Universidad de San Carlos de Guatemala
Facultad de ingeniería
Escuela de Ciencias y Sistemas
Sistemas Operativos 1
Ing. Sergio Arnaldo Méndez Aguilar
Aux. Leonel Aguilar
Aux. Sebastián Sánchez


# MANUAL TÉCNICO

## Proyecto 2

Rafael Alejandro Morales Donis - 200714558
Marlon Abraham Fuentes Zarate - 199911132
Paul Steve Contreras Herrera – 201408489
Glendy Marilucy Contreras González - 201025406

**Grupo 10**

*Guatemala mayo de 2021*

# CONTENIDO

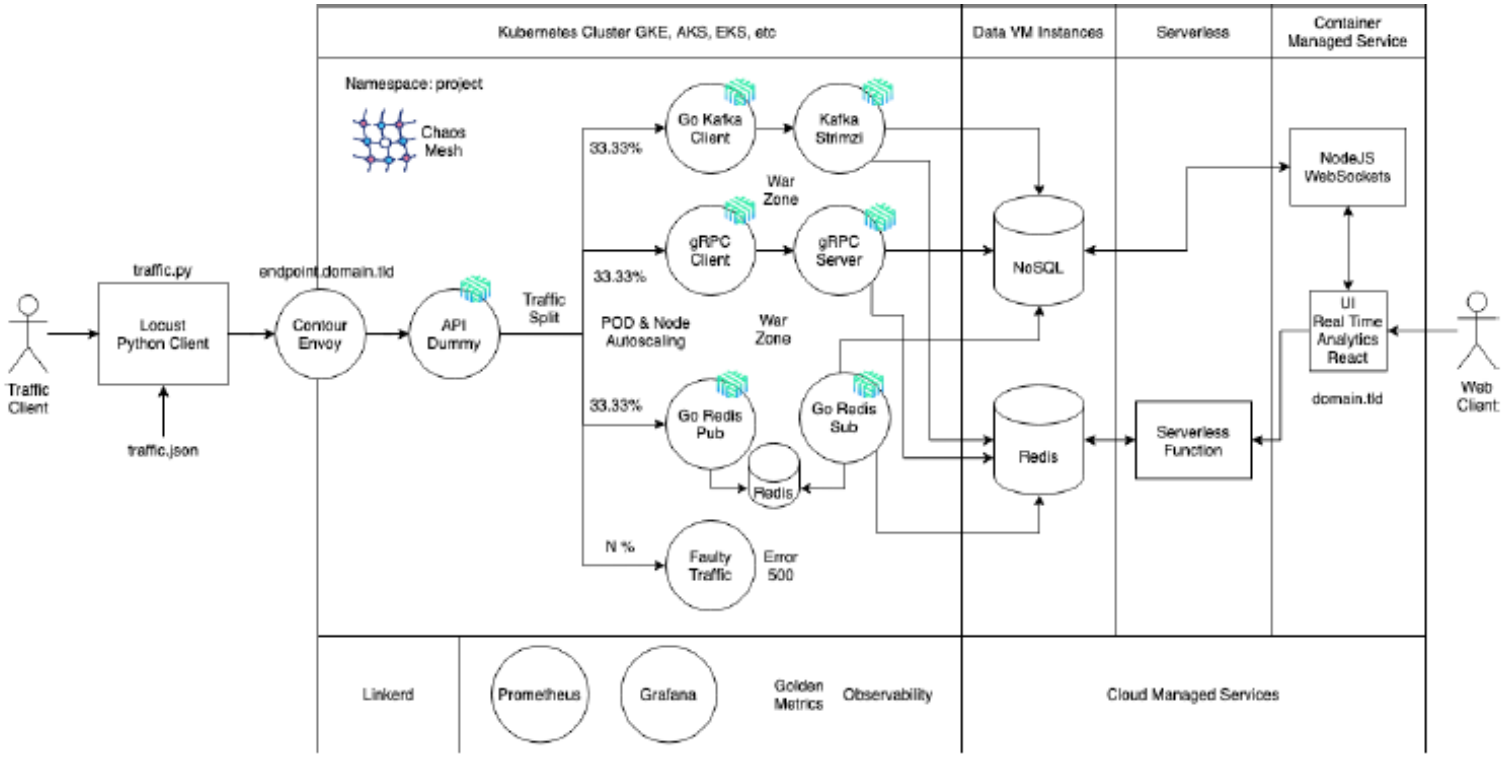
## DESCRIPCIÓN

Se utiliza una service mesh para dividir el tráfico. Adicionalmente, se genera faulty traffic con Linkerd y Chaos Mesh para la implementación de Chaos Engineering.

