




19 DE ABRIL DE 2022

MANUAL TECNICO

PRACTICA 2

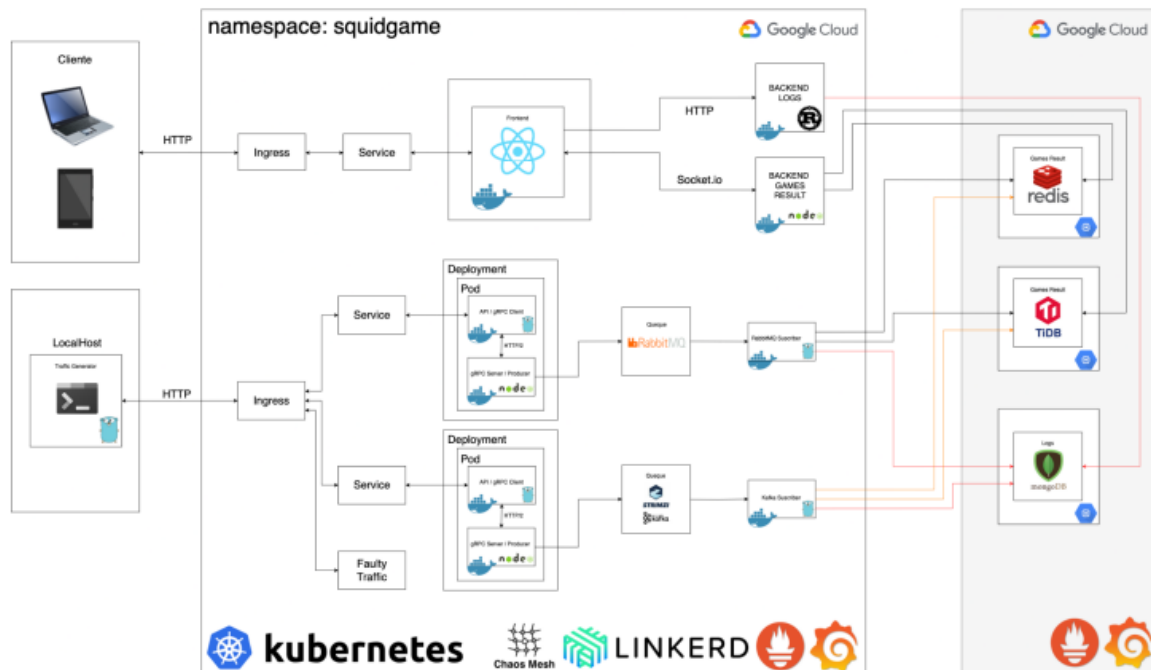
ARIEL RUBELCE MACARIO CORONADO
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA



Herramientas utilizadas.

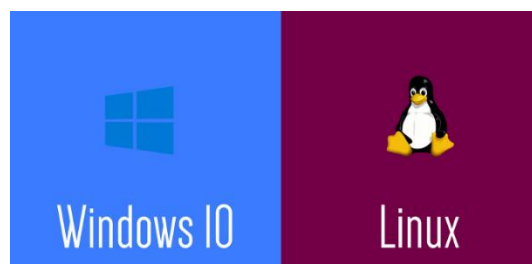
- Sistemas Operativos.
 - Ubuntu.
 - Windows.
- Lenguajes
 - Go
 - Node
- Bases de datos.
 - Mongo DB
- Brokers.
 - Kafka.
- GCP
- Docker
- GRCP

Arquitectura.



Sistemas Operativos.

Se utilizaron dos sistemas operativos para realizar el proyecto, ya que realizar la programación de los distintos módulos algunos se realizaron en Windows y otros en Linux, también se utilizó Linux en las maquinas virtuales ya que se encuentra mayor facilidad para realizar los distintos comandos.



Lenguajes.

Se usó Go y Node en la mayoría de los módulos del proyecto, para realizar el cliente y el servidor de grpc y go también se uso para el subsriber de Kafka.



