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 1

Rafael Alejandro Morales Donis - 200714558
Marlon Abraham Fuentes Zarate - 199911132
Paul Steve Contreras Herrera - 201408489

Grupo 10

Guatemala marzo de 2021

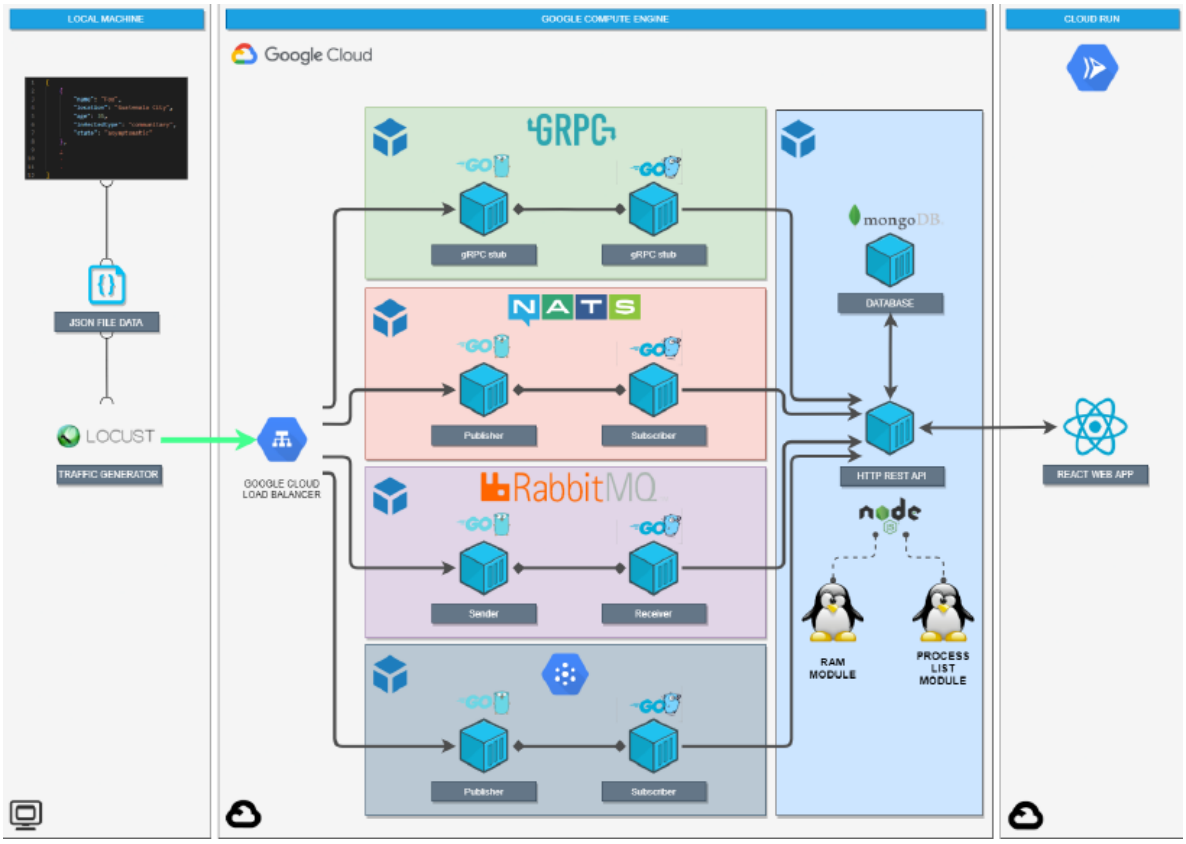
CONTENIDO

DESCRIPCIÓN

El sistema realiza un análisis en tiempo real de los datos de infecciones de COVID-19 en Guatemala. El sistema almacena los datos de infecciones y muestra las gráficas más relevantes; Esto con el objetivo de mejorar el proceso de toma de decisiones y en la búsqueda de métodos para sobrellevar la contingencia de la mejor manera.
El sistema cuenta con una carga masiva de datos además cuenta con una app web que muestra las gráficas y métricas más relevantes de los datos que se suministran al sistema, por último, muestra el estado de la RAM y un listado de procesos del servidor donde se almacenarán los datos.

