Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Sistemas Operativos 1
Catedrático:
Auxiliar:
Primer Semestre 2021

Proyecto 1
Manual Tecnico

GRUPO 13

201313692	Jossie Bismarck Castrillo Fajardo
201314556	Carlos Gabriel Peralta Cambrán
201314697	Katherine Mishelle Serrano del Cid
201020331	Cristian Alexander Azurdia Ajú

LOCUST

La herramienta locust ejecuta un archivo python el cual realiza la lectura del archivo traffic.json que contiene la información para simular el ingreso de datos, genera un valor aleatorio entre 0 y el tamaño del array para enviar en cada petición el json que se encuentre en la posición aleatoria, enviándolo como un post a la dirección IP del balanceador de carga.

```python
import time
from locust import HttpUser, task, between
import json
from random import seed
from random import randint

class QuickstartUser(HttpUser):
    # this will set a random time between request to the server
    wait_time=between(1,10)

    # task enable the method to work in a loop.
    @task
    def on_start(self):
        reg=""
        # We read the Json file.
        with open("traffic.json") as file:
            data=json.load(file)
            # Generate a random value between 0 and the size of your Json file - 1

            # Get the value position of the Json load in data variable to send it in the request.
            print("tamaño del file: ",len(data))
            value=randint(0,len(data)-1)
            reg=data[value]
            print(reg)

        #self.client.get("http://localhost/")
        self.client.post("http://localhost/", json=reg)
```

El resultado de la informacion cargada con sus estadísticas

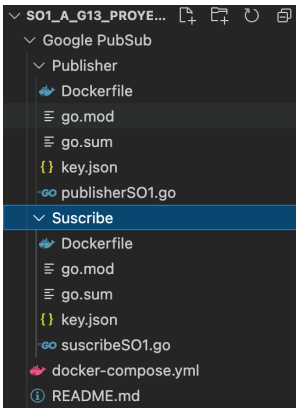
GRPC

- Cliente
  Para el cliente se habilitó una función ResponseWriter con métodos Get y Post, el Get servirá únicamente para verificar el estado del cliente que devolverá un Ok, el Post por el contrario tomara un paquete Json enviado a su IP al puerto donde está escuchando el cual será el 80 de la instancia mapeado hacia el 3000 del contenedor donde está el cliente, y enviará la solicitud al servidor por medio del mapeo en docker compose del servidor y su puerto.

- Servidor
  El servidor estará escuchando en el puerto 4000 de su contenedor al cual el cliente le envía las solicitudes post únicamente, el servidor posee un método post que obtiene el paquete del cliente y lo envía a la dirección IP de la instancia que contiene la API en Node Js.

