Instituto Tecnológico de Costa Rica

Campus Tecnológico Central Cartago

Escuela de Ingeniería en Computación

Proyecto Opcional Redes

Redes - Grupo 2

Prof. Gerardo Nereo Campos Araya

Daniel Granados Retana, carné 2022104692

Diego Manuel Granados Retana, carné 2022158363

David Fernández Salas, carné 2022045079

Diego Mora Montes, carné 2022104866

Eduardo Gutierrez Conejo, carné 2019073558

14 de Marzo del 2024

IS 2025

Tabla de contenidos {#tabla-de-contenidos}

1. Introducción

Este proyecto consiste en implementar una arquitectura distribuida y automatizada utilizando tecnologías de contenedores y orquestación de Kubernetes. Además se divide el sistema en dos entornos lógicos llamados Namespaces, donde uno es público, que expone servicios accesibles a los usuarios externos, y uno privado, donde se alojan los servicios internos protegidos.

El objetivo es diseñar una infraestructura escalable y modular que permita la comunicación entre múltiples servicios, incluyendo servidores web y una central telefónica basada en Asterisk. Para conseguir esta tarea se necesita emplear herramientas como Docker, Helm y Kubernetes permitiendo un despliegue automatizado, fácil y escalable. Además, se establecen reglas de enrutamiento con `iptables` y un controlador Ingress para gestionar el tráfico entre los diferentes componentes del sistema.
La automatización en entornos de red es fundamental porque permite optimizar la gestión, escalabilidad y resiliencia de diversos servicios. Mediante la automatización se crea un proceso mucho más seguro ante errores humanos, mejorando la eficiencia operativa y garantizando una mayor disponibilidad. Por esta razón es de gran importancia que todo el sistema esté automatizado y que este se encargue del despliegue de todos los servicios. En la implementación de Kubernetes, esto fue posible gracias a la herramienta Helm. Adicionalmente, se adaptó la implementación de Kubernetes para ser desplegada en la nube por medio de recursos análogos de AWS. Para la construcción automatizada del ambiente y de los objetos necesarios, se utilizó la herramienta de Terraform.

2. Instalación y configuración

2.1. Instalación de herramientas

Para correr el proyecto, se deben instalar las siguientes herramientas:

1. Windows Subsystem for Linux
2. Helm
3. Docker
4. Kubernetes: Se puede hacer desde Docker Desktop.
5. Zoiper5 y MicroSIP
6. Lens
7. Git Bash
8. Terraform

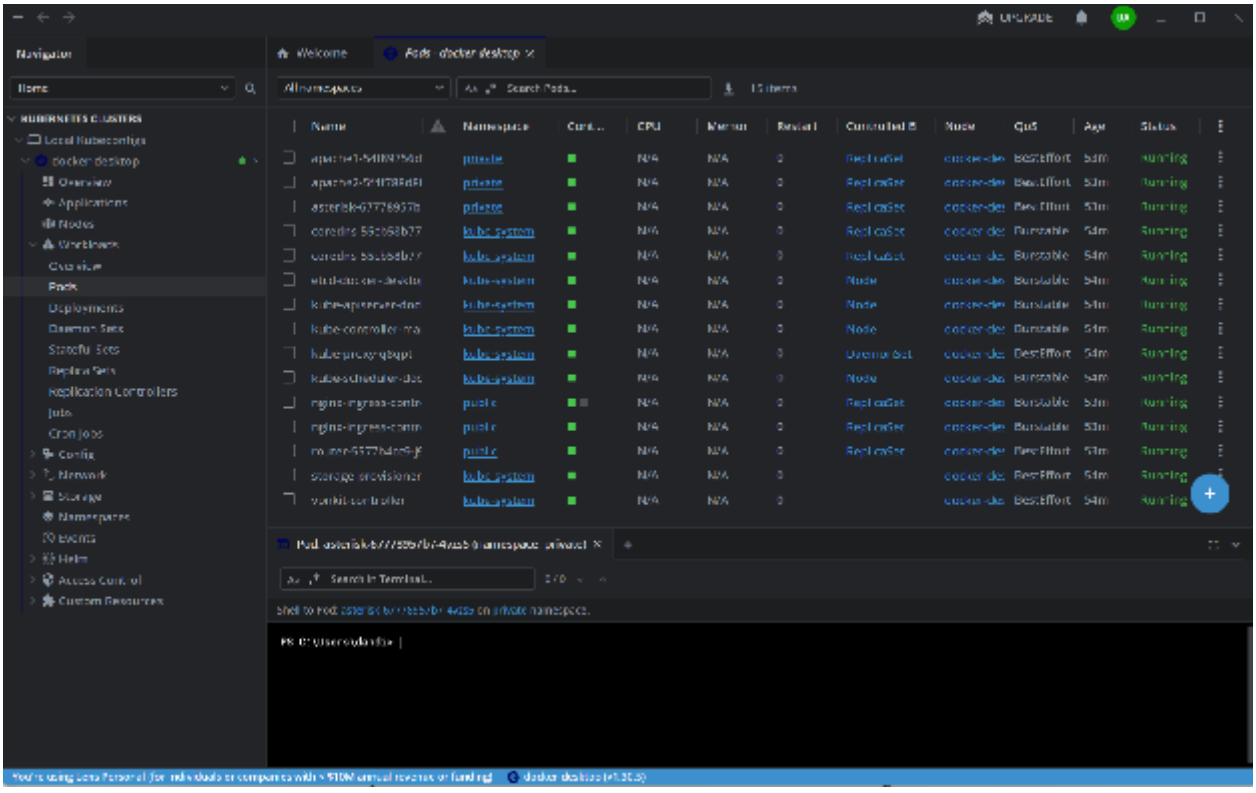
2.2. Instalación del proyecto en Kubernetes

Para facilitar la instalación del proyecto, escribimos un script que instala todos los componentes automáticamente. Por lo tanto, para instalar se debe:

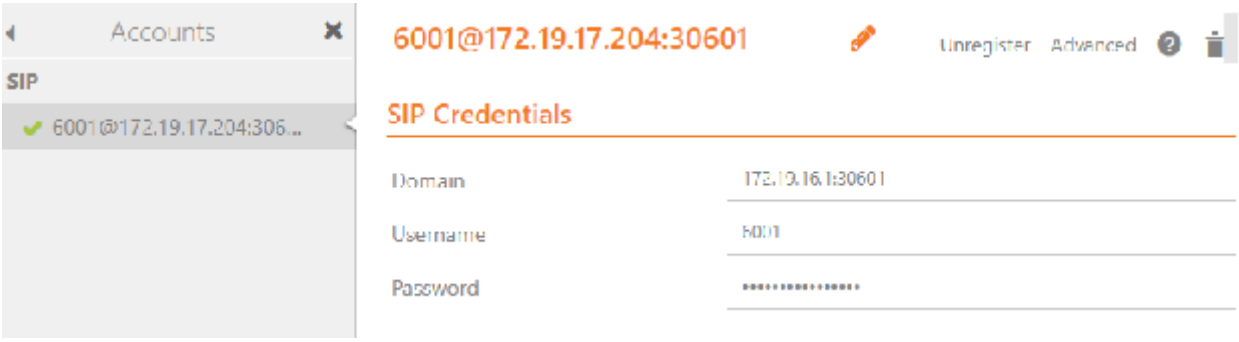
1. Descargar el repositorio con `git clone` o descomprimir el código fuente.
2. Correr Docker con Kubernetes. Esto se puede realizar de manera muy sencilla, activando la opción de “Activar Kubernetes” en la configuración de Docker Desktop.
3. Navegar a la dirección `\2025-01-2019073558-IC7602\po\charts`
4. Ejecutar el script de instalación con git bash: `./install.sh`
5. Este instala todos los Helm charts, tanto los del namespace público como privado.
6. Si se quiere revisar el progreso de los pods se puede ejecutar los siguientes comandos “kubectl get pods -n private” para las pods privadas y “kubectl get pods -n public” para las pods públicas.
7. El script también imprime la dirección IP del WSL. Esta se usa para configurar el teléfono SIP.

```
dandi@Legion MINGW64 ~/Redes/2025-01-IC7602/po/charts (main)
$ ./install.sh
Running script from: /c/Users/dandi/Redes/2025-01-IC7602/po/charts
WSL IP: 172.22.160.1
Release "namespace" does not exist. Installing it now.
NAME: namespace
LAST DEPLOYED: Thu Mar 13 15:36:56 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
Release "private" does not exist. Installing it now.
NAME: private
LAST DEPLOYED: Thu Mar 13 15:37:06 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
Release "public" does not exist. Installing it now.
NAME: public
LAST DEPLOYED: Thu Mar 13 15:37:27 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

8. Se puede verificar que todo esté corriendo bien en Lens. En la sección de Workloads, deberían aparecer todos los pods en los namespaces de public y private.

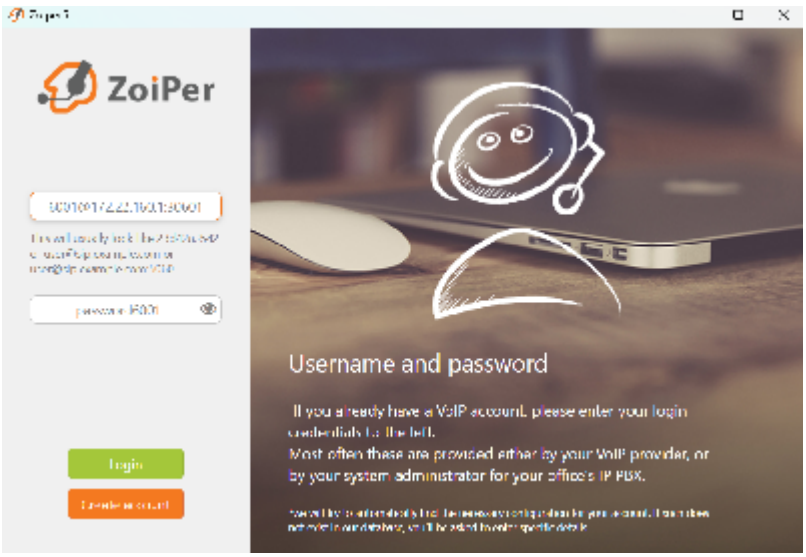


9. Para poder conectar dos teléfonos en la misma computadora, se necesitan dos teléfonos SIP. Se recomienda usar Zoiper5 y MicroSIP.
10. Para configurar Zoiper, hay que entrar a la configuración, después entrar a Accounts y aquí se puede añadir un número. Aquí, se tiene que poner en el Domain la dirección IP del WSL y el puerto de 30601, que es el puerto del NodePort que conecta Asterisk con el exterior. Luego, como usuario, hay dos opciones: 6001 y 6002. En uno de los teléfonos se debe usar uno u el otro. La contraseña que se de pone es `password####`, donde `####` es el número que tiene esa cuenta. La configuración se podría ver así:

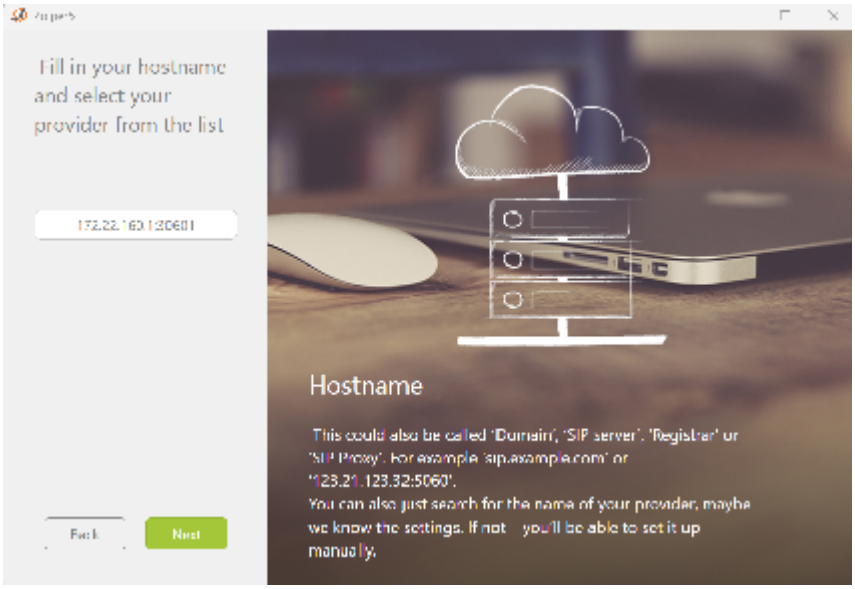


Luego, hay que presionar Register.

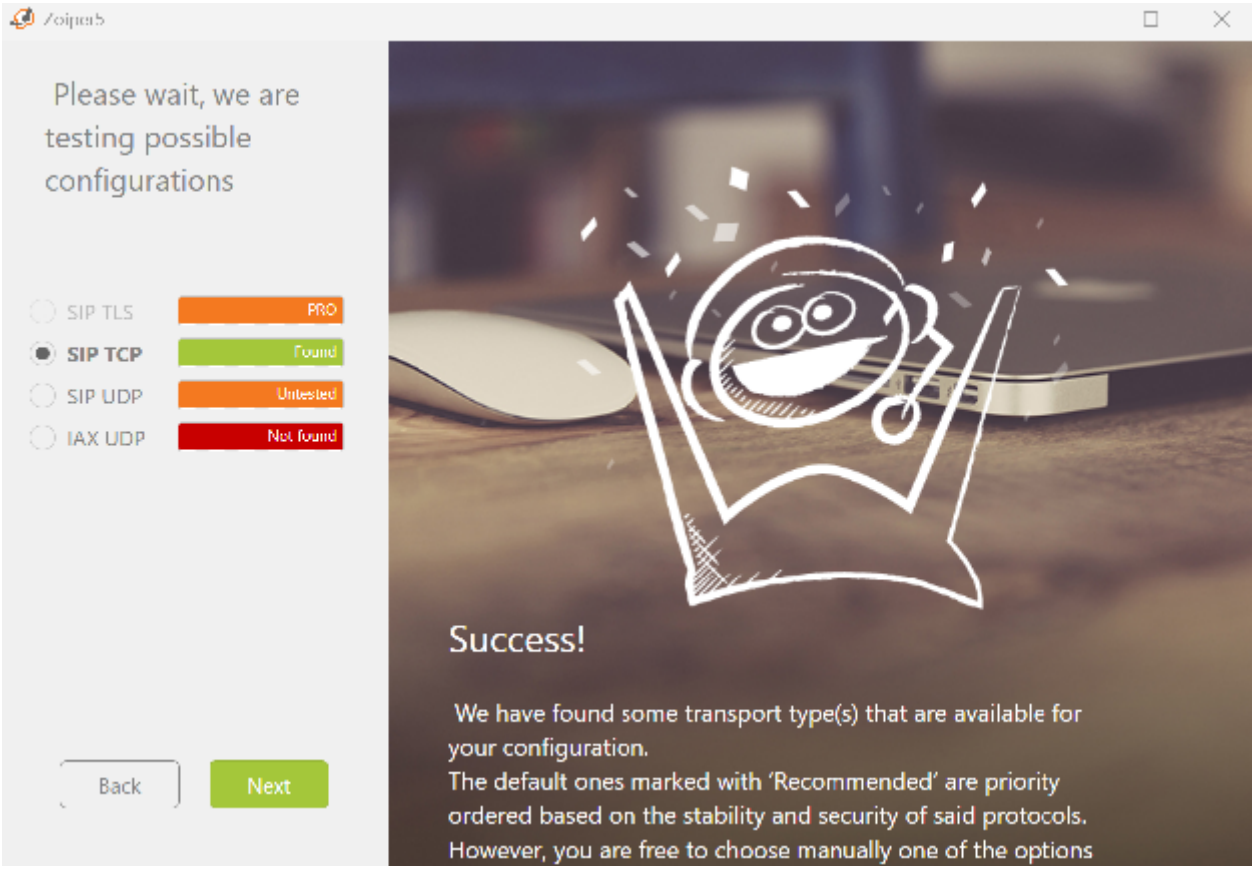
1. En el caso de que se está registrando Zoiper desde un inicio, la interfaz es ligeramente distinta. Primero, solicita activar una licencia “Pro”. Esto no es necesario. Se debe seleccionar la opción de “Continuar como usuario gratuito”.
2. Luego, Zoiper da la opción de ingresar un nombre de usuario y una contraseña. Abajo está el botón de crear cuenta. Esto da la idea de que es necesario crear una cuenta. No obstante, este no es el caso. El usuario y la contraseña que se ingresan son los de la cuenta de Asterisk. En usuario, se pone `usuario@dominio:puerto`. El dominio se refiere a la dirección IP que se imprimió con el script de instalación de los helm charts. La configuración se podría ver así:



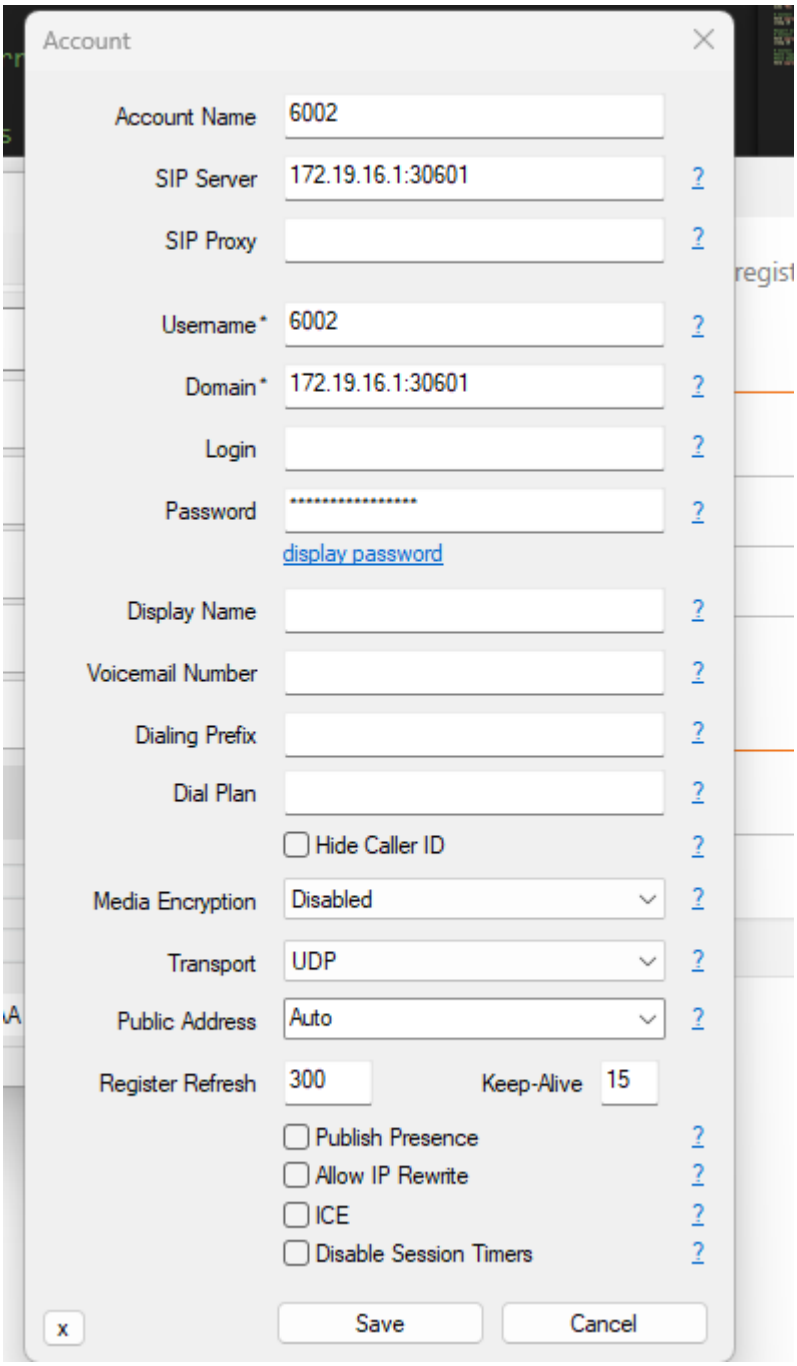
3. Después, si se ingresó el usuario bien, el nombre del anfitrión debería llenarse solo automáticamente:



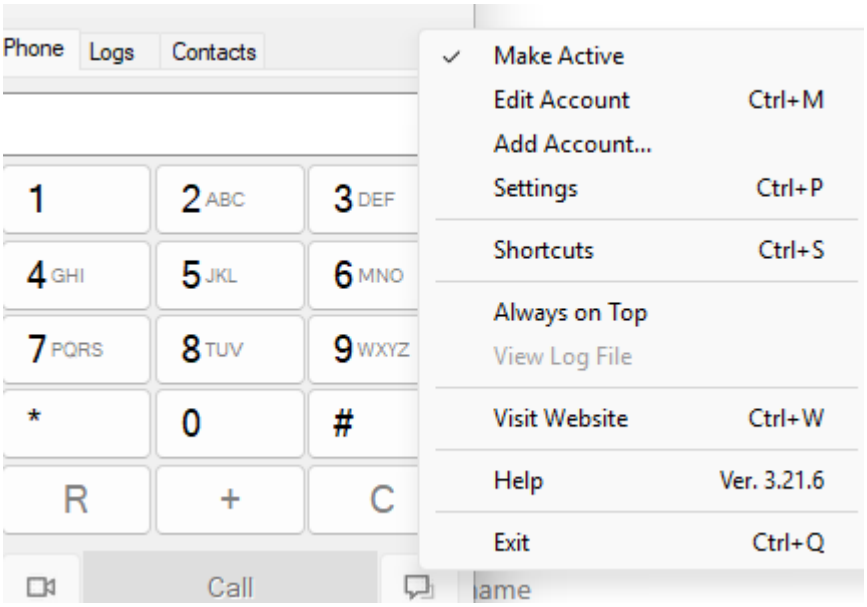
4. Cuando solicita autenticación, se puede saltar esa pestaña con el botón de “Skip”:
5. Después, Zoiper va a empezar a probar las posibles configuraciones de acuerdo con el protocolo. Cuando encuentre una conexión válida, el protocolo pasa a estar marcado como “Found”.



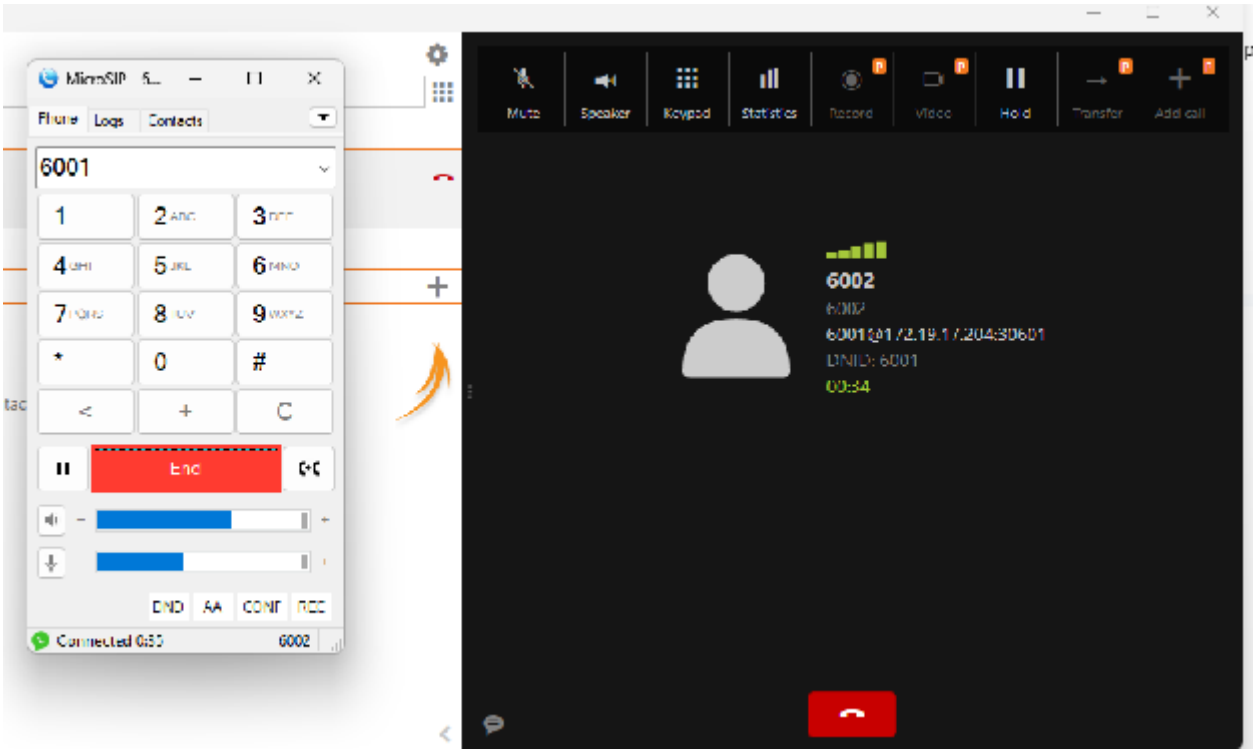
6. Al apretar “Next”, ya la cuenta queda configurada y registrada.
7. Para configurar MicroSIP, es muy similar. Se presiona en la flecha y se presiona Add o Edit Account. Aquí se puede poner la información del Domain y hay que poner la misma información en el SIP Server. Se debe ingresar en el Account Name el otro número. En nuestro caso, 6002.



Luego hay que presionar Save y volver a presionar en la flecha superior y presionar “Make Active”.



12. Finalmente, para llamarse entre sí, se debe llamar al número contrario en uno de los teléfonos. Ya con esto, se pueden comunicar.

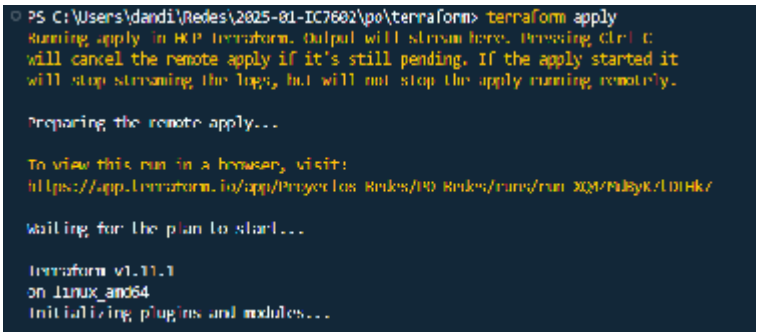


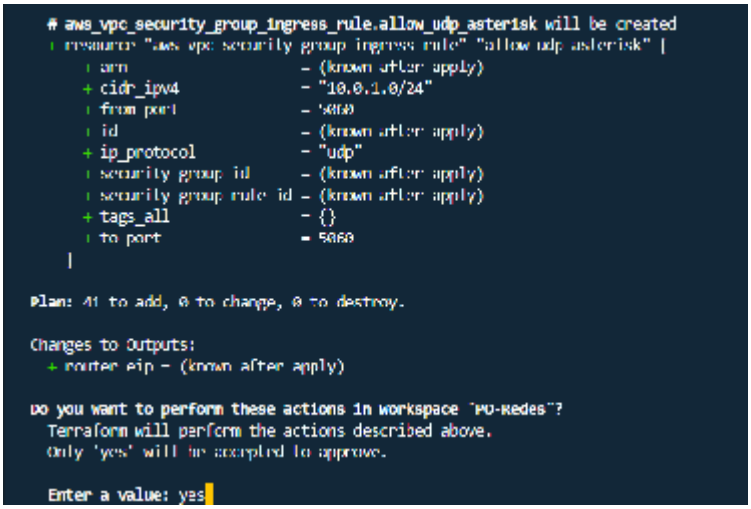
13. Para entrar a los servidores Apache, se entra al localhost:30080 y 30081.

2.3. Instalación con Terraform

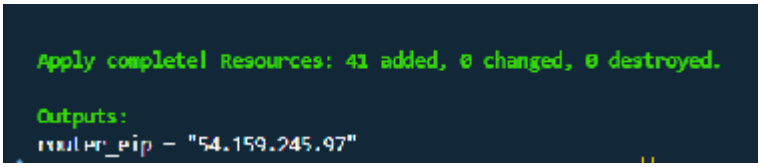
Instalar el proyecto en AWS con Terraform es igual de sencillo.

1. Es necesario configurar el proyecto en Terraform.
   a. Se deben obtener las credenciales de AWS, específicamente el AWS_ACCESS_KEY_ID y el AWS_SECRET_ACCESS_KEY. Estos se pueden obtener siguiendo la guía de AWS de cómo obtener las llaves de acceso.
   b. Si se desea almacenar el estado de Terraform en un repositorio compartido, se puede realizar esto con HCP Terraform. Se puede seguir la guía para configurar el ambiente.
   c. Si se desea almacenar el estado localmente, se puede seguir esta guía. En ella, se muestra cómo configurar las variables de entorno de AWS para configurar el ambiente.
2. Hay que inicializar el proyecto de terraform con el comando: `terraform init`.
3. Para desplegar la infraestructura, se ejecuta el comando: `terraform apply`. Se van a desplegar una serie de cambios. Se escribe la palabra “yes” para confirmar los cambios. Alternativamente, se puede usar el comando `terraform apply -auto-approve`.





4. Esto eventualmente imprime una dirección IP, la cual está compuesta de una serie de números. Por ejemplo, podría ser algo como 54.159.245.97.



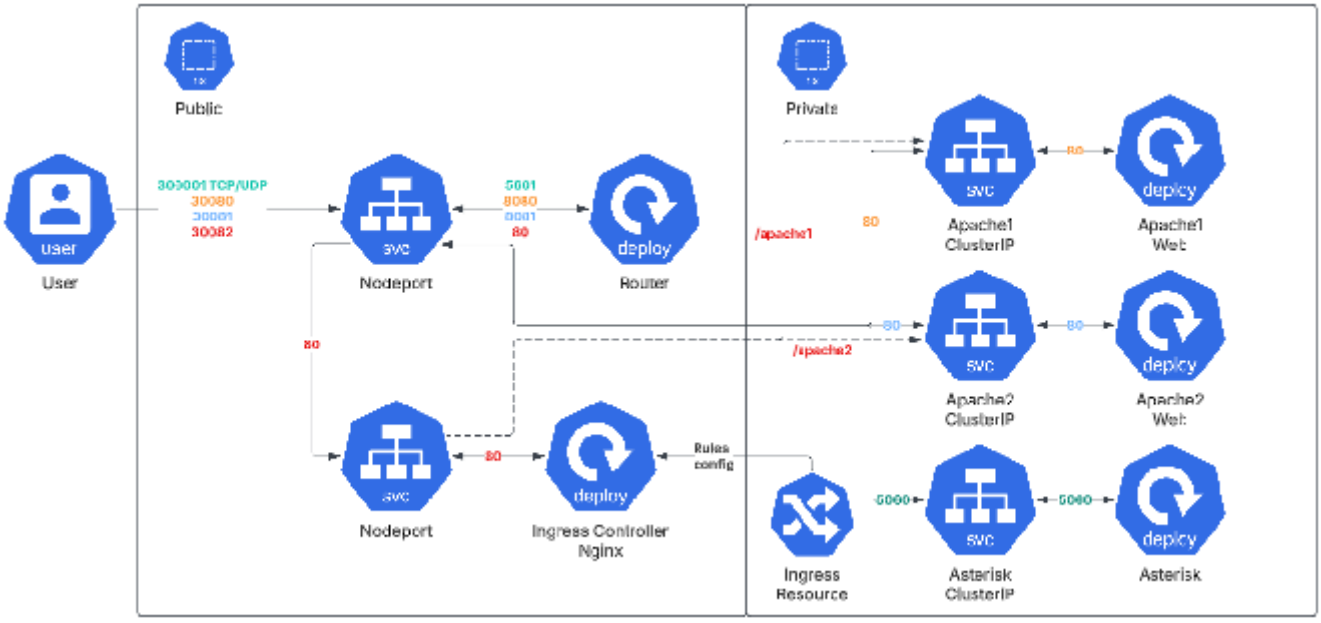
5. Con esta dirección IP, se puede registrar en Asterisk siguiendo el mismo proceso descrito a partir del paso 9 en la sección 2.2. Es importante destacar que el puerto sería el 5601. Así, se registraría la cuenta con el dominio o SIP server de 54.159.245.97:5601, como se muestra en la siguiente imagen:

6. Suponiendo que la dirección IP obtenida fue ,para acceder a los servidores de Apache directamente, se utiliza la dirección http://54.159.245.97:8080 para el primer servidor y http://54.159.245.97:8081 para el segundo servidor.