El objetivo del sistema es visualizar la información histórica de las personas vacunadas contra la COVID-19 alrededor del mundo.

Se presenta un sistema genérico de arquitectura distribuida que muestra estadísticas en tiempo real utilizando Kubernetes y service mesh como Linkerd y otras tecnologías Cloud Native.

## ARQUITECTURA GENERAL DEL SISTEMA



# HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO

## GOOGLE CLOUD:

Es una plataforma de Google en la nube en dónde se puede tener cualquier programa que esté en desarrollo. Se puede utilizar como hosting para una página web, pero también se puede tener APIs, apps en desarrollo, etc.

En esta plataforma se crearon las maquinas virtuales que se describen en la arquitectura.

## DOCKER (Versión 20.10.5)

Es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

## LOCUST (versión 1.4.3)

Locust es una herramienta que permite generar tráfico de prueba, se utiliza python para configurarlo, se utiliza  para enviar el tráfico contenido en el archivo JSON.

## MONGO DB

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, guarda la información en estructuras de datos BSON, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

## KUBERNETES

sistema de código libre para la automatización del despliegue, ajuste de escala y manejo de aplicaciones en contenedores.

## GO (versión 2.38)

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++.

Es el lenguaje utilizado para programar los intermediarios.

## REACT WEB APP

Es una librería Javascript focalizada en el desarrollo de interfaces de usuario interactivas de forma sencilla. Diseña vistas simples para cada estado de una aplicación, y se actualiza y renderiza de manera eficiente los componentes correctos cuando los datos cambien.

## STREAMZI

Conjunto de operadores para ejecutar un cluster Kafka en Kubernetes permitiendo de forma sencilla diferentes configuraciones de despliegue.

## REDIS

Redis es un almacén de estructura de datos de valores de clave en memoria rápido y de código abierto. Redis incorpora un conjunto de estructuras de datos en memoria versátiles que le permiten crear con facilidad diversas aplicaciones personalizadas.

## LINKERD

Linkerd es un proxy de red open source diseñado para ser desplegado como Service Mesh y que está basado en finagle y netty. Su principal cometido es hacer de link, como su nombre indica, entre las diferentes piezas de sistemas distribuidos.

# DATOS DE SISTEMA UTILIZADOS EN LAS MAQUINAS VIRTUALES

## REQUISITOS MÍNIMOS

### SISTEMA OPERATIVO

- UBUNTU 16.04 LTS
- UBUNTU 18.04 LTS

### RAM

4GB

### ESPACIO DE ALMACENAMIENTO

10GB

## ESPECIFICACIÓN DE CÓDIGO Y COMANDOS UTILIZADOS EN CADA MODULO

| Elemento | Configuraciones |
|---|---|
| **DOCKER** | • Primero se actualiza la lista de paquetes existente:<br>**sudo apt update**<br>• Se instalan paquetes de requisitos previos que le permiten a apt usar paquetes mediante HTTPS:<br>**sudo apt install apt-transport-https ca-certificates curl software-properties-common**<br>• Se agrega la clave GPG para el repositorio oficial de Docker:<br>**curl -fsSL https://download.docker.com/linux/ubuntu/gpg \| sudo apt-key add –**<br>• Se agrega el repositorio de Docker a las fuentes de APT:<br>**sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"**<br>• Se actualiza la base de datos de paquetes usando los paquetes de Docker del repositorio que se acaba de agregar.<br>**sudo apt update**<br>• Se corrobora que se va a instalar desde el repositorio de Docker en vez del repositorio de Ubuntu predeterminado:<br>**sudo apt-cache policy docker-ce**<br>• Verifique que se esté ejecutando con el comando:<br>**sudo systemctl status Docker** |
| **Docker Compose** | • Comando para descargar la versión estable actual de Docker Compose:<br>**sudo curl -L "https://github.com/docker/compose/releases /download/1.26.2/docker -compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose**<br>• Aplicar permisos ejecutables al binario:<br>**sudo chmod +x /usr/local/bin/docker-compose**<br>• Crear un vínculo simbólico a o cualquier otro directorio de la ruta de acceso:<br>**sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose**<br>• Pruebe la instalación:<br>**docker-compose --version** |
| | |