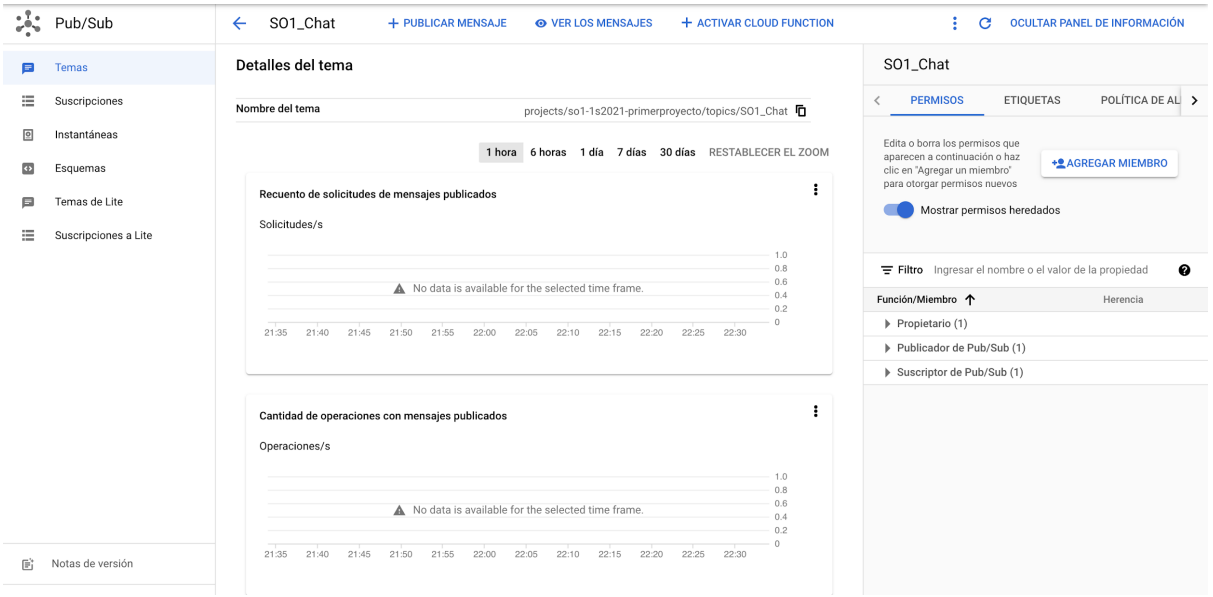
- Docker-Compose
  En la estructura de este archivo se definen 2 contenedores a los cuales se les asigna de nombre cliente y servidor, donde se mapea el para el cliente que redirecciona el tráfico del puerto 80 del host al puerto 3000 del contenedor, y un enlace al contenedor del servidor en lugar del uso de una dirección IP para la red interna de estos.

GOOGLE PUB SUB



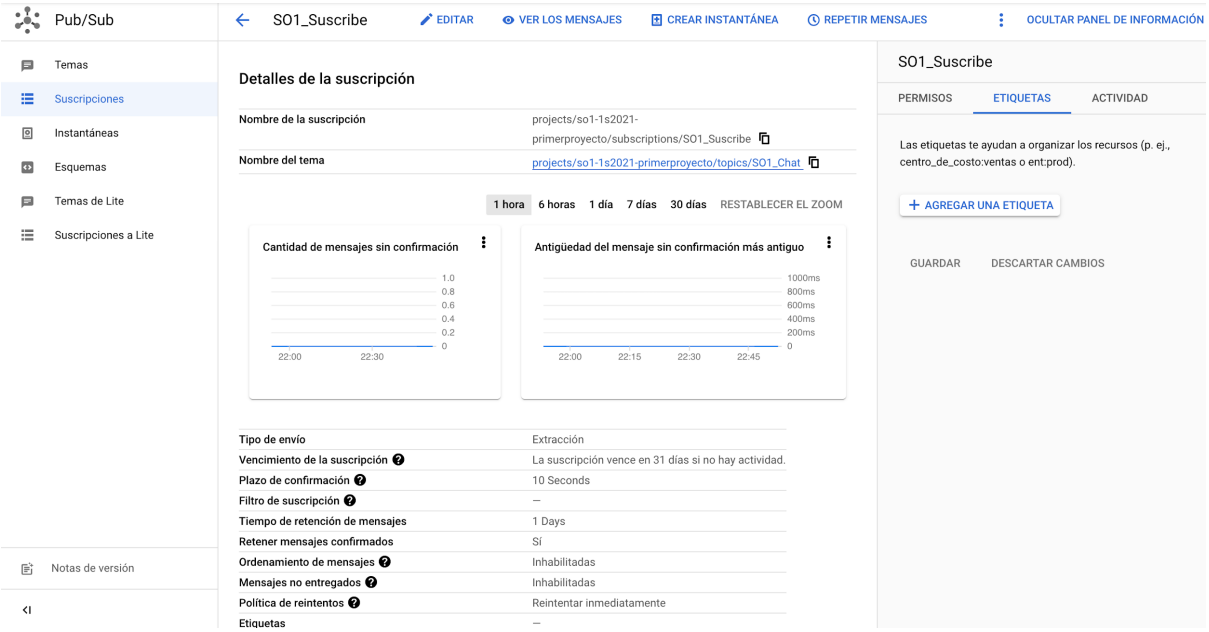
Temas:
Es la forma que google cloud permite realizar canales comunicación, estos canales permiten generar reglas e incluso encriptarlas bajo claves que la misma plataforma proporciona.

