes Ingress) en el clúster de Kubernetes utilizando 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 como broker, la segunda usando RabbitMQ y la tercera Google Google PubSub. Toda la información de entrada pasará a través del ingress controller.

RUTAS DE MENSAJERÍA:

1. Kafka
2. RabbitMQ
3. PubSub

Funciones de cada ruta:

➔ Generador de Trafico
➔ Ingress
➔ API RabbitMQ
➔ RabbitMQ
➔ Go RabbitMQ Worker
➔ Escribir en las bases de datos NoSQL (Redis y MongoDB)

Se implementaron el escalado automático vertical y horizontal, subprocesos de acuerdo con la naturaleza del servicio a implementar.

TRAFFIC SPLITTING:

Para implementar Traffic Splitting, el proyecto utiliza 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 se usó 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 #3 100%
- Queue #1 50%, faulty traffic 50%
- Queue #2 50%, faulty traffic 50%
- Queue #3 50%, faulty traffic 50%
- Queue #1 33.33%, Queue #2 33.33%, Queue #3 33.33%

Cada Queue corresponde al servicio de mensajería a utilizar; PubSub, Kafka o RabbitMQ.

gRPC AND BROKERS

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

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.

Estructura de cada Log guardado en la base de datos de **MongoDB**:

```
type MongoLog struct {
    Request_number int    `json:"request_number"`
    Gameid         int    `json:"gameid"`
    Gamename       string `json:"gamename"`
    Winner         string `json:"winner"`
    Players        int    `json:"players"`
    Worker         string `json:"worker"`
}
```

Datos y contadores que se actualizan en **Redis**:

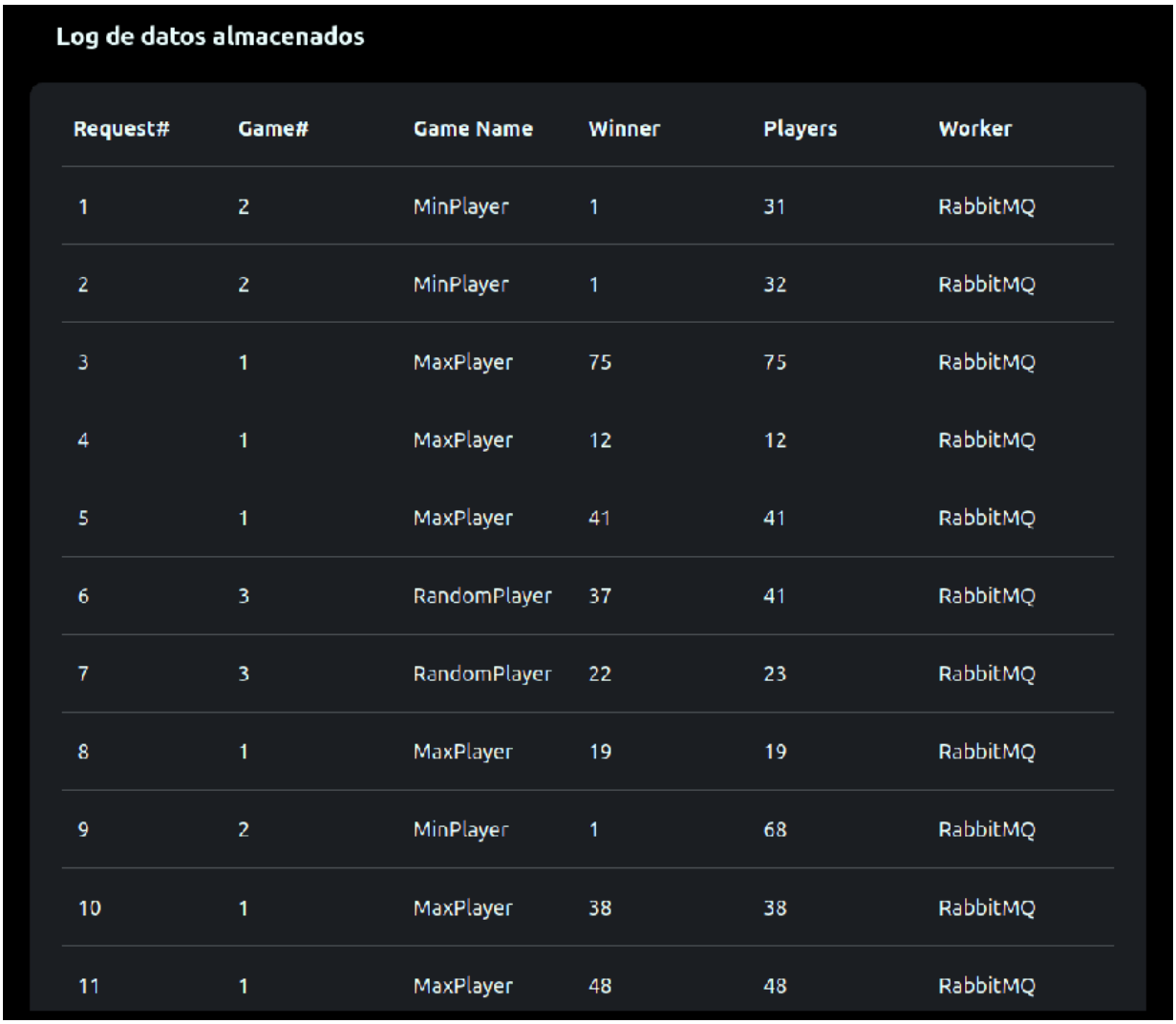
```
client.Set("last_request", l.Request_number, 0)
client.Set("last_gameid", l.Gameid, 0)
client.Set("last_gamename", l.Gamename, 0)
client.Set("last_winner", l.Winner, 0)
client.Set("last_players", l.Players, 0)
client.Set("last_worker", l.Worker, 0)
```