Se utilizan cuatro alternativas de middlewares de mensajería; cada uno de ellos es utilizado para enviar el tráfico generado en conjunto, esto con el fin de tener una respuesta más rápida al momento de cargar datos.

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.

HTTP REST API

Es una interfaz de programación de aplicaciones que se ajusta a los límites de la arquitectura REST. Es un conjunto de definiciones y protocolos que se usa para diseñar e integrar el software de aplicaciones.

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.

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)
```

VM GRPC

Docker-Compose, Se cambia al puerto 80:

```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
}
```

VM NATS

Docker-Compose:

```
version: "3.3"
services:
  servern:
    image: nats:2.2.0
    ports:
      - "4222:4222"
      - "8222:8222"
      - "6222:6222"
    restart: always
    tty: true
    networks:
      - Nats

  emisor:
    build: ./Sender
    ports:
      - "80:80"
    depends_on:
      - servern
    networks:
      - Nats

  receptor:
    build: ./Receiver
    restart: on-failure
    depends_on:
      - servern
    networks:
      - Nats

networks:
    Nats:
      driver: bridge
```

MAIN Reciber:

```
func main() {
	received = 0

	go subscriptor()
	go check()

	for {

	}
}
```

Función Subscriptor: Código para suscribirnos y escuchar mensajes.

```
func subscriptor() {

	nc, err := nats.Connect("nats://servern:4222")
	if err != nil {
		log.Fatal(err)
	}
	defer nc.Close()

	//Nos suscribimos para escuchar mensajes
	nc.Subscribe("colaid", func(msg *nats.Msg) {
		log.Printf("%s: %s", msg.Subject, msg.Data)
		received++

		log.Printf("Mensaje Recibido: %s", msg.Data)

		postBody := []byte(string(msg.Data))

		req, err := http.Post("http://34.121.234.71:3000/subscribers", "application/json", bytes.NewBuffer(postBody))
		req.Header.Set("Content-Type", "application/json")
		failOnError(err, "Recibido")
		defer req.Body.Close()

		//Read the response body
```

SENDER:

Main del Sender y handleRequest:

```
func handleRequests() {
	http.HandleFunc("/", elemento)
	log.Fatal(http.ListenAndServe(":80", nil))
}

func main() {
	handleRequests()
}
```

Sender recibe peticiones http y se agrega el origen en el body:

```
func elemento(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        var body map[string]interface{}
        err := json.NewDecoder(r.Body).Decode(&body)
        failOnError(err, "Parsing JSON")
        body["origen"] = "Nats"
        data, err := json.Marshal(body)

        nc, err := nats.Connect("nats://servern:4222")
        if err != nil {
                log.Fatal(err)
        }
        defer nc.Close()
```

VM RabbitMQ

Main del receiver:

```
func main() {
        // Connecting to server
        conn, err := amqp.Dial("amqp://guest:guest@rabbitmq:5672/")
        failOnError(err, "Failed to connect to RabbitMQ")
        defer conn.Close()

        // Openning a channel
        ch, err := conn.Channel()
        failOnError(err, "Failed to open a channel")
        defer ch.Close()
```

Receiver envía el mensaje al Api Rest de Node:

```
go func() {
        for d := range msgs {
                log.Printf("Received a message: %s", d.Body)

                postBody := []byte(string(d.Body))
                req, err := http.Post("http://34.121.234.71:3000/subscribers", "application/json", bytes.NewBuffer(postBody))
                req.Header.Set("Content-Type", "application/json")
                failOnError(err, "POST new document")
                defer req.Body.Close()

                //Read the response body
                newBody, err := ioutil.ReadAll(req.Body)
                failOnError(err, "Reading response from HTTP POST")
                sb := string(newBody)
                log.Printf(sb)
        }
}()
```

Sender recibe peticiones http y se agrega el origen en el body (RabbitMQ):

```
func newElement(w http.ResponseWriter, r *http.Request) {
        // Adding headers
        w.Header().Set("Content-Type", "application/json")
        if r.Method == "GET" {
                w.WriteHeader(http.StatusOK)
                w.Write([]byte("{\"message\": \"ok\"}"))
                return;
        }