Cada tema puede manejar los servicios y suscriptores asignados, eliminar suscripciones o cambiar las reglas que se manejan en el tema, el tiempo en que se almacenarán los mensajes, políticas de reintento, filtros, etc.



Suscriptor
Una suscripción es el punto de salida que genera Google Cloud, las suscripciones están ligadas a un único tema. cuando se asigna un mensaje a un tema, este se “copia” a todas las suscripciones que tiene asignados el Tema.

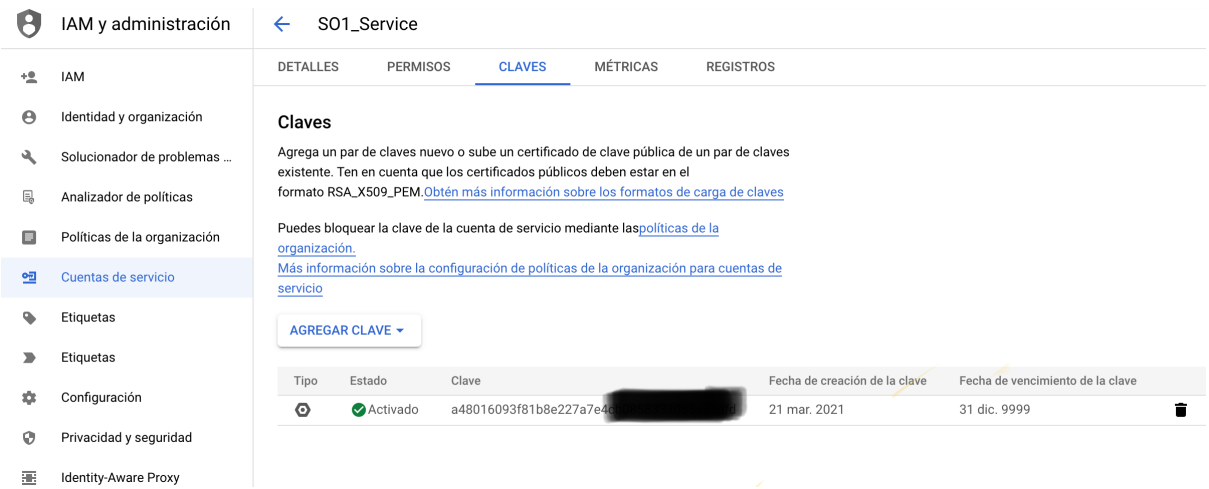
Las suscripciones permiten repetir mensajes a otros suscripciones así como generar etiquetas en base a filtros que el usuario puede definir.



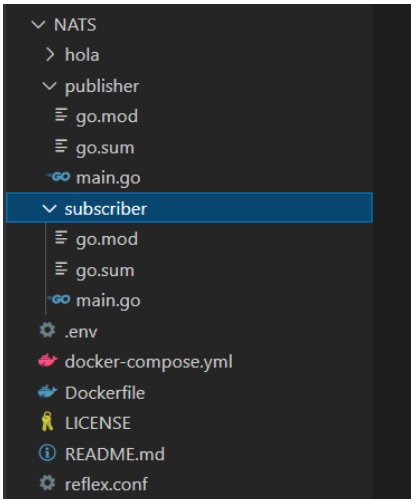
Servicio:

Es necesario generar una cuenta de servicio, esta cuenta, permite poder acceder a los servicios de temas y suscriptores creados a través de una llave que se genera dentro del servicio.