Estas bases de datos fueron instaladas en una máquina virtual que será accesible en la VPC del clúster de Kubernetes.

PÁGINA WEB

Se creó un sitio web que muestra en tiempo real los datos insertados, desarrollado en NodeJS y React. Se utilizaron web sockets en NodeJS para mostrar los datos en tiempo real, la página principal muestra los siguientes reportes:

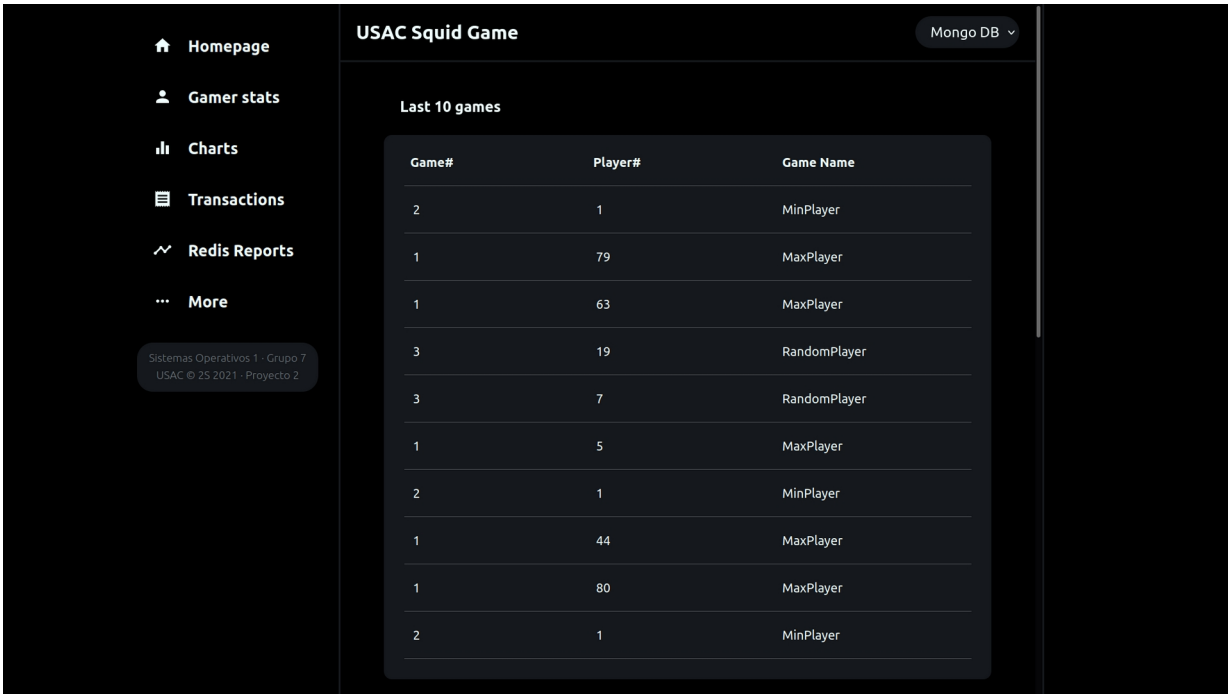
● Datos almacenados en MongoDB. (Tabla con los datos almacenados)