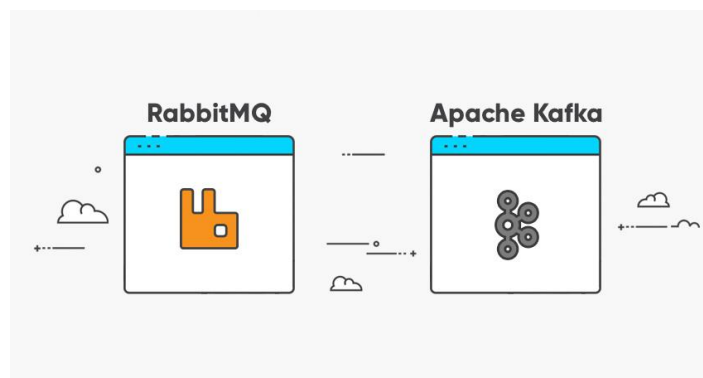
Bases de datos.

Se usó 1 bases de datos las cuales son, mongo db, la cual es una base de datos de documentos la cual nos sirve para guardar nuestros logs.



Brokers.

Kafka que 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.



GCP.

Se trata de la suite de infraestructuras y servicios que Google utiliza a nivel interno y ahora y disponible para cualquier empresa de tal forma que sea aplicable a multitud de proceso empresariales.

Básicamente nos aparta todas las herramientas necesarias para diseñar hacer testing y lanzar aplicaciones desde gcloud con mucha mas seguridad y escalabilidad que cualquiera herramienta gracias a la propia infraestructura con la que Google cuenta.



Docker.

Docker es una plataforma de software que le permite crear, probar e implementar aplicaciones rápidamente. Docker empaqueta software en unidades estandarizadas llamadas contenedores que incluyen todo lo necesario para que el software se ejecute, incluidas bibliotecas, herramientas de sistema, código y tiempo de ejecución. Con Docker, puede implementar y ajustar la escala de aplicaciones rápidamente en cualquier entorno con la certeza de saber que su código se ejecutará.



Creación del cluster.

Crear clúster

Selecciona el modo de clúster que quieres usar.



Compara los modos de clúster para obtener más información sobre sus diferencias.

[COMPARAR](#)



GKE Standard

Un clúster de Kubernetes de pago por nodo en el que configuras y administras los nodos.

[Más información](#)

CONFIGURAR



GKE Autopilot

Un clúster de Kubernetes de pago por Pod en el que GKE administra los nodos con la configuración mínima requerida.

[Más información](#)

CONFIGURAR

CANCELAR

Debemos de seleccionar una GKE Standard para crear nuestro cluster.

Aspectos básicos del clúster

El clúster nuevo se creará con el nombre, la versión y la ubicación que especifiques aquí. El nombre y la ubicación no se podrán cambiar después de que se cree el clúster.



Para experimentar con un clúster asequible, prueba **Mi primer clúster** en la [Guía de configuración de clústeres](#)

Nombre

cluster-proyecto



Tipo de ubicación

Los precios de los recursos pueden variar entre regiones determinadas. [Más información](#)

☒ Zonal

☐ Regional

Zona

us-central1-c



Se selecciona el nombre del cluster y la región en la que deseemos el cluster.

Versión del plano de control

Elige un canal de versiones para administrar de forma automática la versión y la cadencia de actualización del clúster. Elige una versión estática para administrar de forma más directa la versión del clúster. [Obtén más información.](#)

☒ Versión estática

☐ Canal de versiones

Versión estática

1.21.10-gke.2000 (predeterminado)



En versión de plano de control seleccionamos versión estática y la versión predeterminada.

Detalles del grupo de nodos

El clúster nuevo se creará con al menos un grupo de nodos. Un grupo de nodos es una plantilla para los conjuntos de nodos creados en este clúster. Después de la creación del clúster, se pueden agregar y quitar más grupos de nodos.