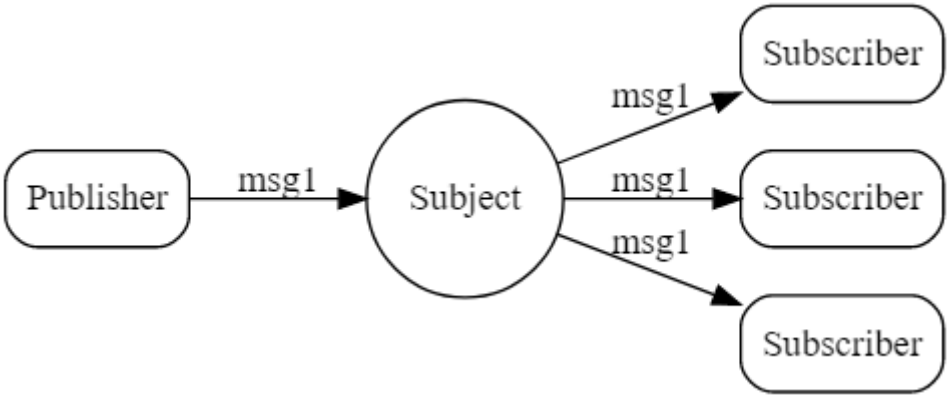
Un servicio también permite generar reportes para analizar el comportamiento que tienen los temas o los suscriptores generados y asociados al servicio.



NATS



NATS implementa un modelo de distribución de mensajes de publicación-suscripción para la comunicación de uno a varios.



Para el proyecto se utilizó la herramienta Nats la cual se puede descargar de la página oficial https://hub.docker.com/_/nats

Se crea un archivo docker-compose donde podemos ver la dirección de la parte de configuración del publisher y del subscriber:

docker-compose.yml

NATS > docker-compose.yml

```yaml
version: '3'
services:
  publisher:
    # building image from ./Dockerfile
    build: .
    volumes:
      - ./publisher:/app
    working_dir: /app
    env_file:
      - .env
    # ports exposed to localhost
    ports:
      - 80:5000

  subscriber:
    # building image from ./Dockerfile
    build: .
    volumes:
      - ./subscriber:/app
    working_dir: /app
    env_file:
      - .env

  nats:
    image: nats-streaming:0.16.2
    restart: on-failure

```

En la parte del publish:

Con este método se puede ir publicando lo que va a estar llegando a los subscribers.
en el cual también se le agrega el camino de por donde va pasando.

```go
}

func (h handler) publish(w http.ResponseWriter, r *http.Request) error {

    w.Header().Set("Content-Type", "application/json")
    var body map[string]interface{}
    err := json.NewDecoder(r.Body).Decode(&body)
    body["CAMINO"] = "NATS"
    data, err := json.Marshal(body)

    payload, err := ioutil.ReadAll(r.Body)
    if err != nil {
        return err
    }
    uuid := ulid.MustNew(ulid.Timestamp(time.Now()), rand.Reader)
    msg := message.NewMessage(uuid.String(), payload)
    if err := h.publisher.Publish(h.topic, msg); err != nil {
        return err
    }
    postBody := []byte(string(data))
    http.Post("http://104.196.23.85:80", "application/json", bytes.NewBuffer(postBody))
    _, err = fmt.Fprint(w, string(data))
    if err != nil {
        return err
    }
    return nil
}
```

En la parte del suscribe:

Se verifica como los clientes los cuales van a estar recibiendo lo que se está publicando.

```go
func startSubscriber(natsURL, clusterID, topic string) error {
    logger := watermill.NewStdLogger(false, false)
    router, err := message.NewRouter(message.RouterConfig{}, logger)
    if err != nil {
        return err
    }
    subscriber, err := nats.NewStreamingSubscriber(
        nats.StreamingSubscriberConfig{
            ClusterID:   clusterID,
            ClientID:    "subscriber",
            QueueGroup:  "example-group",
            DurableName: "example-durable",
            StanOptions: []stan.Option{
                stan.NatsURL(natsURL),
            },
            Unmarshaler: nats.GobMarshaler{},
        },
        logger,
    )
    if err != nil {
        return err
    }
    router.AddMiddleware(middleware.Recoverer)
    router.AddNoPublisherHandler(
        "messages_handler",
        topic,
        subscriber,
        handler,
    )
    log.Print("Subscribed for messages")
    return router.Run(context.Background())
}
```