**LOCUST**

```
# Esta variable controlara si queremos que salgan todas las salidas, o
unicamente las mas importantes
debug = True

# Esta funcion utilizaremos para las salidas que no queremos que
salgan siempre
# excepto cuando estamos debuggeando
def printDebug(msg):
    if debug:
        print(msg)

class Reader():

    def __init__(self):
        self.array = []

    def pickRandom(self):
        length = len(self.array)

    if (length > 0):
         random_index = randrange(0, length - 1) if length > 1 else 0

            return self.array.pop(random_index)

        else:
            print (">> Reader: No hay más valores para leer en el
archivo.")
            return None

    def load(self):
        print (">> Reader: Iniciando con la carga de datos")

        try:
            with open("traffic.json", 'r') as data_file:
                self.array = json.loads(data_file.read())

            print (f'>> Reader: Datos cargados correctamente,
{len(self.array)} datos -> {getsizeof(self.array)} bytes.')

        except Exception as e:
            print (f'>> Reader: No se cargaron los datos {e}')

# Deriva de HTTP-User, simulando un usuario utilizando nuestra APP.
```

```
# En esta clase definimos todo lo que necesitamos hacer con locust.
class MessageTraffic(HttpUser):


# Tiempo de espera entre peticiones
    #  entre cada llamada HTTP
    wait_time = between(0.1, 0.9)

    # Este metodo se ejecutara cada vez que empecemos una prueba
    # Este metodo se ejecutara POR USUARIO
    def on_start(self):
        print (">> MessageTraffic: Iniciando el envio de tráfico")
        self.reader = Reader()
        self.reader.load()

 # Este es una de las tareas que se ejecutara cada vez que pase el
tiempo
    @task
    def PostMessage(self):
        random_data = self.reader.pickRandom()

        if (random_data is not None):

            data_to_send = json.dumps(random_data)
            printDebug (data_to_send)

            myheaders = {'Content-Type': 'application/json', 'Accept':
'application/json'}
            self.client.post("/", data= json.dumps(random_data),
headers = myheaders)
            #self.client.post("/", json=random_data)

        else:
            print(">> MessageTraffic: Envio de tráfico finalizado, no
hay más datos que enviar.")
            self.stop(True)
```

| | |
|---|---|
| **Base de Datos en Mongo** | **Docker-Compose** |

```
version:
"3.9"
```

```
services:
  # node:
  #   restart: on-failure
  #   build: ./nodejsapp
  #   depends_on:
  #     - mongo
  #   ports:
  #     - 3000:3000
  #   networks:
  #     - grpctuiter
  #   volumes:
  #     - ./:/code
  mongo:
    image: mongo
    ports:
      - 27017:27017
    networks:
      - grpctuiter
    volumes:
      - mongodb:/data/db
    command: [--auth]
volumes:
 mongodb:


networks:
  grpctuiter:
    driver: "bridge"
```

**VM GRPC**

### Docker Compose:

```yaml
version: "3.9"
services:
  grpcserver:
    build: ./server
    ports:
      - "50052:50051"
    networks:
      - grpctuiter

  grpcclient:
    build: ./cliente
    environment:
      - CLIENT_HOST=:5000
      - SERVER_HOST=grpcserver:50051
      - NAME=instanciagrpc
    ports:
      - "80:5000"
    networks:
      - grpctuiter

networks:
  grpctuiter:
    driver: "bridge"
```

### CLIENTE:

### Main del Cliente:

```go
func main() {
	instance_name := os.Getenv("NAME")
	client_host := os.Getenv("CLIENT_HOST")

	fmt.Println(">> -------- CLIENTE ", instance_name, " --------")

	fmt.Println(">> CLIENT: Iniciando servidor http en ", client_host)

	// Asignar la funcion que controlara las llamadas http
	http.HandleFunc("/", http_server)

	// Levantar el server, si existe un error levantandolo hay que apagarlo
	if err := http.ListenAndServe(client_host, nil); err != nil {
		log.Fatal(err)
	}
}
```