Nombre

default-pool

Versión de nodo

1.21.10-gke.2000 (versión del plano de control)

Tamaño

Cantidad de nodos *

3

El rango de direcciones del pod limita el tamaño máximo del clúster. [Más información](#)

☐ Habilitar el ajuste de escala automático ?

☐ Especificar las ubicaciones de los nodos ?

Seleccionamos en default pool seleccionamos la cantidad de nodos que deseemos para nuestro cluster.

Serie

E2

Selección de la plataforma de CPU según la disponibilidad

Tipo de máquina

e2-medium (2 CPU virtuales, 4 GB de memoria)



vCPU

1 núcleo compartido

Memory

4 GB

▼ PLATAFORMA DE CPU Y GPU

Tipo de disco de arranque

Disco persistente estándar

Tamaño de disco de arranque (GB)

100

☐ Habilitar encriptación administrada por el cliente para el disco de arranque ?

Seleccionamos el tipo de máquina que deseamos y luego precedemos a crear nuestro cluster, tardara alrededor de 5 minutos.

Archivos YAML.

clients-servers

Deployment de aplicación grpc con servidor en go, y cliente en node, con conexión a Kafka stimzi

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grpc-app-macario
  namespace: squidgame
  labels:
    app: grpc-app-macario
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grpc-app-macario
  template:
    metadata:
      annotations:
        linkerd.io/inject: enabled
    labels:
      app: grpc-app-macario
    spec:
      hostname: grpc-host-macario
      containers:
        - name: client-node
          image: amacario502/clientgrpc
          ports:
            - containerPort: 3000
          env:
            - name: GRCP_SERVER
              value: "grpc-host-macario:50051"

        - name: server-go
          image: amacario502/servergrpc_201905837
          ports:
            - containerPort: 50051
          env:
            - name: ADD_KAFKA
              value: my-cluster-kafka-bootstrap.squidgame:9092
```

Deployment de aplicación grpc con servidor en node, y cliente en go, con conexión a RabbitMQ

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grpc-app-oscar
  namespace: squidgame
  labels:
    app: grpc-app-oscar
spec:
  replicas: 1
  selector:
    matchLabels:
      pod: grpc-app-oscar
  template:
    metadata:
      annotations:
        linkerd.io/inject: enabled
      labels:
        pod: grpc-app-oscar
    spec:
      hostname: grpc-host-oscar
      containers:
        - name: client-go
          image: cocacore7/client_go_201908335
          ports:
            - containerPort: 3000
          env:
            - name: GRCP_SERVER
              value: "grpc-host-oscar:50051"
        - name: server-node
          image: cocacore7/server_node_201908335
          ports:
            - containerPort: 50051
          env:
            - name: RABBIT_HOST
              value: "rabbitmq-0.rabbitmq.rabbits.svc.cluster.local"
            - name: RABBIT_PORT
              value: "5672"
            - name: RABBIT_USERNAME
              value: "guest"
            - name: RABBIT_PASSWORD
              value: "guest"
```

Se crean los servicios de tipo ClusterIP para cada aplicación grpc

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  name: service-macario
  namespace: squidgame
  labels:
    app: grpc-app-macario
spec:
  ports:
    - port: 3000
      protocol: TCP
      targetPort: 3000
  selector:
    app: grpc-app-macario
  type: ClusterIP
status:
  loadBalancer: {}
```

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  name: service-oscar
  namespace: squidgame
  labels:
    app: grpc-app-oscar
spec:
  ports:
    - port: 3000
      protocol: TCP
      targetPort: 3000
  selector:
    app: grpc-app-oscar
  type: ClusterIP
status:
  loadBalancer: {}
```

Se crea el servicio ingress inyectando el ingress controller que nos ayudara a implementar traffic splitting entre los servicios clusterip