RABBIT MQ

RabbitMQ es un message broker bastante popular, utiliza el protocolo Advanced Message Queuing Protocol (AMQP). Esto permite mantener una transmisión sólida de los datos y que utiliza una cola para almacenar los mensajes. Esto a su vez, brinda comunicación asíncrona entre emisor y receptor (publisher and subscriber).

Su implementación se reduce a tres contenedores desplegados en una misma máquina virtual. A nivel interno se compone de:

- RabbitMQ Server
    - Es el encargado de permitir la comunicación entre el publisher y subscriber
    - Utiliza el puerto 5672 para comunicación interna con el protocolo AMQP
- RabbitMQ Publisher
    - Es el encargado de emitir el mensaje en protocolo AMQP, luego de recibir una petición HTTP POST en el puerto 5000
    - Envía el mensaje al Server para ingresarlo en la cola especificada
- RabbitMQ Subscriber

- Es el encargado de escuchar al Server y consumir los mensajes que se encuentren en la cola
- Recibe el mensaje del Server y procede a hacer la petición a la API de NodeJS

RabbitServer

Para el Server fue necesario obtener la imagen disponible de RabbitMQ en Docker Hub, posterior a ello se creó el contenedor y únicamente se permite el tráfico interno dentro de la Docker Network. Se utilizó la versión 3 y el puerto 5672 para permitir el tráfico en protocolo AMQP.

```
carlospecam@rabbitmq:~$ sudo docker ps
CONTAINER ID        IMAGE
badb06b1fa4a        rabbitmq_subscriber-rabbit
f3ecd0e889cf        rabbitmq_publisher-rabbit
d9797416b544        rabbitmq
carlospecam@rabbitmq:~$
```

RabbitMQ Publisher

- **Función handleRequests**
  Es una función que permite escuchar en el puerto 5000 las peticiones HTTP que sean realizadas. Cabe destacar que la petición va a root “/” y se utiliza la función newElement para procesar los datos enviados.

```go
func handleRequests() {
    http.HandleFunc("/", newElement)
    log.Fatal(http.ListenAndServe(":5000", nil))
}
```

- **Función newElement**

  Esta función se encarga de verificar que la petición sea POST, para continuar con el proceso decodificación del body de la petición. Esto con el objetivo de agregar el key CAMINO, ya que es utilizado para indicar que la petición redirigida por el Load Balancer fue enviada a RabbitMQ.

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

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

Seguido de añadir la key CAMINO y codificar el JSON nuevamente, se procede a crear la conexión con el Server. Utilizando el protocolo AMQP a través de la función Dial. Si se establece una conexion se procede con la creación del canal AMQP además de la especificación de la cola que contendrá los mensajes.

```
conn, err := amqp.Dial("amqp://guest:guest@rabbit-server")
failOnError(err, "RabbitMQ connection")
defer conn.Close()

amqpchannel, err := conn.Channel()
failOnError(err, "New channel connection")
defer amqpchannel.Close()

// Declaring a new Queue
queue, err := amqpchannel.QueueDeclare(
    "rabbitqueue",
    false,
    false,
    false,
    false,
    nil,
)
failOnError(err, "Failed to declare a queue")
```

Si los tres procesos son exitosos se publicará un nuevo mensaje a través del amqpchannel en la cola deseada, en este caso rabbitqueue. El mensaje es enviado en formato plain text a través de un arreglo de bytes dentro del body del request.

```
    // Publishing a new message
    newData := string(data)
    err = amqpchannel.Publish(
        "",
        queue.Name,
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(newData),
        })
    failOnError(err, "Failed to publish a message")
    log.Printf(" [x] Sent %s", newData)

    w.WriteHeader(http.StatusCreated)
    w.Write([]byte(newData))
}
```

Si el proceso se completa de forma exitosa, se garantiza la comunicación asíncrona entre el publisher y el subscriber.