7. Para acceder a los servidores de Apache a través del controlador de Nginx, se accede a la dirección http://54.159.245.97/apache1 y http://54.159.245.97/apache2

3. Arquitectura del sistema

3.1. Kubernetes

El proyecto se organiza en dos entornos lógicos separados mediante Namespaces: Público y Privado. El siguiente diagrama muestra la relación entre los diferentes componentes del sistema, destacando las rutas de comunicación entre ellos y cómo interactúan dentro del clúster.



3.1.1 Namespace y segmentación de servicios

Namespace público: Este se encarga de guardar los componentes externos, los cuales pueden ser accedidos por el usuario. Su función principal es enrutar el tráfico y gestionar la accesibilidad de los servicios internos. Este namespace lo compone el router y el Ingress controller de Nginx los cuales son los únicos que se comunican con el usuario.

Namespace Privado: Contiene los servicios internos protegidos, que no son accesibles directamente desde el exterior. En este caso son los servicios Apaches y Asterisk.

3.1.2 Componentes del sistema

Públicos:
Router: Actúa como un punto de entrada que redirige el trabajo a los servicios internos. En este caso tiene los siguientes puertos:

- TCP/5601 y UDP/5601 → Asterisk
- TCP/8080 → Apache1
- TCP/8081 → Apache2
- TCP/80 → Ingress Controller

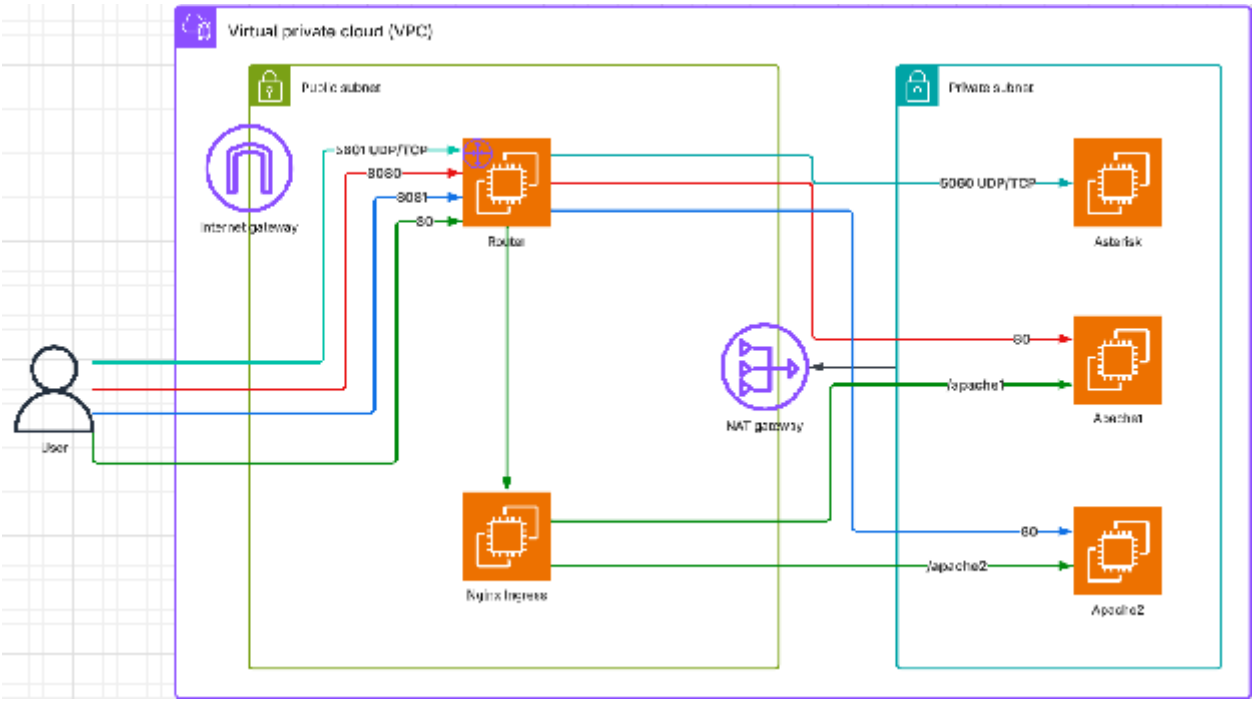
Ingress Controller: Se encarga de la manipulación y redirección de las solicitudes HTTP en base a la URL. Básicamente permite redirigir a los siguientes servicios

- `/apache1` → Apache1

- `/apache2` → Apache2

Privados:
Apaches: Muestran un mensaje simple en HTML donde se acceden a través del ingress controller. Existen dos Apaches, los cuales son accedidos con base en el puerto.
Asterisk: Servicio de telefonía utilizando el protocolo SIP en el puerto 5060.

3.2. Arquitectura en AWS



La arquitectura del sistema en AWS es análoga a la expuesta en la sección 3.1. Toda la infraestructura se encuentra circunscrita a un Virtual Private Cloud, la cual aísla los recursos de manera lógica en la nube. Los namespaces se representan por medio de subnets, las cuales limitan un rango de direcciones IP dentro del VPC. En este caso, tenemos dos subnets: una pública y otra privada.
En la subnet pública, tenemos dos instancias EC2: el router y el controlador de Nginx. El router se configura con las reglas de IPTables y con las direcciones IP privadas de las instancias en la red privada. El Nginx también utiliza el IP privado de las instancias de Apache para hacer la redirección de la comunicación. La subnet pública además está relacionada con un recurso de tipo Internet Gateway. Este le permite al VPC tener acceso a Internet.
En la subnet privada, tenemos tres instancias EC2: una para Asterisk y dos para los servidores Apache. En cada instancia, se está corriendo la misma imagen de Docker que se utiliza en la implementación de Kubernetes. Para poder descargar estas imágenes, había que permitir que las instancias en la subnet privada tuvieran acceso a Internet. Para lograr esto, se utilizó un recurso de tipo NAT Gateway, el cual permite que instancias sin IP públicos accedan a la Internet.
Cada una de las instancias tiene asociado su security group para gestionar los puertos e IP permitidos de conexión.

Las guías principales para la implementación con Terraform fueron:

- https://spacelift.io/blog/terraform-aws-vpc
- https://rhuaridh.co.uk/blog/aws-private-subnet.html
- https://developer.hashicorp.com/terraform/tutorials/aws-get-started/aws-build
- https://developer.hashicorp.com/terraform/tutorials/aws-get-started/aws-remote

4. Router

Un router es un dispositivo de capa de red (capa 3) que gestiona el tráfico de paquetes entre múltiples redes. Implementa protocolos mediante los cuales asegura que los paquetes lleguen a su destino de manera correcta, define reglas de control de tráfico para gestionar el flujo de datos, aplica filtros y establece rutas predefinidas para optimizar la comunicación entre dispositivos.

Referencias:

- https://towardsaws.com/how-to-use-iptables-with-linux-ec2-instances-4acb51d2e1ba
- https://linux.die.net/man/8/iptables
- Proyecto ejemplo brindado por el profesor.

Implementación:
Para nuestra proyecto, el router juega un papel crucial al dirigir y gestionar el tráfico entre los servicios del cluster Kubernetes y utiliza reglas de IPtables mediante las cuales define cómo deben dirigirse las conexiones entre los distintos servicios. Implementa reglas de NAT (Network Address Translation) y de forwarding, con las cuales procesa las solicitudes direccionadas a puertos específicos, para que sean debidamente redirigidas a los respectivos servicios, como Asterisk, Apache y el Ingress Controller, asegurando así la correcta comunicación dentro de la infraestructura.

Actores Importantes:

- **IPTables:** Módulo del núcleo de Linux que se encarga de filtrar los paquetes de red mediante un conjunto de reglas con las cuales se determina cómo se va a procesar un paquete que ingrese al router.
- **NAT Table:** Se utiliza para reescribir el origen y/o destino de los paquetes y/o rastrear conexiones, esta tabla contiene las siguientes cadenas:
    - **PREROUTING:** Lista de reglas aplicadas en paquetes inmediatamente después de ser recibidas por una interfaz de red.
    - **OUTPUT:** Lista de reglas aplicadas sobre paquetes inmediatamente después de ser creados por un proceso local.
    - **POSTROUTING:** Lista de reglas aplicadas en paquetes inmediatamente después de salir por una interfaz de red.
- **FILTER Table:** Se utiliza para el procesamiento estándar de paquetes, es la tabla por defecto si no hay ninguna otra especificada, esta tabla contiene las siguientes cadenas:
    - **INPUT:** Lista de reglas aplicadas en paquetes previos a ser entregados a un proceso local.
    - **FORWARD:** Lista de reglas aplicadas sobre paquetes que ingresan por una interfaz de red y salen por otra
    - **OUTPUT:** Lista de reglas aplicadas sobre paquetes inmediatamente después de ser creados por un proceso local.

Configuración del Router:

Búsqueda de Direcciones IP de Servicios
Previo a configurar las reglas de **iptables**, el router utiliza la instrucción **nslookup** para obtener las direcciones

IP de los servicios internos en Kubernetes:

```
APACHE1=$(nslookup $APACHE1URL | awk '/^Address: / { print $2 }')
APACHE2=$(nslookup $APACHE2URL | awk '/^Address: / { print $2 }')
ASTERISK=$(nslookup $ASTERISKURL | awk '/^Address: / { print $2 }')
INGRESS=$(nslookup $INGRESSURL | awk '/^Address: / { print $2 }')
```

En AWS, los IPs de los servicios los asigna Terraform de acuerdo con el orden de las dependencias de los recursos. Primero crea las instancias privadas para obtener su IP privado y después se los pasa a las instancias del Router y de Nginx.

Reglas de IPTables:

Tráfico de entrada y salida
Se admite todo tráfico de entrada y salida mediante la **FILTER Table**, definiendo las reglas sobre las cadenas **INPUT** y **OUTPUT**:

```
iptables -A INPUT -j ACCEPT
iptables -A OUTPUT -j ACCEPT
```

Reglas de Forwarding
Se permite el reenvío de paquetes a los servicios correspondientes dentro del cluster aplicando las reglas sobre la **FILTER Table** en su cadena **FORWARD**:

```
iptables -A FORWARD -i eth0 -o eth0 -p udp --dport 5060 -j ACCEPT
iptables -A FORWARD -i eth0 -o eth0 -p tcp --dport 5060 -j ACCEPT

iptables -A FORWARD -i eth0 -o eth0 -p tcp --dport 80 -j ACCEPT
```

A los Apaches, se les redirecciona el tráfico al puerto 80. A Asterisk, se redirecciona el tráfico al puerto 5060 por los protocolos TCP y UDP.

Reglas de DNAT (Destination NAT)

Filtran los paquetes por su protocolo de red y puerto destino, y configura el **DNAT (Destination NAT)** aplicando las reglas sobre la **NAT Table** en su cadena **PREROUTING:**

```
iptables -t nat -A PREROUTING -i eth0 -p udp --dport 10000:10010 -j DNAT --to-
destination $ASTERISK
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 8080 -j DNAT --to-destination
$APACHE1:80
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 8081 -j DNAT --to-destination
$APACHE2:80
iptables -t nat -A PREROUTING -i eth0 -p udp --dport 5601 -j DNAT --to-destination
$ASTERISK:5060
```

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 5601 -j DNAT --to-destination
$ASTERISK:5060
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j DNAT --to-destination
$INGRESS:80
```

Estas reglas aplican propiamente la redirección de los paquetes. Estas reglas agarran el tráfico que llega al puerto en la opción de `--dport` y lo redireccionan al IP establecido con la opción `--to-destination`. Así, el tráfico que entra por el puerto 8080, se redirige al destino $APACHE1:80, donde $APACHE1 es el IP privado del contenedor del Apache1. Estas reglas trabajan en conjunto con las reglas de Forward.

Reglas de OUTPUT (DNAT para tráfico generado localmente)

Filtra los paquetes generados por procesos locales en el router y modifica su dirección de destino aplicando las reglas sobre la **NAT Table** en su cadena **OUTPUT :**

```
iptables -t nat -A OUTPUT -p tcp --dport 8080 -j DNAT --to-destination $APACHE1:80
iptables -t nat -A OUTPUT -p tcp --dport 8081 -j DNAT --to-destination $APACHE2:80
iptables -t nat -A OUTPUT -p udp --dport 5601 -j DNAT --to-destination
$ASTERISK:5060
iptables -t nat -A OUTPUT -p tcp --dport 5601 -j DNAT --to-destination
$ASTERISK:5060
iptables -t nat -A OUTPUT -p tcp --dport 80 -j DNAT --to-destination $INGRESS:80
```

Reglas de Masquerade (SNAT)

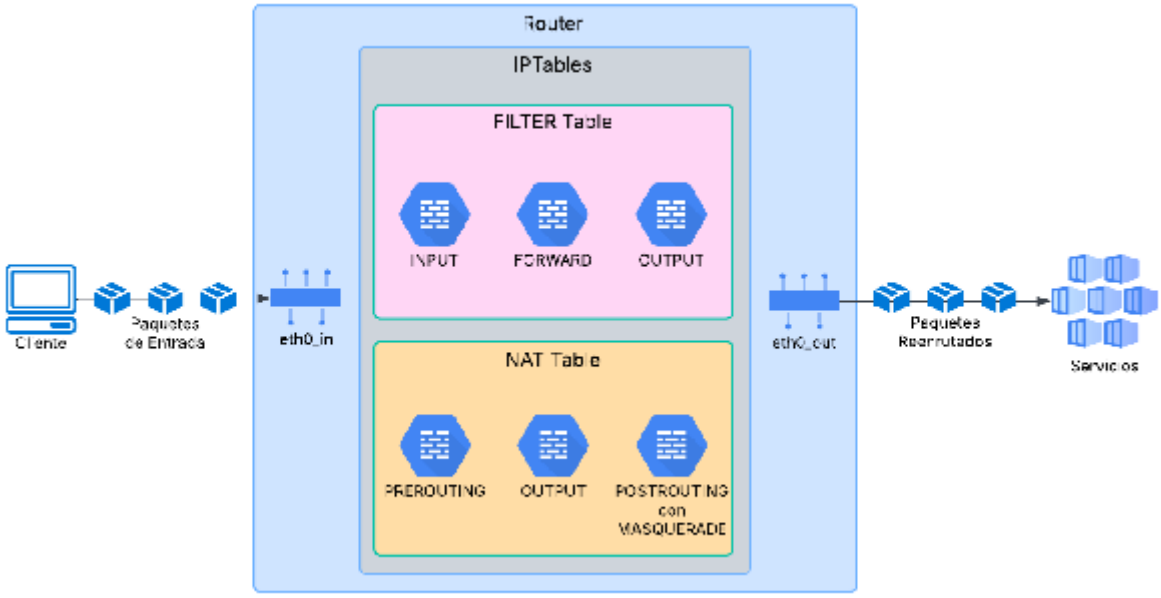
Se configura **MASQUERADE** en la **POSTROUTING** chain dentro de la **NAT Table**, lo que permite que los paquetes salientes utilicen la dirección IP del router como su origen dinámico:

```
iptables -t nat -A POSTROUTING -j MASQUERADE
```

Flujo de paquetes en el router

El siguiente diagrama ilustra cómo los paquetes son enviados por un cliente, ingresa el router en el cual se le

aplican las reglas y filtros previamente definidos y son enrutados a los distintos servicios:



Configuración en Kubernetes:
El router se define en Kubernetes dentro del namespace public, mediante archivos de configuración .yaml y Helm Charts. Dentro del archivo router.yaml se especifica un Deployment y un Service de tipo NodePort para exponer el router como un recurso accesible en la red del clúster.