        // Parsing body
        var body map[string]interface{}
        err := json.NewDecoder(r.Body).Decode(&body)
        failOnError(err, "Parsing JSON")
        body["origen"] = "RabbitMQ"
        data, err := json.Marshal(body)
```

Main del Sender y handleRequest:

```
func handleRequests() {
        http.HandleFunc("/", newElement)
        log.Fatal(http.ListenAndServe(":3000", nil))
}

func main() {
        handleRequests()
}
```

VM Google PubSub

PUBLISHER:

MAIN Publisher:

```
func main(){
        fmt.Println("Server Google PubSub iniciado")

        http.HandleFunc("/", http_server)

        http_port := ":" + goDotEnvVariable("PORT")

    if err := http.ListenAndServe(http_port, nil); err != nil {
        log.Fatal(err)
    }
}
```

Enviar Variables:

```
func goDotEnvVariable(key string) string {

        err := godotenv.Load(".env")

        if err != nil {
          log.Fatalf("Error cargando las variables de entorno")
        }

        return os.Getenv(key)
}
```

Función utilizada para crear el Publisher:

```
func publish(msg string) error {
        projectID := goDotEnvVariable("PROJECT_ID")
        topicID := goDotEnvVariable("TOPIC_ID")

        ctx := context.Background()

        client, err := pubsub.NewClient(ctx, projectID)
        if err != nil {
                fmt.Println("Error encontrado")
                return fmt.Errorf("Error al conectarse %v", err)
        }

        t := client.Topic(topicID)

        result := t.Publish(ctx, &pubsub.Message {Data: []byte(msg), })

        id, err := result.Get(ctx)
        if err != nil {
                fmt.Println("error")
                fmt.Println(err)
                return fmt.Errorf("Error encontrado: %v",err)
        }

        fmt.Println("Published a message; msg ID: %v\n", id)
        return nil
}
```

HTML para los mensajes recibidos:

```
<html>

<head>
    <title>Pub/Sub</title>
</head>

<body>
    <div>
        <p>Last ten messages received by this instance:</p>
        <ul>
            {{ range . }}
            <li>{{ . }}</li>
            {{ end }}
        </ul>
    </div>
    <form method="post" action="/" name="formulary" target="_blank">
        <textarea name="msg" placeholder="Enter message here"></textarea>
        <input type="submit">
    </form>
</body>

</html>`))
```

SUSCRIBER:

MAIN Suscriber:

```
func main(){
	fmt.Println("A la espera de mensajes...")

	projectID := "august-edge-306320"
	subID := "mensaje"

	ctx := context.Background()

	client, err := pubsub.NewClient(ctx, projectID)
	if err != nil {
		//return fmt.Errorf("pubsub.NewClient: %v", err)
		log.Fatal(err)
	}

	// Consume 10 messages.
	var mu sync.Mutex
	received := 0
	sub := client.Subscription(subID)
	cctx, cancel := context.WithCancel(ctx)
	err = sub.Receive(cctx, func(ctx context.Context, msg *pubsub.Message) {
		mu.Lock()
		defer mu.Unlock()
		fmt.Println("Mensaje en suscriptor recevido: %q\n", string(msg.Data))

		sendMongo(string(msg.Data))

		msg.Ack()
		received++
		if received == 100 {
			cancel()
		}
	})
	if err != nil {
		//return fmt.Errorf("Receive: %v", err)
		log.Fatal(err)
	}
```

Función Enviar mensajes a MONGO:

```
func sendMongo(d string){
        postBody := []byte(string(d))
        req, err := http.Post("http://34.121.234.71:3000/subscribers", "application/json", bytes.NewBuffer(postBody))
        req.Header.Set("Content-Type", "application/json")
        failOnError(err, "POST new document")
        defer req.Body.Close()

        //Read the response body
        newBody, err := ioutil.ReadAll(req.Body)
        failOnError(err, "Reading response from HTTP POST")
        sb := string(newBody)
        log.Printf(sb)
}
```