RabbitMQ Subscriber

- **main**
  En este caso, la función main es la encargada de ejecutar todo el trabajo para consumir la cola. Se debe establecer la conexión al Server con el protocolo AMQP, crear un canal (amqpchannel) por el cual se consumirán los mensajes y la especificación de la cola que se debe escuchar.

```
nc main() {
 conn, err := amqp.Dial("amqp://guest:guest@rabbit-server")
 failOnError(err, "Failed to connect to RabbitMQ")
 defer conn.Close()

 ampqchannel, err := conn.Channel()
 failOnError(err, "Failed to open a channel")
 defer ampqchannel.Close()

 queue, err := ampqchannel.QueueDeclare(
     "rabbitqueue",
     false,
     false,
     false,
     false,
     nil,
 )
 failOnError(err, "Failed to declare a queue")

 messagechannel, err := ampqchannel.Consume(
     queue.Name,
     "",
     true,
     false,
     false,
     false,
     nil,
 )
 failOnError(err, "Failed to register a consumer")
```

Luego de realizadas las configuraciones requeridas para escuchar los mensajes enviados por el publisher, es necesario el uso de una función anonima que se encargará de procesar los mensajes y realizar una petición POST a la API de NodeJS encargada de almacenar los datos en la capa de persistencia de MongoDB

```
    forever := make(chan bool)

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

            postBody := []byte(string(d.Body))
            req, err := http.Post("http://34.69.47.240:80/", "application/json", bytes.NewBuffer(postBody
            req.Header.Set("Content-Type", "application/json")
            failOnError(err, "POST new document")
            defer req.Body.Close()

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

    log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
    <-forever
}
```

La ejecución de estas sentencias se realiza por siempre, de manera que el subscriber siempre se encontrará consultando por nuevos mensajes almacenados en la cola rabbitqueue.

SERVIDOR

La API intermedia consiste de 5 métodos para su funcionamiento funcionando en el puerto 5000 de su contenedor mapeado hacia el puerto 80 de la instancia:

- Verificación de estado
  Es una petición tipo GET que únicamente se utiliza para verificar que el API esté activa, teniendo como respuesta un estado OK.

```
app.get('/', (req, res) => {
    res.json({message: 'OK'})
});
```

- Inserción a Mongo

  Es una petición tipo POST que recibe las peticiones de las 4 rutas anteriores en formato Json, al obtener la petición obtiene la variable de conexión hacia el contenedor de MongoDB, en la colección establecida y envía la información esperando un Json de respuesta con la llave asignada en Mongo.

```
app.post('/', async (req, res) => {
    const data = req.body;
    try {
        let collection = db.collection("personita2");
        let result = await collection.insertOne(data);
        console.log("ingreso algo",data);
        res.json(result.ops[0]);
    } catch (err) {
        console.log(err);
        res.status(500).json({ 'message': 'failed' });
    }
});
```

- Consulta a Mongo
  Es una petición tipo GET que consulta la ruta /data y envía una consulta a la base de datos, la cual devuelve un arreglo con toda la información obtenida desde mongo.

```
app.get('/data', async (req, res) => {
    var dat=[];
    try {
        db.collection("personita2").find({}).toArray(function(err, result) {
            if (err) throw err;
            console.log(result);
            res.json(result);
          });

    } catch (err) {
        console.log(err);
        res.status(500).json({ 'message': 'failed' });
    }
});
```

