

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

*Guatemala mayo de 2021*

Rafael Alejandro Morales Donis - 200714558

Marlon Abraham Fuentes Zarate - 199911132

Paul Steve Contreras Herrera – 201408489

**Grupo 10**

**MANUAL TÉCNICO**

**Proyecto 2**

Contenido

[DESCRIPCIÓN 2](#_Toc67582379)

[ARQUITECTURA GENERAL DEL SISTEMA 2](#_Toc67582380)

[HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO 3](#_Toc67582381)

[GOOGLE CLOUD: 3](#_Toc67582382)

[DOCKER (Versión 20.10.5) 3](#_Toc67582383)

[LOCUST (versión 1.4.3) 3](#_Toc67582384)

[MONGO DB 3](#_Toc67582385)

[HTTP REST API 3](#_Toc67582386)

[GO (versión 2.38) 4](#_Toc67582387)

[REACT WEB APP 4](#_Toc67582388)

[DATOS DE sistema utilizados en las maquinas virtuales 5](#_Toc67582389)

[REQUISITOS MÍNIMOS 5](#_Toc67582390)

[SISTEMA OPERATIVO 5](#_Toc67582391)

[RAM 5](#_Toc67582392)

[ESPACIO DE ALMACENAMIENTO 5](#_Toc67582393)

[ESPECIFICACIÓN DE código y comandos utilizados en cada MODULO 6](#_Toc67582394)

# 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:   **sudo apt update**   * Se instalan paquetes de requisitos previos que le permiten a apt usar paquetes mediante HTTPS:   **sudo apt install apt-transport-https ca-certificates curl**  **software-properties-common**   * Se agrega la clave GPG para el repositorio oficial de Docker:   **curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo**  **apt-key add –**   * Se agrega el repositorio de Docker a las fuentes de APT:   **sudo add-apt-repository "deb [arch=amd64]**  **https://download.docker.com/linux/ubuntu bionic stable"**   * Se actualiza la base de datos de paquetes usando los paquetes de Docker del repositorio que se acaba de agregar.   **sudo apt update**   * Se corrobora que se va a instalar desde el repositorio de Docker en vez del repositorio de Ubuntu predeterminado:   **sudo apt-cache policy docker-ce**   * Verifique que se esté ejecutando con el comando:   **sudo systemctl status Docker** |
| Docker Compose | * Comando para descargar la versión estable actual de Docker Compose:   **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**   * Aplicar permisos ejecutables al binario:   **sudo chmod +x /usr/local/bin/docker-compose**   * Crear un vínculo simbólico a o cualquier otro directorio de la ruta de acceso:   **sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose**   * Pruebe la instalación:   **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:**    **CLIENTE:**  **Main del Cliente:**    **Crear conexión con el servidor y enviar mensajes:**    **SERVER:**  **Main del server:**    **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. |
| 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}`); | |