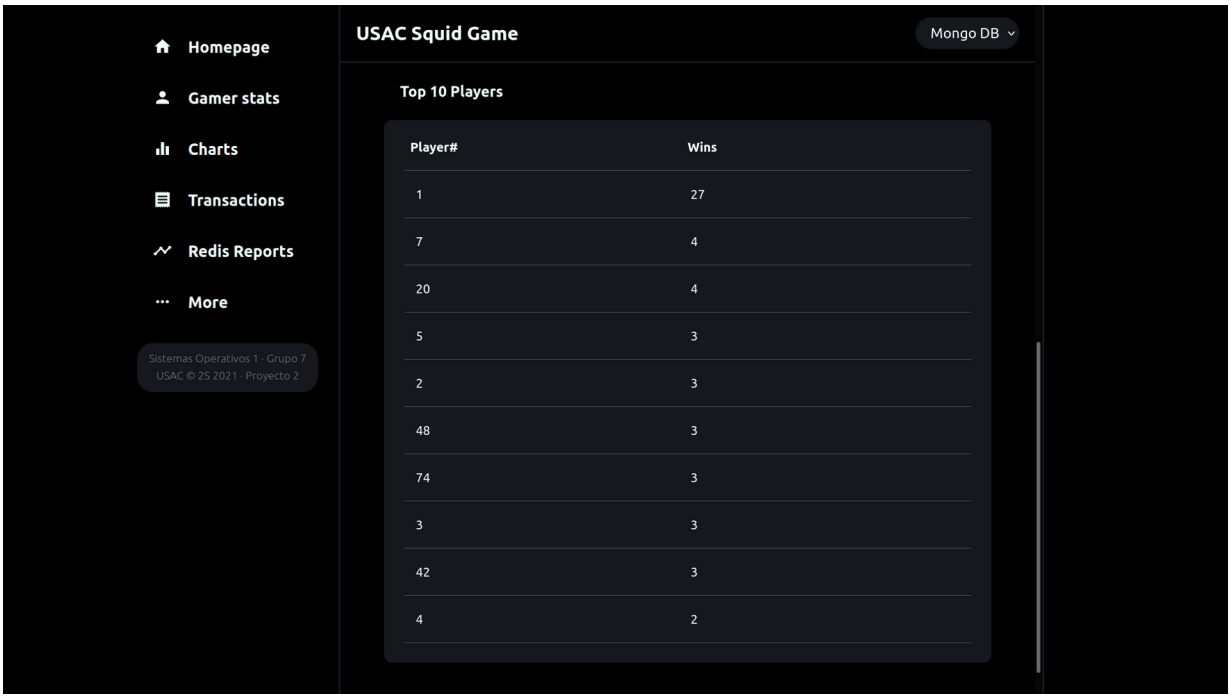
Log de datos almacenados

Request#	Game#	Game Name	Winner	Players	Worker
1	2	MinPlayer	1	31	RabbitMQ
2	2	MinPlayer	1	32	RabbitMQ
3	1	MaxPlayer	75	75	RabbitMQ
4	1	MaxPlayer	12	12	RabbitMQ
5	1	MaxPlayer	41	41	RabbitMQ
6	3	RandomPlayer	37	41	RabbitMQ
7	3	RandomPlayer	22	23	RabbitMQ
8	1	MaxPlayer	19	19	RabbitMQ
9	2	MinPlayer	1	68	RabbitMQ
10	1	MaxPlayer	38	38	RabbitMQ
11	1	MaxPlayer	48	48	RabbitMQ