- Lectura de Memoria
  Es una petición GET que consulta la ruta /mem, la cual realiza la lectura del archivo instalado en /proc del contenedor mapeado hacia el host para obtener los datos actualizados de la memoria del host.

```
app.get('/mem', async (req, res) => {
    var dat="";
    try {
        var archivo="/proc/mem_grupo13";
        fs.readFile(archivo, 'utf8' , (err, data) => {
            if (err) {
              console.error(err);
              return
            }
            dat=JSON.parse(data);
            console.log("salida1"+dat);
            res.json(dat);
          });
          console.log("salida2"+dat);

    } catch (err) {
        console.log(err);
        res.status(500).json({ 'message': 'failed reading' });
    }
});
```

- Lectura de Procesos
  Es una petición GET que consulta la ruta /proc, la cual realiza la lectura del archivo instalado en /proc del contenedor mapeado hacia el host para obtener los datos actualizados de la memoria del host.

```
app.get('/proc', async (req, res) => {
    var dat="";
    try {
        var archivo="/proc/proc_grupo13";
        fs.readFile(archivo, 'utf8' , (err, data) => {
            if (err) {
              console.error(err);
              return
            }
            //dat=JSON.parse(data);
            console.log("salida1"+data);
            //res.json(dat);
            res.send(data)
          });
          console.log("salida2"+dat);

    } catch (err) {
        console.log(err);
        res.status(500).json({ 'message': 'failed reading' });
    }
});
```

- Mongo DB
  Para la creación del contenedor con MongoDB se realiza desde el archivo docker-compose, en el cual a la API se le agrega como un link el contenedor de base de datos para el uso de la red y en el mismo archivo se crea el volumen compartido con el host.

```
version: "3.4"
services:
  api:
    container_name: ejemplo
    restart: always
    build: ./API
    ports:
      - "80:5000"
    links:
      - mongodb
    volumes:
    - /proc:/proc

  mongodb:
    container_name: mimongo
    image: mongo
    ports:
      - "8080:27017"
```

MÓDULOS KERNEL

Para la creación de ambos módulos de Kernel se crearon 2 archivos tipo C llamados mem_grupo13.c y proc_grupo13.c en los cuales en ambos se importaron primero los headers que son parte del sistema Linux para la manipulación de datos del sistema

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/sched/signal.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/sysinfo.h>
#include <linux/seq_file.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/swap.h>
```

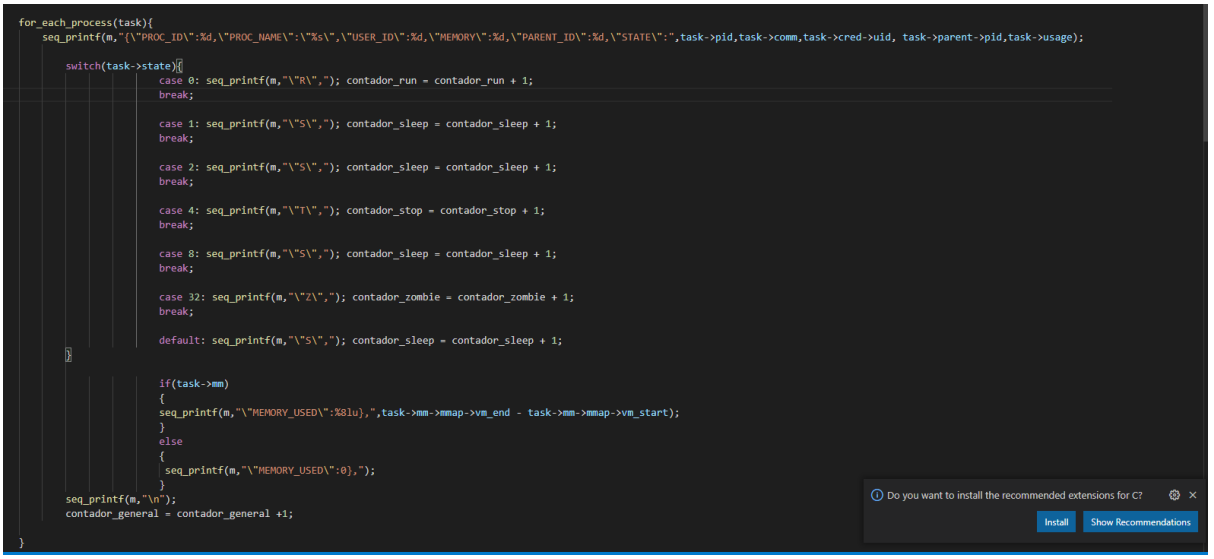
- Memoria

  Para el monitor de memoria se realizó la llamada de las variables del struc seq_file para obtener el total de ram, la ram libre y el calculo de su porcentaje de consumo construyendo en la misma función cada dato en formato Json que se escribirá en el archivo de la carpeta /proc.