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: grcp-services-apps
  annotations:
    nginx.ingress.kubernetes.io/service-upstream: "true"
  namespace: squidgame
spec:
  ingressClassName: nginx
  rules:
  - host: "backend.104.197.37.174.nip.io"
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: service-macario
            port:
              number: 3000
```

Se configura un servicio de faulty traffic para simular perdida de trafico en nuestro cluster con traffic splitting

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: faulty-traffic
  namespace: squidgame
  labels:
    app: faulty-traffic
spec:
  selector:
    matchLabels:
      app: faulty-traffic
  replicas: 1
  template:
    metadata:
      annotations:
        linkerd.io/inject: enabled
      labels:
        app: faulty-traffic
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        volumeMounts:
        - name: nginx-config
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
      volumes:
      - name: nginx-config
        configMap:
          name: faulty-traffic
---
```

Rabbit-conf

Se configuran y deployan 4 pods de rabbit

```
kind: StatefulSet
metadata:
  name: rabbitmq
spec:
  serviceName: rabbitmq
  replicas: 4
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      serviceAccountName: rabbitmq
      initContainers:
        - name: config
          image: busybox
          command: ['/bin/sh', '-c', 'cp /tmp/config/rabbitmq.conf /config/rabbitmq.conf && ls -l /config/ && cp /tmp/config/enabled_plugins /etc/rabbitmq/enabled_plugins']
          volumeMounts:
            - name: config
              mountPath: /tmp/config/
              readOnly: false
            - name: config-file
              mountPath: /config/
            - name: plugins-file
              mountPath: /etc/rabbitmq/
      containers:
        - name: rabbitmq
          image: rabbitmq:3.8-management
          ports:
            - containerPort: 4369
              name: discovery
            - containerPort: 5672
              name: amqp
          env:
            - name: RABBIT_POD_NAME
              valueFrom:
                fieldRef:
                  apiVersion: v1
                  fieldPath: metadata.name
            - name: RABBIT_POD_NAMESPACE
              valueFrom:
```

Subscribers

Se crean dos deployments con las aplicaciones de subscribers para cada sistema de mensajería implementado

```
etcd / ! subscribers.yml
name: subscriber-deployment-rabbit
namespace: squidgame
labels:
  app: subscriber-rabbit
spec:
  replicas: 1
  selector:
    matchLabels:
      app: subscriber-rabbit
  template:
    metadata:
      labels:
        app: subscriber-rabbit
    spec:
      hostname: grpc-host-rabbit
      containers:
        - name: subscriber-rabbit-cont
          image: cocacore7/suscriber_go_rabbit_201908335
          env:
            - name: ADD_RABBIT
              value: rabbitmq-0.rabbitmq.squidgame.svc.cluster.local:5672
            - name: ADD_MONGO
              value: "34.125.105.99"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: subscriber-deployment-kafka
  namespace: squidgame
  labels:
    app: subscriber-kafka
spec:
  replicas: 1
  selector:
    matchLabels:
      app: subscriber-kafka
  template:
    metadata:
      labels:
        app: subscriber-kafka
    spec:
      hostname: grpc-host
      containers:
        - name: subscriber-kafka-cont
          image: amacario502/subscriber
          env:
            - name: ADD_KAFKA
```

Traffic-splitting

Se implementa un archivo de configuracion de carga para nuestros servicios

```
apiVersion: split.smi-spec.io/v1alpha1
kind: TrafficSplit
metadata:
  name: faulty-split
  namespace: squidgame
spec:
  service: grpc-app-macario
  backends:
  - service: grpc-app-macario
    weight: 400m
  - service: grpc-app-oscar
    weight: 400m
  - service: faulty-traffic
    weight: 200m
```