Deployment:

- Define el nombre y namespace del router (**public**)
- Se especifica la imagen del contenedor (**dandiego235/router-po:latest**)
- Expone los puertos correspondientes a los servicios a los que se redirige el tráfico (**8080, 8081, 5601**).
- Define las variables de entorno con los nombres de los servicios internos (**APACHE1URL, APACHE2URL, ASTERISKURL, INGRESSURL**)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.config.router.name }}
  namespace: public
  labels:
    app: {{ .Values.config.router.name }}
spec:
  replicas: {{ .Values.config.router.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.config.router.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.config.router.name }}
    spec:
      containers:
        - name: {{ .Values.config.router.name }}
```

```
      securityContext:
        privileged: true
      image: {{ .Values.config.router.image }}
      ports:
        - containerPort: 8080
        - containerPort: 8081
        - containerPort: 5601
        - containerPort: 80
      env:
        - name: APACHE1URL
          value: "apache1.private.svc.cluster.local"
        - name: APACHE2URL
          value: "apache2.private.svc.cluster.local"
        - name: ASTERISKURL
          value: "asterisk.private.svc.cluster.local"
        - name: INGRESSURL
          value: "nginx-ingress-controller.public.svc.cluster.local"
```

Service:

- Expone los puertos del router utilizando NodePort, lo que permite el acceso desde fuera del clúster.
- Define las correspondencias entre los puertos internos del router y los puertos del clúster.

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.config.router.name }}
  namespace: public
  labels:
    app: {{ .Values.config.router.name }}
spec:
  type: NodePort
  ports:
    - port: 8080
      protocol: TCP
      name: apache1
      nodePort: 30080
    - port: 8081
      protocol: TCP
      name: apache2
      nodePort: 30081
    - port: 5601
      protocol: UDP
      name: asterisk-udp
      nodePort: 30601
    - port: 5601
      protocol: TCP
      name: asterisk-tcp
      nodePort: 30601
    - port: 80
      protocol: TCP
```

```
      name: ingress
      nodePort: 30082
  selector:
    app: {{ .Values.config.router.name }}
```

Configuración en AWS con Terraform

En AWS, el router está dado por una instancia EC2 pública. Se le asigna un IP público estático por medio del enlace con un Elastic IP. La configuración está dada por el siguiente código de terraform:

```
resource "aws_security_group" "router_sg" {
  vpc_id = aws_vpc.po_vpc.id
  name   = "router_sg"


  tags = {
    Name = "Router Security Group"
  }
}


resource "aws_vpc_security_group_ingress_rule" "allow_ssh_router" {
  security_group_id = aws_security_group.router_sg.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port         = 22
  to_port           = 22
  ip_protocol       = "tcp"
}


resource "aws_vpc_security_group_ingress_rule" "allow_apache" {
  security_group_id = aws_security_group.router_sg.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port         = 8080
  to_port           = 8081
  ip_protocol       = "tcp"
}


resource "aws_vpc_security_group_ingress_rule" "allow_asterisk_tcp" {
  security_group_id = aws_security_group.router_sg.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port         = 5601
  to_port           = 5601
  ip_protocol       = "tcp"
}


resource "aws_vpc_security_group_ingress_rule" "allow_asterisk_udp" {
  security_group_id = aws_security_group.router_sg.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port         = 5601
```

```
  to_port           = 5601
  ip_protocol       = "udp"
}


resource "aws_vpc_security_group_ingress_rule" "allow_http_router" {
  security_group_id = aws_security_group.router_sg.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port         = 80
  to_port           = 80
  ip_protocol       = "tcp"
}


resource "aws_vpc_security_group_egress_rule" "allow_outbound_traffic_router" {
  security_group_id = aws_security_group.router_sg.id
  cidr_ipv4         = "0.0.0.0/0"
  ip_protocol       = "-1"
}


resource "aws_key_pair" "router_key" {
  key_name = "router_key"
  public_key = file("${path.module}/ssh_keys/router_key.pub")
}


resource "aws_instance" "router_instance" {
  ami             = var.aws_ami
  instance_type   = "t2.micro"
  subnet_id       = aws_subnet.public_subnet.id
  vpc_security_group_ids = [aws_security_group.router_sg.id]
  key_name = aws_key_pair.router_key.key_name


  tags = {
    Name = "Router Instance"
  }


  source_dest_check = false


  user_data = templatefile("${path.module}/instance_scripts/router.sh", {
    APACHE1URL_PLACEHOLDER = aws_instance.apache1_instance.private_ip
    APACHE2URL_PLACEHOLDER = aws_instance.apache2_instance.private_ip
    ASTERISKURL_PLACEHOLDER = aws_instance.asterisk_instance.private_ip
    INGRESSURL_PLACEHOLDER = aws_instance.ingress_instance.private_ip
  })
}


# Elastic IP for Router Instance
# https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/eip
```

```
resource "aws_eip" "router_eip" {

}


resource "aws_eip_association" "router_eip_assoc" {
  instance_id   = aws_instance.router_instance.id
  allocation_id = aws_eip.router_eip.id
}
```

En este código, primero se crea el security group, el cual se podría comparar con el servicio de Nodeport. Este establece los puertos permitidos de entrada y de salida. Para la entrada, se permite el acceso al puerto 8080, 8081, 5060 por TCP y UDP, y el 80. Se establece el grupo de direcciones fuente, o CIDR block, en 0.0.0.0 para que permite el acceso desde cualquier dirección de Internet. También, al router se le crea y asocia el recurso de Elastic IP para darle una dirección IP pública estática.

5. Ingress Controller

El Ingress Controller es un componente de capa de aplicación que permite manipular peticiones HTTP y con base en las características de los Uniform Resource Identifiers (URIs), redireccionarlas. El Ingress Controller está implementado con Nginx y el recurso Ingress de Kubernetes. Para el Ingress Controller, se utilizó el Helm Chart de Bitnami, versión 11.6.10. Para la configuración del controller, se usa el recurso de Ingress. Este tiene la siguiente configuración obtenida de la documentación oficial de Kubernetes.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ .Values.config.ingress.name }}
  namespace: private
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: {{ .Values.config.ingress.ingressClassName }}
  rules:
    \- host: {{ .Values.config.ingress.host }}
      http:
        paths:
        {{- range .Values.config.apache.instances }}
        \- path: "/{{ .name }}"
          pathType: Prefix
          backend:
            service:
              name: {{ .name }}
              port:
                number: {{ .port }}
        {{- end }}
```

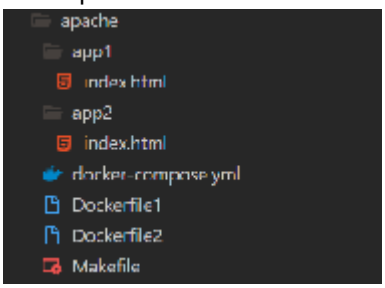
Para configurar los URI's que acepta el controller, el recurso itera por los valores de la configuración del namespace privado en el archivo values.yaml. De este, obtiene los nombres de los servidores Apache y los utiliza como el path. También, obtiene los puertos por los que los servidores reciben las peticiones. Es importante recalcar que el recurso Ingress debe estar en el namespace privado para que se pueda hacer la redirección. Esto es porque a pesar de que el ingress controller funciona a nivel de cluster, el ingress resource solo tiene alcance dentro del namespace donde es creado, es decir, no puede buscar el recurso a través de los diferentes namespaces.

Configuración en AWS
En AWS, el Ingress Controller es una instancia EC2 que está corriendo Nginx. Para configurar el servicio de Nginx, se crea un archivo de configuración de tipo `reverse_proxy.conf`. En este, se establece el enlace de las direcciones IP de los servidores de Apache con las rutas del URL. La configuración de Nginx como reverse proxy se basa principalmente en la directiva `proxy_pass`, la cual permite establecer la dirección de enrutamiento. Se usan variables especiales de nginx, como $host y $remote_addr para pasarle información relevante al servidor.
Referencias:

- Sección de "Setting Up a Simple Proxy Server": https://nginx.org/en/docs/beginners_guide.html
- https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/
- https://www.digitalocean.com/community/tutorials/how-to-configure-nginx-as-a-reverse-proxy-on-ubuntu-22-04

6. Servidores Apache

Apache es uno de los servidores web más utilizados en el mundo. Desarrollado y mantenido por la Apache Software Foundation, es un software de código abierto que permite la entrega eficiente de contenido web. Su flexibilidad, seguridad y compatibilidad con diversos sistemas operativos lo han convertido en una opción popular. Apache soporta una gran cantidad de módulos que permiten extender sus capacidades y además tiene muchas posibles integraciones con otras tecnologías como PHP, MySQL entre otras. Este es el folder correspondiente a todo lo relacionado con la creación de Apaches (Domantas, 2025).



Primeramente mediante la herramienta Helm se permite generar los archivos YAML de Kubernetes para que sean desplegados en el clúster. Eeste código, que se encuentra en la dirección:**\charts\private\values.yaml**, permite definir la creación de los apaches. En la línea que dice Apache habilita dicho servicio donde crea dos instancias, apache1 y apache2, cada una con su propia imagen y puerto.

```
config:
  apache:
    enabled: true
    replicas: 1
```

```
    instances:
      - name: apache1
        image: dandiego235/apache-po:1.0
        port: 80
      - name: apache2
        image: dandiego235/apache-po:2.0
        port: 80
```

Los directorios app1 y app2 contienen un archivo HTML básico, el cual es servido por Apache de manera sencilla. El objetivo es verificar que, si se puede acceder al archivo HTML a través del navegador, significa que el servicio de Apache se ha ejecutado correctamente. La siguiente imagen es un ejemplo del documento HTML que apache hostea:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Apache1</title>
  </head>
  <body>
    <header>
      <h1>Hola Apache 1</h1>
    </header>
    <main>
      <p>Este es el primer servidor de Apache.</p>
    </main>
  </body>
</html>
```

El siguiente archivo llamado docker-compose.yml se encarga de definir dos servicios apache donde cada uno tiene su propio Dockerfile. Permite construir cada imagen de apache con una versión específica y levanta ambos mediante el comando “docker-compose up --build”. El Dockerfile especifica la imagen que se va a utilizar, en estos casos Dockerfile1 y Dockerfile2. Finalmente en tags lo que se asigna es una etiqueta en la imagen construida.

```
services:
  apache1:
    build:
      context: .
      dockerfile: Dockerfile1
      tags:
        - dandiego235/apache-po:1.0

  apache2:
    build:
      context: .
      dockerfile: Dockerfile2
      tags:
        - dandiego235/apache-po:2.0
```

El archivo llamado Dockerfile1, por ejemplo, se encarga de tomar un contenedor de Apache ya creado previamente y utilizarlo. Este es el código que hace básicamente la construcción de las imágenes. La primera línea siendo from httpd:2.4 usa la imagen oficial de Apache HTTP Server en la versión 2.4 como base para el contenedor. La segunda línea simplemente copia el contenido en la dirección registrada. Esto significa que, gracias a la imagen, la copia de Apache va a quedar guardada en la carpeta app1. El contenedor va a empezar a servir este archivo HTML por defecto.

```
FROM httpd:2.4
COPY ./app1/ /usr/local/apache2/htdocs/
```

Ahora el último archivo llamado Makefile se encarga de realizar un push en Docker Hub sobre las imágenes. La primera línea se encarga de mostrar un mensaje para el usuario describiendo que se están subiendo las imágenes a Docker Hub. Las siguientes líneas se encargan de subir la imagen al registro de Docker Hub donde se empuja la imagen con el nombre dandiego235/apache-po y la etiqueta 1.$(version) donde version es una variable

```
push:
    @echo "Pushing to DockerHub"
    docker push dandiego235/apache-po:1.$(version)
    docker push dandiego235/apache-po:2.$(version)
```

En conclusión, el flujo general para el despliegue de los servicios de apache comienza con la importación de las imágenes de Docker. Cuando ya se importaron las imágenes y se copiaron en la máquina con base en los directorios de app, el docker-compose.yml se encarga de definir los servicios de apache1 y apache2 con tags basados en las imágenes previamente creadas. Finalmente gracias al archivo Makefile se automatiza la subida de las imágenes en Docker Hub.
En AWS, se ejecutan los Apaches en instancias EC2 que corren la imagen de Docker. Se declara un security group compartido para ambos servidores, ya que comparten las mismas reglas. Se establece acceso al puerto 22 desde todos los IPs para permitir el acceso desde SSH. El acceso al puerto 80 se limita a los IPs de la subnet pública, los cuales tienen la forma de 10.0.1.x. Así, se restringe el acceso a solo las instancias en la subnet pública internas al VPC. Se establece una regla de acceso no restringido para el tráfico de salida.

7. Asterisk

Asterisk es una plataforma de comunicación que permite la implementación de sistemas de telefonía basados en software, como centrales telefónicas, pasarelas VoIP y servidores de conferencia. Su flexibilidad lo hace ideal para manejar comunicaciones de voz sobre IP utilizando protocolos como SIP (Session Initiation Protocol). En este proyecto, Asterisk se ejecuta dentro de un contenedor Docker, permitiendo un despliegue reproducible y eficiente. Mediante archivos de configuración específicos (pjsip.conf, extensions.conf, rtp.conf) y un script (script.sh), se definen las reglas de comunicación, transporte y autenticación para gestionar llamadas entre extensiones.