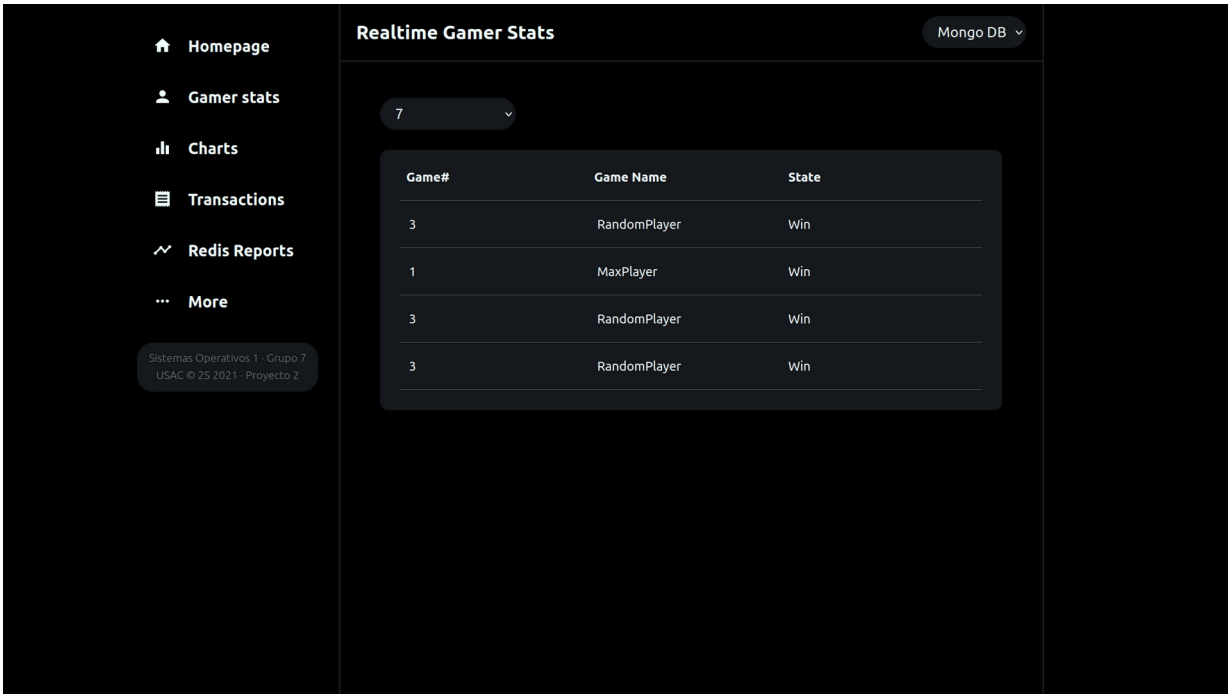
● Últimos 10 juegos.



● Los 10 mejores jugadores.