## Crear conexión con el servidor y enviar mensajes:

```
func sendMessage(name string, location string, age string, infectedtype string, state string) {
	server_host := os.Getenv("SERVER_HOST")

	fmt.Println(">> CLIENT: Iniciando cliente")
	fmt.Println(">> CLIENT: Iniciando conexion con el servidor gRPC ", server_host)

	// Crear una conexion con el servidor (que esta corriendo en el puerto 50051)
	// grpc.WithInsecure nos permite realizar una conexion sin tener que utilizar SSL
	cc, err := grpc.Dial(server_host, grpc.WithInsecure())
	if err != nil {
		log.Fatalf(">> CLIENT: Error inicializando la conexion con el server %v", err)
	}

	// Defer realiza una axion al final de la ejecucion (en este caso, desconectar la conexion)
	defer cc.Close()

	// Iniciar un servicio NewGreetServiceClient obtenido del codigo que genero el protofile
	// Esto crea un cliente con el cual podemos escuchar
	// Le enviamos como parametro el Dial de gRPC
	c := greetpb.NewGreetServiceClient(cc)

	fmt.Println(">> CLIENT: Iniciando llamada a Unary RPC")

	// Crear una llamada de GreetRequest
	// Este codigo lo obtenemos desde el archivo que generamos con protofile
	req := &greetpb.GreetRequest{
		// Enviar un Greeting
		// Esta estructura la obtenemos desde el archivo que generamos con protofile
		Greeting: &greetpb.Greeting{
			Name:         name,
			Location:     location,
			Age:          age,
			Infectedtype: infectedtype,
			State:        state,
		},
	}

	fmt.Println(">> CLIENT: Enviando datos al server")
	// Iniciar un greet, en background con la peticion que estamos realizando
	res, err := c.Greet(context.Background(), req)
	if err != nil {
		log.Fatalf(">> CLIENT: Error realizando la peticion %v", err)
	}
```

## SERVER:

## Main del server:

```
func main() {

	// Leer el host de las variables del ambiente
	host := os.Getenv("HOST")
	fmt.Println(">> SERVER: Iniciando en ", host)

	// Primero abrir un puerto para poder escuchar
	// Lo abrimos en este puerto arbitrario
	lis, err := net.Listen("tcp", host)
	if err != nil {
		log.Fatalf(">> SERVER: Error inicializando el servidor: %v", err)
	}

	fmt.Println(">> SERVER: Empezando server gRPC")

	// Ahora si podemos iniciar un server de gRPC
	s := grpc.NewServer()

	// Registrar el servicio utilizando el codigo que nos genero el protofile
	greetpb.RegisterGreetServiceServer(s, &server{})

	fmt.Println(">> SERVER: Escuchando servicio...")
	// Iniciar a servir el servidor, si hay un error salirse
	if err := s.Serve(lis); err != nil {
		log.Fatalf(">> SERVER: Error inicializando el listener: %v", err)
	}
```

**Función Greet:** Será llamada desde el cliente, se le pasa un contexto donde se ejecutará la función.

Retorna una respuesta como la definimos en nuestro protofile o un error.

```
func (*server) Greet(ctx context.Context, req *greetpb.GreetRequest) (*greetpb.GreetResponse, error) {
	fmt.Printf(">> SERVER: Función Greet llamada con éxito. Datos: %v\n", req)

	// Todos los datos podemos obtenerlos desde req
	// Tendra la misma estructura que definimos en el protofile
	// Para ello utilizamos en este caso el GetGreeting
	Name := req.GetGreeting().GetName()
	Location := req.GetGreeting().GetLocation()
	Age := req.GetGreeting().GetAge()
	Infectedtype := req.GetGreeting().GetInfectedtype()
	State := req.GetGreeting().GetState()

	result := Name + " - " + Location + " - " + Age + " - " + Infectedtype + " - " + State

	fmt.Printf(">> SERVER: %s\n", result)
	// Creamos un nuevo objeto GreetResponse definido en el protofile

	jsonData := map[string]string{"name": Name, "location": Location, "age": Age, "infectedtype": Infectedtype, "state": State, "origen": "grpc"}
	jsonValue, _ := json.Marshal(jsonData)
	//client := &http.Client{}
	request, err := http.Post("http://34.121.234.71:3000/subscribers", "application/json", bytes.NewBuffer(jsonValue))
	if err != nil {
		fmt.Print(err.Error())
	}
	//result.Header.Add("Accept", "application/json")
	//result.Header.Add("Content-Type", "application/json")

	//resp, err := client.Do(request)
	//request, err = http.Post("https://localhost/subscribers", "application/json", bytes.NewBuffer(jsonValue))
	if err != nil {
		fmt.Printf("The HTTP request failed with error %s\n", err)
	} else {
		data, _ := ioutil.ReadAll(request.Body)
		fmt.Println(string(data))
	}
	res := &greetpb.GreetResponse{
		Result: result,
	}

	return res, nil
}
```