```c
static int meminfo_proc_show(struct seq_file *m, void *v)
{
        struct sysinfo i;
        si_meminfo(&i);
        // datos que se escriben en el Json
        seq_printf(m,"{\"MemoriaTotal\":%8lu,\n",i.totalram);
        seq_printf(m,"\"MemoriaLibre\":%8lu,\n",i.freeram);
        seq_printf(m,"\"PorcentajeConsumo\":%lu}",((i.totalram-i.freeram)*100)/i.totalram);
        return 0;
}
```

- Procesos

  Para el monitor de procesos se utilizo for each para leer el registro “task” que los contiene para construir de forma iterativa un json con los datos solicitados y una variable para definir abreviadamente cada estado.



```
for_each_process(task){
    seq_printf(m,"{\"PROC_ID\":%d,\"PROC_NAME\":\"%s\",\"USER_ID\":%d,\"MEMORY\":%d,\"PARENT_ID\":%d,\"STATE\":",task->pid,task->comm,task->cred->uid, task->parent->pid,task->usage);

        switch(task->state){
                    case 0: seq_printf(m,"\"R\","); contador_run = contador_run + 1;
                    break;

                    case 1: seq_printf(m,"\"S\","); contador_sleep = contador_sleep + 1;
                    break;

                    case 2: seq_printf(m,"\"S\","); contador_sleep = contador_sleep + 1;
                    break;

                    case 4: seq_printf(m,"\"T\","); contador_stop = contador_stop + 1;
                    break;

                    case 8: seq_printf(m,"\"S\","); contador_sleep = contador_sleep + 1;
                    break;

                    case 32: seq_printf(m,"\"Z\","); contador_zombie = contador_zombie + 1;
                    break;

                    default: seq_printf(m,"\"S\","); contador_sleep = contador_sleep + 1;
        }

                    if(task->mm)
                    {
                    seq_printf(m,"\"MEMORY_USED\":%8lu},",task->mm->mmap->vm_end - task->mm->mmap->vm_start);
                    }
                    else
                    {
                     seq_printf(m,"\"MEMORY_USED\":0},");
                    }
        seq_printf(m,"\n");
        contador_general = contador_general +1;
}
```


1. <u>Cambio de privilegios en las carpetas MEMORIA Y PROCESOS</u>
   chmod +w+x+r MEMORIA
   chmod +w+x+r PROCESOS

2. <u>INSTALAR GCC</u>
   sudo apt update
   sudo apt install build-essential
   sudo apt-get install manpages-dev
   gcc --version

3. <u>INSTALAR MAKE</u>
   sudo apt install make
   sudo apt install build-essential
   ls /usr/bin/make
   /usr/bin/make --version
   echo $PATH
   sudo dpkg-reconfigure make

4. <u>INSTALAR LINUX-HEADERS</u>
   sudo apt update
   sudo apt install linux-headers-$(uname -r)
   ls -l /usr/src/linux-headers-$(uname -r)

5. <u>INSTALAR MODULO EN HOST</u>
   En cada carpeta de los módulos ejecutar : Make
   insmod mem_grupo13.ko
   insmod proc_grupo13.ko