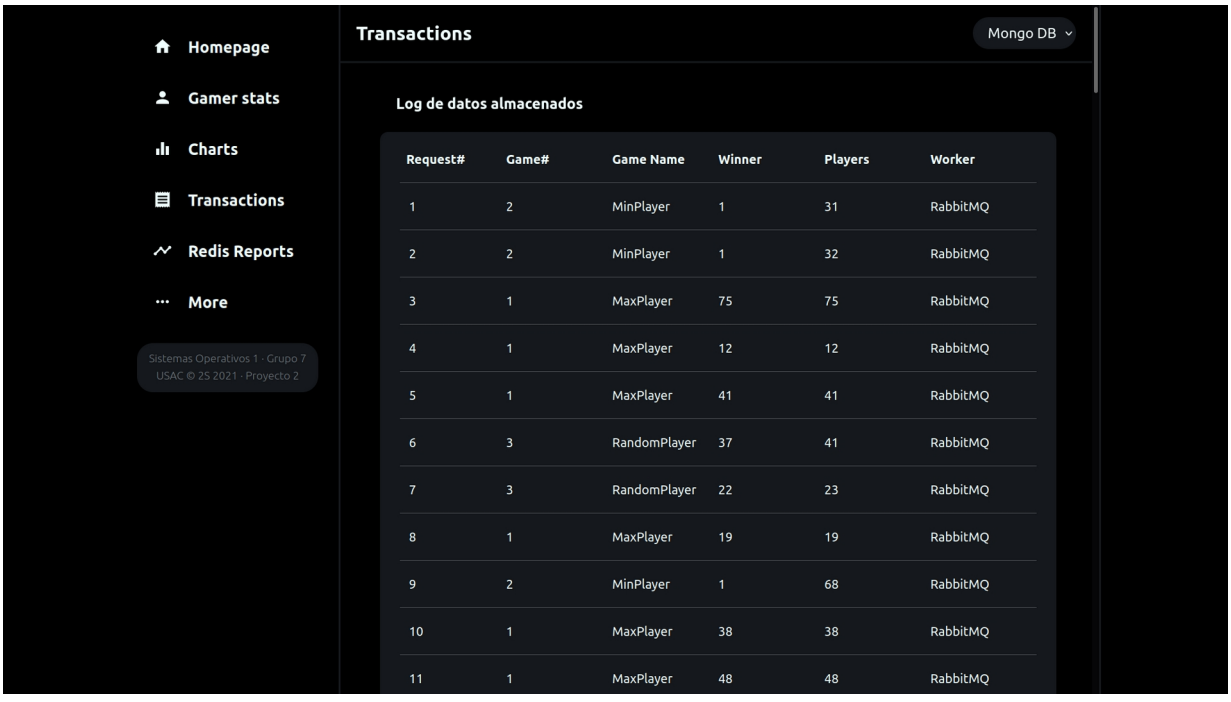


● Estadísticas del jugador en tiempo real.



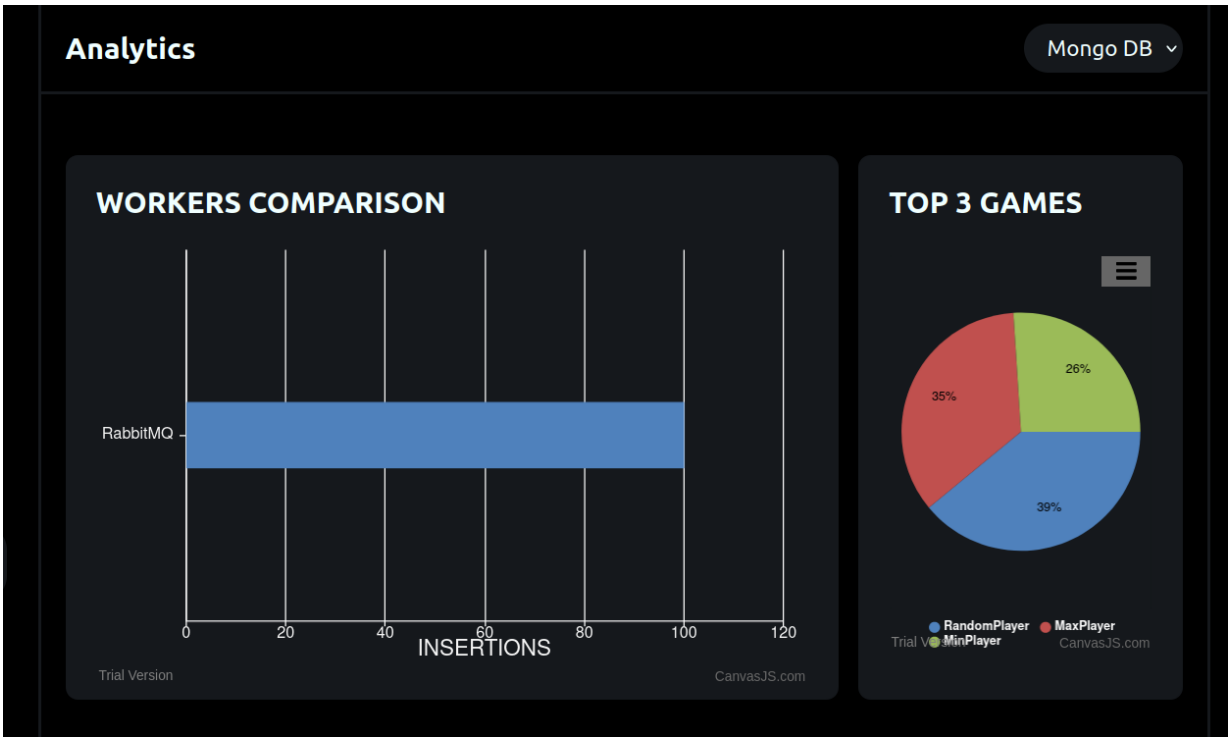
● Log de transacciones en MongoDB.