**Redis**

**Docker Compose:**

```
redis:
          image: redis
          container_name: cache
          ports:
            - 6379:6379
          expose:
            - 6379
        # app:
        #   build: ./
        #   volumes:
        #     - ./:/var/www/app
        #   links:
        #     - redis
        #   ports:
        #     - 3500:3500
        #   environment:
        #     - REDIS_URL=redis://cache
        #     - NODE_ENV=development
        #     - PORT=3500
        #   command:
        #     sh -c 'npm I && node server.js'
```

**Redis Client:**

```
const redis =
require('redis');
                  const {promisify} = require('util');
                  const client =
                  redis.createClient(process.env.REDIS_URL);

                  module.exports = {
                    ...client,
                    getAsync: promisify(client.get).bind(client),
                    setAsync: promisify(client.set).bind(client),
                    keysAsync: promisify(client.keys).bind(client)
                  };
```

**Redis Server:**

```
const express =
require('express');
const app = express();

const redisClient = require('./redis-client');
//client = redisClient.createClient();

app.get('/store/:key', async (req, res) => {
  const { key } = req.params;
  const value = req.query;
  console.log(key);
  console.log(value);
  // var multi = redisClient.multi();

  // multi.rpush(key, JSON.stringify(value));
  // await multi.exec(function(errors, results)
{

  // })
    await redisClient.setAsync(key,
JSON.stringify(value));
    //client.set(key, JSON.stringify(value));
  return res.send('Success');
});
app.get('/:key', async (req, res) => {
  const { key } = req.params;
  const rawData = await
redisClient.getAsync(key);
  return res.json(rawData);
});

app.get('/', (req, res) => {
  return res.send('Hello world');
});

const PORT = process.env.PORT || 3500;
app.listen(PORT, () => {
  console.log(`Server listening on port
${PORT}`);
```

NODE JS APP

## Server:

```
const mongoose = require('mongoose')

const subscriberSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  location: {
    type: String,
    required: true
  },
  age: {
    type: Number,
    required: true
  },
  infectedtype: {
    type: String,
    required: true
  },
  state: {
    type: String,
    required: true
  },
  origen: {
    type: String,
    required: true
  }
})

module.exports = mongoose.model('Subscriber', subscriberSchema)
```

## Subscriber:

```
const express = require('express')
const router = express.Router()
const Subscriber = require('../models/subscriber')

// Getting all
router.get('/', async (req, res) => {
  try {
    const subscribers = await Subscriber.find()
    res.json(subscribers)
  } catch (err) {
```

```
    res.status(500).json({ message: err.message })
  }
})

router.get('/origen/', async (req, res) => {
  try {
    var query = { origen: req.body.origen };
    const subscribers = await Subscriber.find(query)
    res.json(subscribers)

  } catch (err) {
    res.status(500).json({ message: err.message })
  }
})



router.get('/totalorigen/', async (req, res) => {
  try {
    //var query = { origen: req.body.origen };
    //const subscribers = await Subscriber.find(query)
    //res.json(subscribers)
    const subscribers = await Subscriber.find()

    subscribers.col.aggregate(
      [
          { "$group": {
              "origen": "$ssubscribers",
              "count": { "$sum": 1 }
          }}
      ],
      function(err,docs) {
          if (err) console.log(err);
          console.log( docs );
      }
  );
```

```
    res.json(subscribers)
  } catch (err) {
    res.status(500).json({ message: err.message })
  }
})



// Getting One
router.get('/:id', getSubscriber, (req, res) => {
  res.json(res.subscriber)
})

// Creating one
router.post('/', async (req, res) => {
  const subscriber = new Subscriber({
    name: req.body.name,
    location: req.body.location,
    age: req.body.age,
    infectedtype: req.body.infectedtype,
    state: req.body.state,
    origen: req.body.origen

  })
  try {
    const newSubscriber = await subscriber.save()
    res.status(201).json(newSubscriber)
  } catch (err) {
    res.status(400).json({ message: err.message })
  }
})

// Updating One
router.patch('/:id', getSubscriber, async (req, res) => {
  if (req.body.name != null) {
    res.subscriber.name = req.body.name
```

```
  }
  if (req.body.location != null) {
    res.subscriber.location = req.body.location
  }
  try {
    const updatedSubscriber = await res.subscriber.save()
    res.json(updatedSubscriber)
  } catch (err) {
    res.status(400).json({ message: err.message })
  }
})
// Deleting One
router.delete('/:id', getSubscriber, async (req, res) => {
  try {
    await res.subscriber.remove()
    res.json({ message: 'Deleted Subscriber' })
  } catch (err) {
    res.status(500).json({ message: err.message })
  }
})
async function getSubscriber(req, res, next) {
  let subscriber
  try {
    subscriber = await Subscriber(req.params.id)
    if (subscriber == null) {
      return res.status(404).json({ message: 'Cannot find subscriber' })
    }
  } catch (err) {
    return res.status(500).json({ message: err.message })
  }
  res.subscriber = subscriber
  next()
}
module.exports = router
```