Front-conf

Se implementa el deployment de la aplicación de rust con su servicio Load Balancer

```
apiVersion: apps/v1
kind: Deployment
▼ metadata:
  namespace: squidgame
  name: rust-gerson
▼ labels:
  deployment: rust-gerson
▼ spec:
▼ selector:
▼   matchLabels:
     pod: rust-gerson-pod
  replicas: 1
▼ template:
▼   metadata:
▼     labels:
▼       pod: rust-gerson-pod
▼   spec:
▼     containers:
▼       - name: rust-gerson
         image: gersonquinia/rust_image
▼       ports:
▼         - containerPort: 8000
▼       resources:
         # Sin limites de recursos
    ---
apiVersion: v1
kind: Service
▼ metadata:
  namespace: squidgame
  name: svc-rust-gerson
▼ labels:
  service: svc-rust-gerson
▼ spec:
  type: LoadBalancer
▼ selector:
  pod: rust-gerson-pod
▼ ports:
▼   - port: 8000
     targetPort: 8000
     name: http
     protocol: TCP
```


Se implementa el deploy de la aplicación de node y su respectivo servicio de tipo load balancer

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: squidgame
  name: node-gerson
  labels:
    deployment: node-gerson
spec:
  selector:
    matchLabels:
      pod: node-pod-gerson
  replicas: 1
  template:
    metadata:
      labels:
        pod: node-pod-gerson
    spec:
      containers:
        - name: node-gerson
          image: gersonquinia/backend_node
          ports:
            - containerPort: 8080
          resources:
            # Sin limites de recursos
---
apiVersion: v1
kind: Service
metadata:
  namespace: squidgame
  name: svc-node-gerson
  labels:
    service: svc-node-gerson
spec:
  type: LoadBalancer
  selector:
    pod: node-pod-gerson
  ports:
    - port: 8080
      targetPort: 8080
      protocol: TCP
```

Se implementa el deploy de la aplicación frontend con su respectivo servicio ClusterIP

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: squidgame
  name: frontend-gerson
  labels:
    deployment: frontend-gerson
spec:
  selector:
    matchLabels:
      pod: frontend-pod-gerson
  replicas: 1
  template:
    metadata:
      labels:
        pod: frontend-pod-gerson
    spec:
      containers:
        - name: frontend-gerson
          image: gersonquinia/run_front
          ports:
            - containerPort: 4000
          env:
            - name: REACT_APP_RUST_HOST
              value: '34.123.117.181'
            - name: REACT_APP_NODEJS_HOST
              value: '34.136.221.182'
          resources:
            # Sin limites de recursos
---
apiVersion: v1
kind: Service
metadata:
  namespace: squidgame
  name: svc-frontend-gerson
  labels:
    service: svc-frontend-gerson
spec:
  type: ClusterIP
  selector:
    pod: frontend-pod-gerson
  ports:
    - port: 4000
      targetPort: 4000
      protocol: TCP
```

Se implementa el servicio ingress con inyección de ingress controller para acceder al servicio de la aplicación frontend

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
▼ metadata:
  namespace: squidgame
  name: ingress-frontend-gerson
▼ annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /
  nginx.ingress.kubernetes.io/service-upstream: "true"
▼ spec:
  ingressClassName: nginx
▼ rules:
▼   - host: "front.104.197.37.174.nip.io"
▼     http:
▼       paths:
▼         - pathType: Prefix
▼           path: /
▼           backend:
▼             service:
▼               name: svc-frontend-gerson
▼               port:
▼                 number: 4000
```

Preguntas.

RPC AND BROKERS

¿Qué sistema de mensajería es más rápido?

Rabbit

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

¿Cuáles son las ventajas y desventajas de cada sistema?

Las ventajas de Kafka esta fase fue que la instalación de este sistema de mensajería es muy rápida ya que existe strimzi además podemos observa que las cola es mas eficiente, una de las desventajas es que es algo complejo y se necesitan varios días para aprender a utilizarla.

Una ventaja de rabbit es que es un fácil de utilizar y aprender y una desventaja de rabbit es que en un cluster debemos de realizar unos pasos distintos para poder deployarlo.

¿Cuál es el mejor sistema?

El mejor es Rabbit ya que es muy fácil de aprender y fácil de instalar también también por las resultados podemos observar que es mas eficiente.

NOSQL.

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

Redis, ya que según nuestros resultados se puede observar que los datos llevan mucho mas rápido vs tidis db también ya que tidis es basada en redis.