○ Tabla con los logs almacenados.



○ Gráfica del top 3 de juegos.

○ Gráfica que compara a los 3 workers de Go (la cantidad de inserciones que hizo cada worker).



OBSERVABILITY AND MONITORING

Se implementa la observabilidad y las métricas doradas usando Linkerd.

Linkerd: El proyecto tiene implementa 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.

CHAOS ENGINEERING

Se implementa 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, se realizan los siguientes experimentos:

- Tráfico defectuoso según la sección de división del tráfico.

Con Chaos mesh, los siguientes experimentos:

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

Sección de preguntas:

➔ ¿Cómo funcionan las métricas de oro?

R/ Las señales doradas son una forma eficaz de monitorear el estado general del sistema e identificar problemas. Las métricas de recursos casi siempre están disponibles de forma predeterminada desde el proveedor de infraestructura (métricas de Kubernetes) y se utilizan para monitorear el estado de la infraestructura.

➔ ¿Cómo puedes interpretar estas 7 pruebas de faulty traffic, usando como base los gráficos y métricas que muestra el tablero de Linkerd Grafana?

R/ Se pudo determinar por medio de las métricas y el dashboard de Linkerd, que a mayor proporción equitativa de las pruebas asignadas a cada servicio de cola, se comporta de una manera más controlada ya que el tráfico puede irse a cualquiera de las tres direcciones, y que con el 100% se satura de mayor forma un sólo servicio por el comportamiento de los gráficos.

➔ Menciona al menos 3 patrones de comportamiento que hayas descubierto.

R/ La latencia aumenta cuando la saturación de solicitudes es mayor provocando un comportamiento más disperso, el uso de CPU permanece controlado cuyo comportamiento es proporcional a la exigencia del mismo, las solicitudes tienen un comportamiento cerca de la línea de tendencia central a tráfico moderado.

➔ ¿Qué sistema de mensajería es más rápido?

R/ PubSub

➔ ¿Cuántos recursos utiliza cada sistema? (Basándose en los resultados que muestra el Dashboard de Linkerd)

R/ Cada sistema utiliza un leve porcentaje de recursos de CPU dependiendo la cantidad de peticiones y sobrecarga que pueda generarse en una sola ruta. La latencia y tiempo de respuesta no consumen recursos significativos y responde correctamente ante el tráfico pesado.

➔ ¿Cuáles son las ventajas y desventajas de cada sistema?

R/ La principal ventaja de PubSub es la versatilidad que ofrece al ser un servicio que está asociado en la nube y que no depende de ninguna instancia local o contenedor asociado. RabbitMQ y Kafka por su parte ofrecen un servicio de mensajería asíncrona, pero RabbitMQ es algo más tradicional y Kafka cuenta con una implementación más moderna. Para fines del proyecto, cada sistema cumplió con la funcionalidad sin problemas.

➔ ¿Cuál es el mejor sistema?

R/ PubSub y Kafka.

➔ ¿Cuál de las dos bases se desempeña mejor y por qué?

R/ Redis porque al ser un sistema de bases de datos que funciona en memoria, es mucho más rápido de consultar y el tiempo de respuesta entre cada solicitud es mucho menor. Además, se implementa un servicio de escucha continua a cada cambio y actualización en cada variable de Redis para poder ser observada en tiempo real.

➔ ¿Qué diferencia tienen los experimentos?

R/ Cada uno daña y/o corrompe el sistema de una forma diferente, ya que al matar algún servicio es necesario reiniciarlo nuevamente para que funcione al igual que con los contenedores. Cuando se genera caos se presentan daños en el funcionamiento del sistema.