## CONCLUSIONES

1. **Cómo funcionan las golden metrics, cómo pueden interpretarse las cuatro pruebas de faulty traffic utilizando de base las gráficas y métricas que muestra Linkerd y Grafana en los dashboards.**

   **1era Prueba:** Según nuestras pruebas podemos concluir que lo más rápido es Redis PubSub.

   **3era. Prueba:** En nuestro caso, el tiempo de respuesta se mantenía muy similar probablemente por la cantidad de datos que se estaban enviando por lo que Faulty Traffic no fue muy significativo.

   **4ta. Prueba:** Para esta prueba el Faulty Traffic tampoco fue muy significativo, el tiempo de respuesta siempre se mantuvo muy similar.

2. **Mencionar al menos tres patrones de conducta que fueron descubiertos.**
   - Dependiendo de la configuración de tráfico que se le da a error-injector, la gráfica siempre va en disminución.
   - Dado que siempre se recibe el mismo porcentaje en el Request Volum los resultados presentados en las gráficas siempre son muy similares.
   - La gráfica de Latencia en GRPC se mantuvo siempre estable en comparación en Redis y PubSub.

3. **¿Qué sistema de mensajería es más rápido?**
   - El más rápido es Redis PubSub, esto debido a que en Redis no se necesita una conexión cliente-servidor, solo tener los datos en la base de datos para que el suscriptor pueda conectarse por lo que la recepción de los datos es más rápida.

5. **¿Cuáles son las ventajas y desventajas de cada sistema?**

**Redis Pub Sub:**

| Ventajas | Desventajas |
|---|---|
| • Facilidad para enviar mensajes multidifusión.<br>• Velocidad para publicar los mensajes<br>• Es livano. | • Solo se puede enviar mensajes a todos los clientes que están conectados a un mismo canal, pero no individuales.<br>• Para poder generar un canal depende de una base de datos. |

**GRPC:**

| Ventajas | Desventajas |
|---|---|
| • Permite su ejecución desde cualquier ambiente.<br>• Asigna un canal fijo entre cliente y servidor. | • Es un servicio lento en comparación con otros servicios de mensajería.<br>• No acepta multidifusión. |

### 6. ¿Cuál es el mejor sistema?

- Se pudo observar que Redis PubSub es el sistema que funciona con mayor rapidez sin embargo en cuanto a la estabilidad de los datos consideramos que GRPC presenta una mejor estabilidad, por lo que concluimos que el mejor sistema en nuestro caso es **GRPC.**

### 7. ¿Cuál de las dos bases de se desempeña mejor y por qué?

- Basándonos en la persistencia de la información podemos concluir que **MongoDB** fue la mejor base de datos ya que en Redis se perdían los datos almacenados.

### 8. ¿Cómo cada experimento se refleja en el gráfico de Linkerd, qué sucede?

- **Pod Failure:** En el transcurso del experimento se perdió por completo el funcionamiento de los pods y no era posible reiniciarlos.
- **Pod Kill:** Se puede observar en las gráficas de success rate, p95 latency y request volume que en todos los pods se pierde temporalmente la actividad.
- **Container Kill:** Se pudo observar principalmente en la gráfica de Request Rate, que los servicios reciben solicitudes, pero no pueden ser procesadas ya que el container no funciona.
- **Network Emulation Chaos:** Al momento de recibir los datos había un retraso en el tiempo.

### 9. Cómo cada experimento es distinto.

- Se diferencian en que cada configuración de Pods, Container o Red, se ven afectos directamente por cada experimento.

### 10. Cuál de todos los experimentos es el más dañino.

- PodFailure, en el transcurso del experimento se perdió por completo el funcionamiento de los pods y no era posible reiniciarlos.