Primeramente se van describir los componentes de configuración donde el primero se llama extensions.conf encargado de decidir que hacer al recibir una llamada. Se le conoce como “dialplan” al hecho de realizar acciones al recibir una llamada. En este caso, el sistema responde llamadas al número 100 y define dos extensiones: 6001 y 6002. Estas permiten la comunicación entre usuarios. Básicamente, este define la lógica de las llamadas dentro de Asterisk

```
[from-internal]
exten = 100,1,Answer()
same = n,Wait(1)
same = n,Playback(hello-world)
same = n,Hangup()
exten => 6001,1,Dial(PJSIP/6001,20)
exten => 6002,1,Dial(PJSIP/6002,20)
```

El segundo archivo importante se llama pjsip.conf, el cual tiene como funcionamiento principal definir los transportes, plantillas y endpoints para las configuraciones de las extensiones SIP. La primera parte sobre “transport-udp” define como Asterisk se comunica con otros dispositivos SIP.

- type=transport → Define que esta sección es un transporte.
- protocol=udp → Protocolo de transporte es UDP
- bind=0.0.0.0 → Asterisk escuchará en todas las interfaces de red.
- local_net=127.0.0.1/8 → Define la red local
- external_media_address=${EXTERNAL_IP} → Direcciones IP externa para medios RTP.
- external_signaling_address=${EXTERNAL_IP} → Dirección IP externa para la señalización SIP.
- external_signaling_port=${EXTERNAL_PORT} → Puerto para la señalización SIP externa.

Para la función llamada “[transport-tcp]” es el mismo tipo de conexión de la anterior, pero cambia que usa TCP en vez de UDP, que puede permitir conexiones más confiables.

Las endpoint templates tienen las siguientes definiciones:

- type=endpoint → Define que es una plantilla de endpoints.
- context=from-internal → Especifica el contexto de llamadas para estos usuarios.
- direct_media=yes → Permite que el tráfico RTP fluya directamente entre los dispositivos sin pasar por Asterisk.
- force_rport=yes → Uso del puerto RTP especificado por el cliente.
- rewrite_contact=yes → Modifica la dirección de contacto para asegurar conectividad.
- rtp_symmetric=yes → Indica que los paquetes RTP deben enviarse al mismo puerto del que fueron recibidos (importante para NAT).

Además se incluye una plantilla de autenticación, la cual simplemente especifica el tipo de autenticación, en este caso usuario y contraseña.

Finalmente, se definen ambas extensiones SIP utilizando las plantillas anteriores donde se le define un usuario y una contraseña. Además mediante el aors se enlaza el endpoint con el registro AOR.

```
[transport-udp]
type=transport
```

```
protocol=udp
bind=0.0.0.0
local_net=127.0.0.1/8
external_media_address=${EXTERNAL_IP}
external_signaling_address=${EXTERNAL_IP}
external_signaling_port=30601

[transport-tcp]
type=transport
protocol=tcp
bind=0.0.0.0
local_net=127.0.0.1/8
external_media_address=${EXTERNAL_IP}
external_signaling_address=${EXTERNAL_IP}
external_signaling_port=30601

;===============ENDPOINT TEMPLATES
[endpoint-basic](!)
type=endpoint
context=from-internal
disallow=all
allow=ulaw
direct_media=yes
force_rport=yes
rewrite_contact=yes
rtp_symmetric=yes

[auth-userpass](!)
type=auth
auth_type=userpass

[aor-single-reg](!)
type=aor
max_contacts=1
qualify_frequency=30

;===============EXTENSION 6001

[6001](endpoint-basic)
auth=auth6001
aors=6001

[auth6001](auth-userpass)
password=unsecurepassword
username=6001

[6001](aor-single-reg)

;===============EXTENSION 6002

[6002](endpoint-basic)
auth=auth6002
aors=6002
```

```
[auth6002](auth-userpass)
password=unsecurepassword
username=6002

[6002](aor-single-reg)
```

El archivo llamado rtp.conf en Asterisk se encarga de definir el rango de puertos RTP (Real-time Transport Protocol) que Asterisk usará para la transmisión de medios en llamadas de voz y video. En este caso simplemente se define que el puerto inicial va a ser el 10000 y el final 10010. Esto significa que se pueden realizar 5 llamadas simultáneas debido a que cada llamada utiliza 2 puertos.

```
[general]
rtpstart=10000
rtpend=10010
```

El archivo llamado script.sh permite ejecutar Asterisk donde primero muestra los valores del IP. Después, los reemplaza el EXTERNAL_IP y EXTERNAL_PORT de las variables de entorno en el archivo `pjsip.conf`. Finalmente, se ejecuta Asterisk en `verbose mode`.

```
#!/bin/bash


echo "EXTERNAL_IP is set to: $EXTERNAL_IP"
echo "EXTERNAL_PORT is set to: $EXTERNAL_PORT"


# Replace environment variables in the Asterisk configuration file
sed -i "s|\${EXTERNAL_IP}|$EXTERNAL_IP|g" /etc/asterisk/pjsip.conf
sed -i "s|\${EXTERNAL_PORT}|$EXTERNAL_PORT|g" /etc/asterisk/pjsip.conf


# Start Asterisk in verbose mode
asterisk -cvvvvv
```

El archivo llamado Dockerfile es el contenedor como tal donde se define la imagen de Asterisk que incluye toda la configuración necesaria. Primero, toma una imagen base de Ubuntu, donde después ejecuta un comando para realizar todas las actualizaciones y correr varias dependencias importantes para el funcionamiento correcto de Asterisk. Después de instalar todas las dependencias, lo que se hace es instalar el Asterisk en la máquina virtual en la carpeta específica donde eventualmente lo compila. Se instalan todos los archivos de configuración por defecto. Finalmente, se copia un script para la imagen, siendo este el comando inicial.

```
FROM ubuntu:latest

RUN apt update && apt upgrade -y
```

```
RUN apt install -y \
  build-essential \
  gcc \
  g++ \
  make \
  nano \
  libjansson-dev \
  libsqlite3-dev \
  libxml2-dev \
  libxslt1-dev \
  libncurses5-dev \
  libssl-dev \
  uuid-dev \
  wget \
  curl \
  unixodbc \
  unixodbc-dev \
  libspeex-dev \
  libspeexdsp-dev \
  libsamplerate-dev \
  libcurl4-openssl-dev \
  libvorbis-dev \
  libogg-dev \
  libsrtp2-dev \
  libical-dev \
  libiksemel-dev \
  libneon27-dev \
  libgmime-3.0-dev \
  libunbound-dev \
  libedit-dev

WORKDIR /usr/local/src

RUN wget https://downloads.asterisk.org/pub/telephony/asterisk/asterisk-
22.2.0.tar.gz

RUN tar -zxvf asterisk-22.2.0.tar.gz

WORKDIR /usr/local/src/asterisk-22.2.0

RUN ./configure
RUN make && make install
RUN make samples
RUN make config

WORKDIR /etc/asterisk
RUN mv extensions.conf extensions.conf.bak
RUN mv pjsip.conf pjsip.conf.bak
RUN mv rtp.conf rtp.conf.bak
COPY ./app/extensions.conf .
COPY ./app/pjsip.conf .
COPY ./app/rtp.conf .

WORKDIR /app
```

```
COPY ./app/script.sh .

CMD ["sh", "script.sh"]
```

Finalmente, el último archivo de Asterisk es docker-compose.yml, el cual se encarga de llamar al archivo Dockerfile y construir la imagen. Primeramente, se le define un tag, Asterisk, donde en este caso su build se puede construir en lugar de descargarse desde un registro de imágenes. Este se encuentra por la ruta que se da en `context`, en este caso en el mismo folder, y se llama Dockerfile. En la parte de tags se definen nombres y versiones de la imagen, en este caso toma la de asterisk y la que está en el usuario de dandiego235 de Docker, el cual es el repositorio de imágenes en Docker Hub. Finalmente, expone los puertos, en este caso el 5060 y muestra el rango de puertos entre 10000 y 10010 para las llamadas.

```
services:
  asterisk:
    build:
      context: .
      dockerfile: Dockerfile
      tags:
        - asterisk-po:latest
        - asterisk-po:2.6
        - dandiego235/asterisk-po:latest
        - dandiego235/asterisk-po:2.6
    container_name: asterisk
    # volumes:
    #   - .:/app
    ports:
      - "5060:5060/udp"
      - "10000-10010:10000-10010/udp" # RTP media ports
```

En conclusión el sistema basado en Asterisk y su configuración se conforma de la siguiente manera. Primeramente mediante el Makefile se pueden automatizar las tareas que son la compilación, construcción de la imagen de Docker y la ejecución de servicios. Mediante el Dockerfile se define la imagen que contendrá Asterisk e instala los paquetes necesarios, copiando los archivos `extensions.conf`, `pjsip.conf` y `rtp.conf` dentro del contenedor. El archivo llamado docker-compose facilita la ejecución de Asterisk y además mapear los puertos necesarios para la comunicación. Estos archivos permiten una implementación eficiente y escalable del sistema, asegurando su correcto funcionamiento en distintos entornos de red.

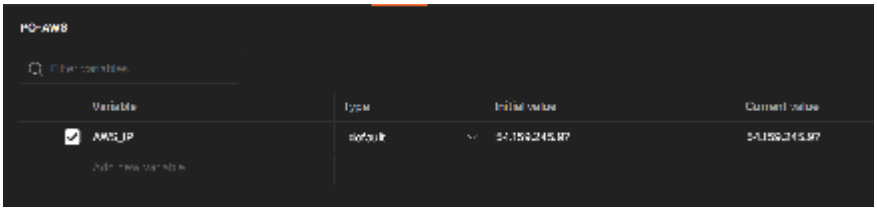
En AWS, Asterisk corre en su propia instancia EC2 en la subnet privada. Se le asigna su propio security group. Este expone el puerto 5060 por TCP y UDP. También, se abre el puerto 22 para acceso por SSH.
Referencias:

- https://docs.asterisk.org/Getting-Started/Installing-Asterisk/Installing-Asterisk-From-Source/What-to-Download/
- *https://docs.asterisk.org/Configuration/Channel-Drivers/SIP/Configuring-res_pjsip/res_pjsip-Configuration-Examples/;a-sip-trunk-to-your-service-provider-including-outbound-registration*
- *https://docs.asterisk.org/Getting-Started/Hello-World/*

- *https://docs.asterisk.org/Configuration/Channel-Drivers/SIP/Configuring-res_pjsip/Configuring-res_pjsip-to-work-through-NAT/#clients-supporting-icestunturn*
- *Domantas, G. (2025). What is Apache web server and how does it work. Hostinger Tutorials. https://www.hostinger.com/tutorials/what-is-apache*

8. Pruebas {#8.-pruebas}

Para las pruebas de AWS, se creó una variable de entorno en Postman que se llamaba AWS_IP con el IP público del router:



Router

Se utilizó la herramienta **Postman** mediante la cual se hicieron requests **HTTP** para probar el correcto enrutamiento de los paquetes enviados, acorde a las reglas definidas en el router:

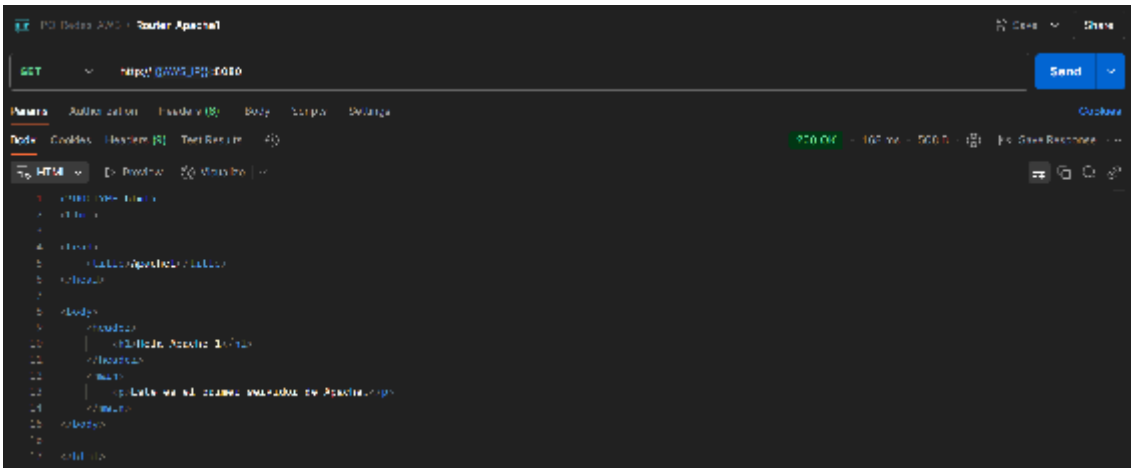
Prueba de redireccionamiento a Apache1

Kubernetes

Se envió un petición **GET** a la dirección localhost, en el puerto **30080** configurado en el router para **Apache1**, mostrando el correcto enrutamiento de los paquetes enviados:

AWS

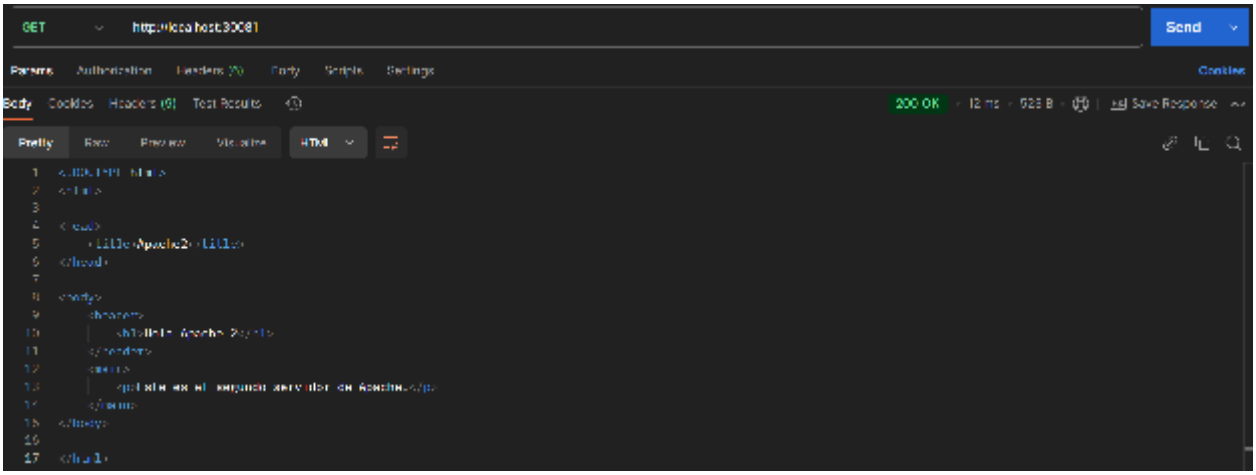
Se envió un petición **GET** a la dirección `54.159.245.97`, en el puerto **8080** configurado en el router para **Apache1**, mostrando el correcto enrutamiento de los paquetes enviados:



Prueba de redireccionamiento a Apache2

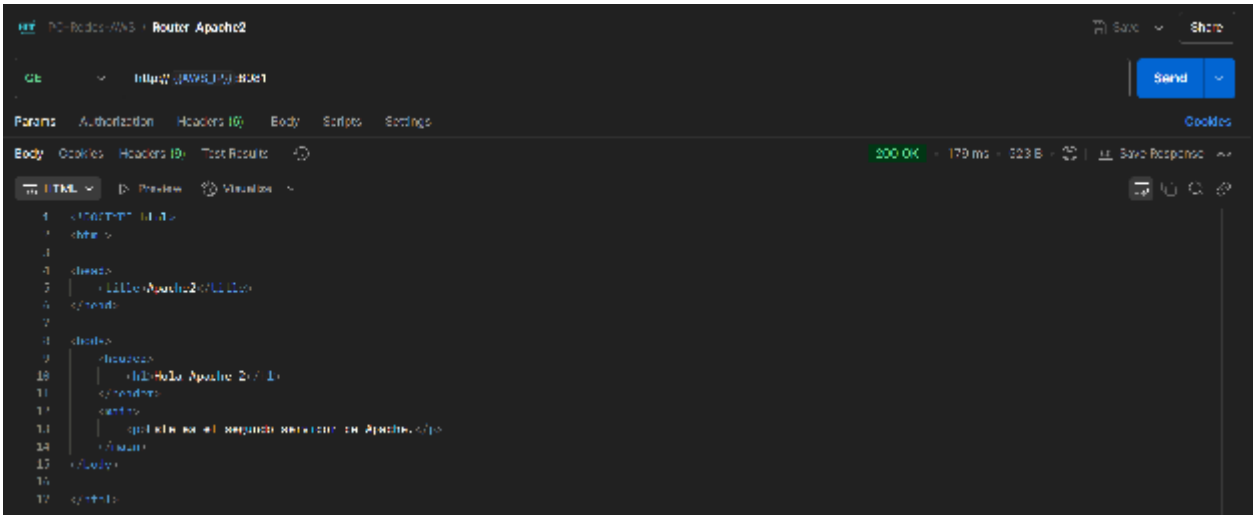
Kubernetes

Se envió un petición **GET** a la dirección localhost, en el puerto **30081** configurado en el router para **Apache2**, mostrando el correcto enrutamiento de los paquetes enviados:



AWS

Se envió un petición **GET** a la dirección `54.159.245.97`, en el puerto **8081** configurado en el router para **Apache2**, mostrando el correcto enrutamiento de los paquetes enviados:



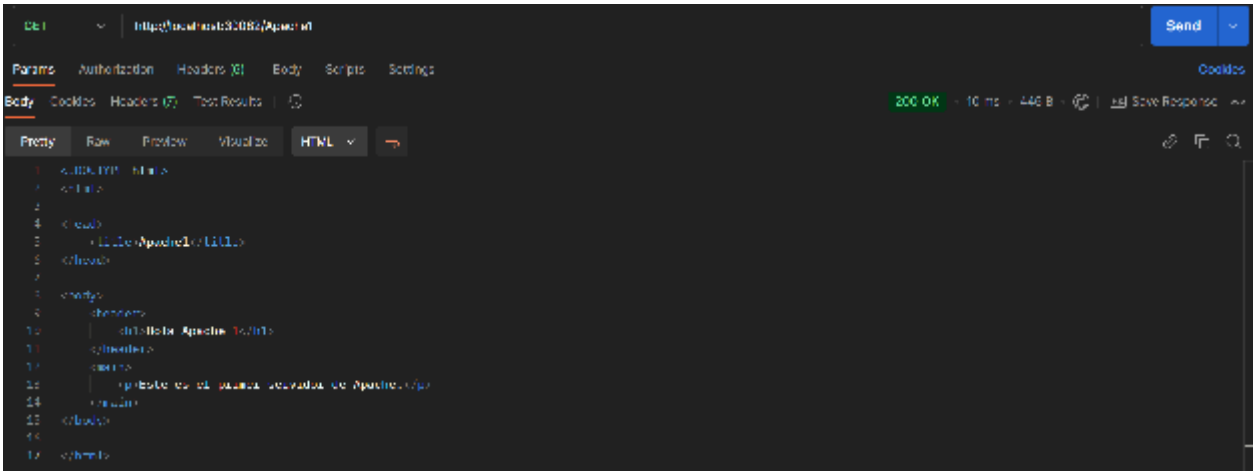
Ingress Controller

Se utilizó la herramienta **Postman** para enviar solicitudes **HTTP** y probar la accesibilidad del servicio a través del Ingress Controller y mediante la herramienta **Lens** se monitorea los logs del Ingress Controller para así verificar el correcto enrutamiento:

Verificación de Enrutamiento a Apache1

Kubernetes

Se realizó una solicitud HTTP a **localhost:30082**, el cual corresponde al puerto expuesto por el router. Este recibe la solicitud y se la redirige al Ingress Controller, que procesa el enrutamiento y dirige el tráfico al servicio **Apache 1**:

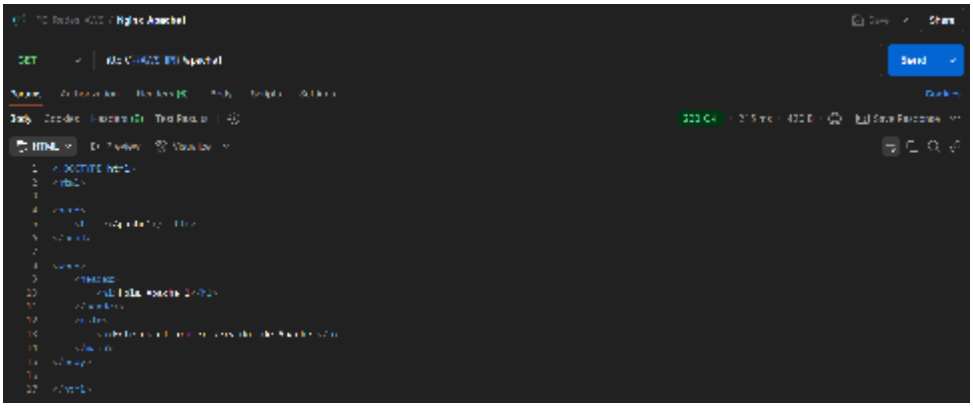


Tras revisar los logs del Ingress Controller se confirma que la solicitud fue redirigida correctamente al servicio **apache1** en el namespace **private**:

```
10.1.0.152 - - [13/Mar/2025:03:49:29 +0000] "GET /Apache1 HTTP/1.1" 200 226 "-"
"PostmanRuntime/7.42.0" 209 0.007 [private-apache1-80] [] 10.1.0.149:80 226 0.007
200 c240b8e753fdcae47d72e4ba1a5d34af
```

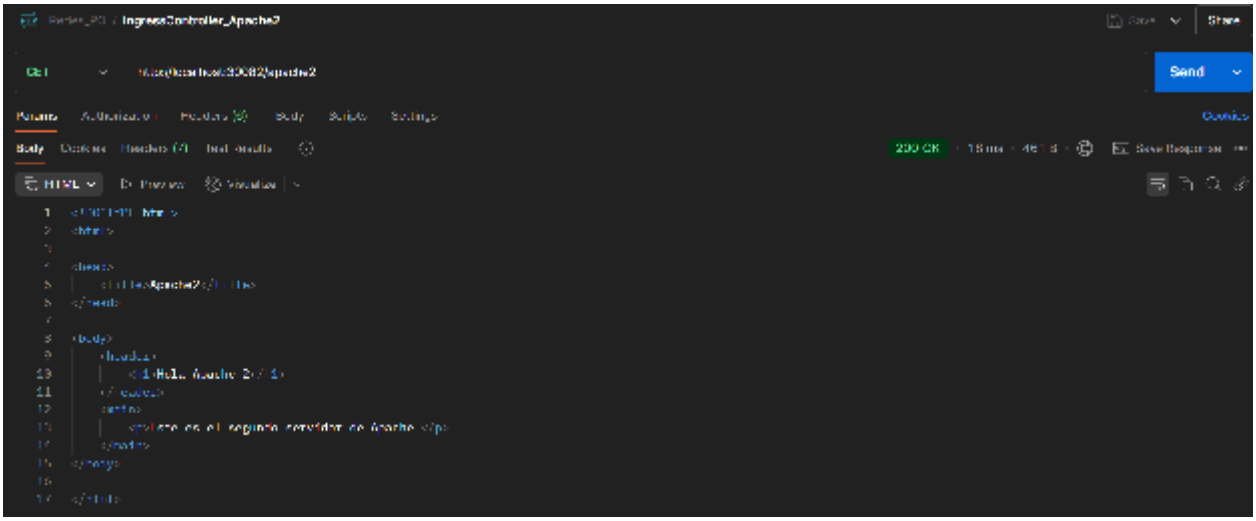
AWS

Se realizó una solicitud HTTP a `54.159.245.97`, el cual corresponde al puerto 80 expuesto por el router. Este recibe la solicitud y se la redirige al Ingress Controller, que procesa el enrutamiento y dirige el tráfico al servicio **Apache 1**:



Kubernetes

Se realizó una solicitud HTTP a **localhost:30082**, el cual corresponde al puerto expuesto por el router. Este recibe la solicitud y se la redirige al Ingress Controller, que procesa el enrutamiento y dirige el tráfico al servicio **Apache 2**:

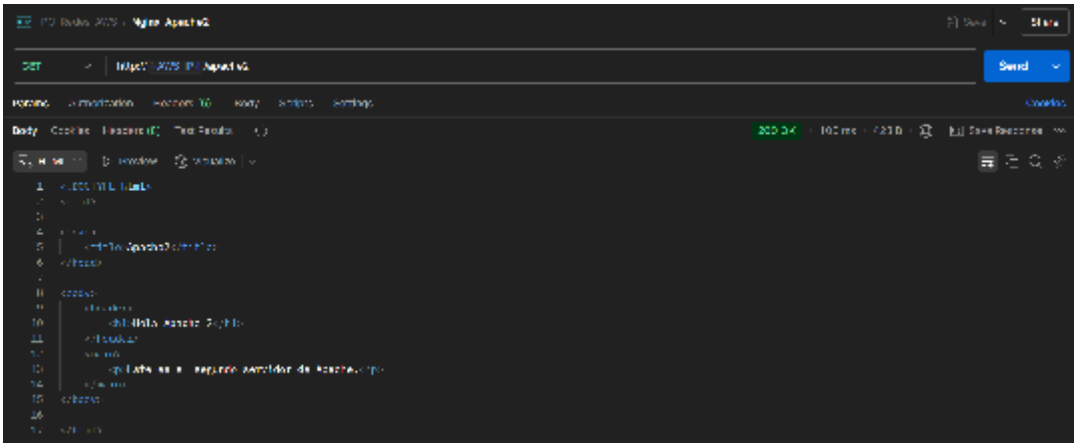


Tras revisar los logs del Ingress Controller se confirma que la solicitud fue redirigida correctamente al servicio **apache2** en el namespace **private**:

```
10.1.0.152 - - [13/Mar/2025:03:49:29 +0000] "GET /apache1 HTTP/1.1" 200 241 "-"
"PostmanRuntime/7.43.2" 209 0.002 [private-apache2-80] [] 10.1.0.162:80 241 0.003
200 2f8fc6462a9a57df7058c0a7496adbd5
```

AWS {#aws-3}

Se realizó una solicitud HTTP a 54.159.245.97, el cual corresponde al puerto 80 expuesto por el router. Este recibe la solicitud y se la redirige al Ingress Controller, que procesa el enrutamiento y dirige el tráfico al servicio **Apache 2**:



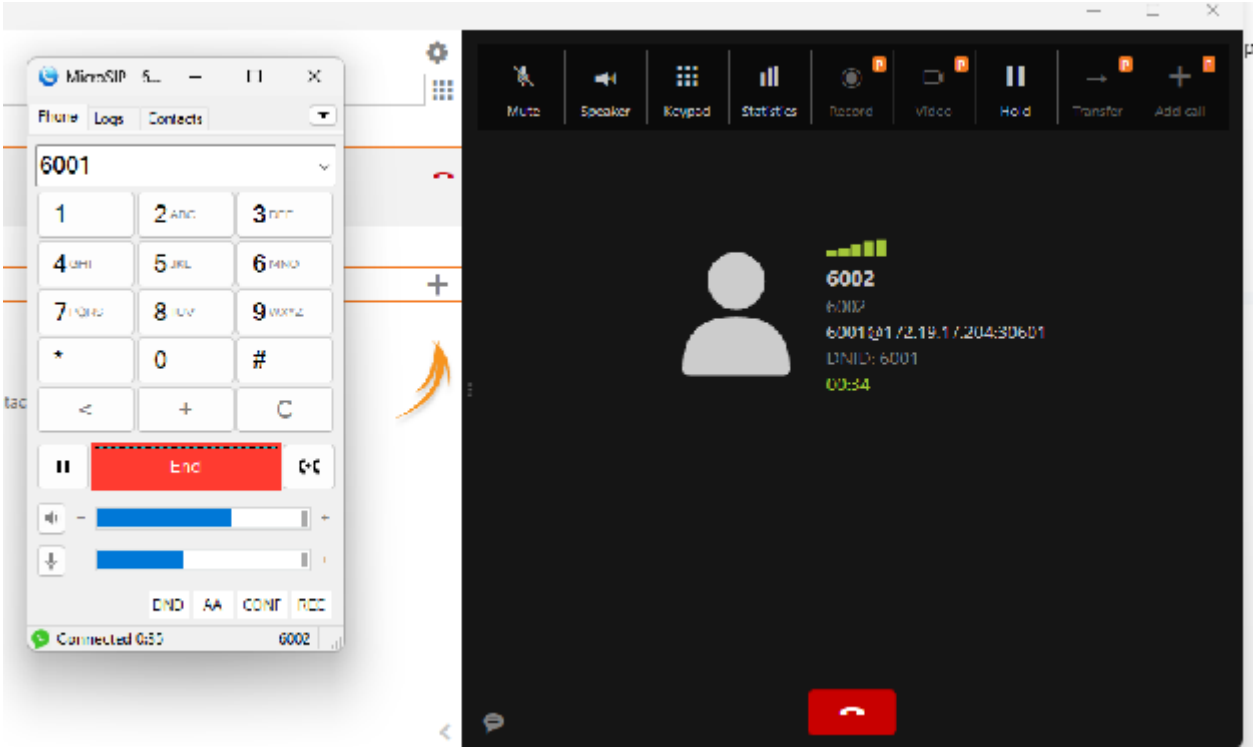
Asterisk

```
Para probar el Asterisk, realizamos dos pruebas. La primera es por medio de la
configuración de Asterisk.
```

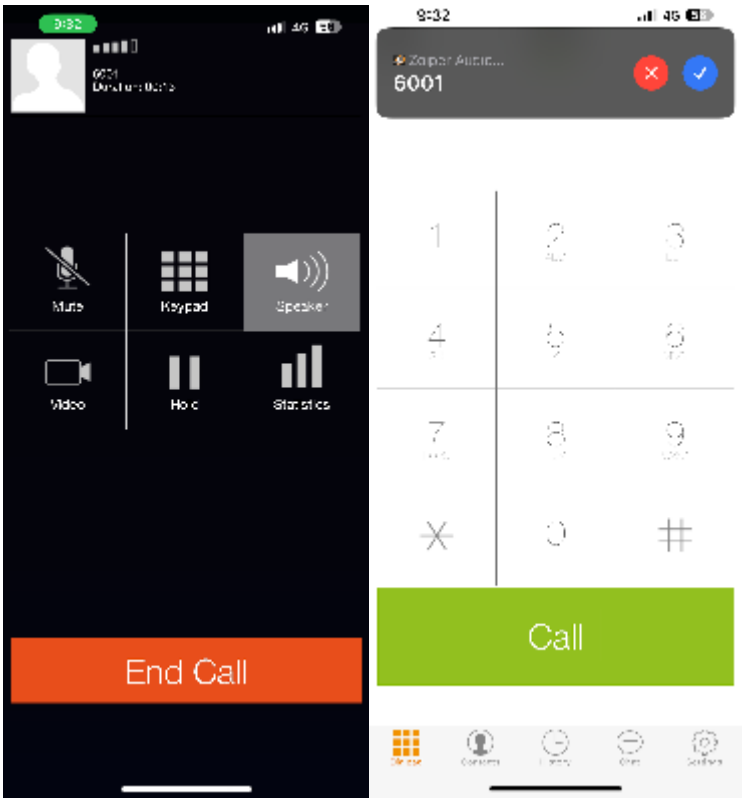
```
[from-internal]
exten = 100,1,Answer()
same = n,Wait(1)
same = n,Playback(hello-world)
same = n,Hangup()
exten => 6001,1,Dial(PJSIP/6001,20)
exten => 6002,1,Dial(PJSIP/6002,20)
```

En el archivo, extensions.conf, habilitamos tres extensiones. La primera extensión es la del número 100. Cuando uno marca este número, el Asterisk reproduce un sonido que dice “Hello World!” y posteriormente cuelga. Esta extensión nos permite probar si el teléfono SIP se puede conectar correctamente con Asterisk.

Las otras dos extensiones se utilizan para las llamadas entre los teléfonos. Para esta prueba, simplemente uno de los teléfonos llama al otro y se verifica si se pueden escuchar entre sí. En el ambiente de Kubernetes, las pruebas se realizaron en la misma computadora corriendo Windows, con las aplicaciones de MicroSIP y Zoiper.



En el ambiente de AWS, se realizó con teléfonos celulares, con la aplicación de Zoiper para el sistema operativo iOS y Android.



Adicionalmente, se probó la conexión con la herramienta netcat para validar que estuviera recibiendo los protocolos TCP y UDP:

En Kubernetes:

```
dandi@Legion:/mnt/c/Users/dandi/Redes/2025-01-IC7602$ nc -zv 172.22.160.1 30601
Connection to 172.22.160.1 30601 port [tcp/*] succeeded!
dandi@Legion:/mnt/c/Users/dandi/Redes/2025-01-IC7602$ nc -zvu 172.22.160.1 30601
Connection to 172.22.160.1 30601 port [udp/*] succeeded!
```

En AWS:

```
dandi@Legion:/mnt/c/Users/dandi/Redes/2025-01-IC7602$ nc -zv 54.159.245.97 5601
Connection to 54.159.245.97 5601 port [tcp/*] succeeded!
dandi@Legion:/mnt/c/Users/dandi/Redes/2025-01-IC7602$ nc -zvu 54.159.245.97 5601
Connection to 54.159.245.97 5601 port [udp/*] succeeded!
```

9. Recomendaciones {#9.-recomendaciones}

1. La arquitectura basada en Kubernetes y contenedores demostró ser una solución robusta para la gestión de servicios en red. La segmentación en namespaces públicos y privados permitió una mejor organización y seguridad, garantizando que los servicios internos permanezcan protegidos mientras que los externos sean accesibles de manera controlada.
2. Buscar una mejor optimización a la hora del despliegue de imágenes, utilizando solo lo necesario. Esto es importante para reducir el tamaño de las imágenes y del contenedor, lo cual ayuda a que el sistema se despliegue más rápidamente. Además sería importante establecer ciertos límites en los requisitos de Kubernetes para evitar que los pods consuman más recursos de los necesarios.
3. Para probar Asterisk, se pueden descargar dos teléfonos SIP y comunicarlos entre sí. Se recomienda utilizar Zoiper y MicroSIP. Otros teléfonos, como 3CX, no resultaron tan intuitivos de usar. MicroSIP resultó funcionar mejor que Zoiper. Zoiper dio más problemas.
4. Es importante configurar o identificar el IP Address del WSL en Asterisk para que los datos que envía Asterisk puedan salir de la máquina virtual y llegar a los teléfonos.

5. El uso de herramientas como Lens para poder acceder a los logs de los Pods y Services ayuda a tener un control sobre las acciones hechas por estos servicios. Es especialmente útil la herramienta de abrir una terminal interactiva dentro del contenedor, ya que de lo contrario, con Kubectl o Docker es un poco más tedioso. Tener la terminal interactiva permite monitorear efectivamente qué está ocurriendo dentro del contenedor, como vigilar los logs de Asterisk y configurarlo directamente, comprobar que las reglas de iptables se configuraron correctamente con el DNS o IP, o que los archivos se copian correctamente.
6. El uso de herramientas como Postman facilita mucho el realizar pruebas sobre los puertos expuestos por el NodePort para así poder comprobar su comportamiento y ver si se obtienen los resultados esperados.
7. Se puede optimizar las reglas definidas en el IPTable del router, para asegurar que solo se permiten las conexiones deseadas y evitar tráfico innecesario que pueda generar vulnerabilidades en la red. Por lo tanto, se recomienda primero estudiar la herramienta de IPTables y entender cómo funcionan las cadenas de envío y la secuencia de enrutamiento. Al inicio, nos pasó muchas veces que no se podía acceder al servicio por haber configurado mal las reglas.
8. El rango de puertos RTP que utiliza Asterisk puede ampliarse para así poder garantizar una escalabilidad en el número de llamadas que se pueden realizar simultáneamente. No obstante, para un caso de uso con pocas conexiones a la vez, se puede reducir el rango de puertos abiertos para que la carga en el contenedor y en la computadora sea menor. Por ejemplo, nos ocurrió que al intentar abrir el rango de puertos de 10000 a 20000 por medio del port-forwarding, ya que este es el rango usual de puertos que utiliza Asterisk para el envío de paquetes RTP, la computadora se quedó congelada y el contenedor nunca logró empezar a correr bien.
9. Se recomienda utilizar Terraform para configurar los recursos en el Cloud. La alternativa de crear los objetos de infraestructura en la nube desde el portal de AWS es menos eficiente y puede resultar engorroso. Se recomienda revisar la documentación antes de usar Terraform y de los recursos que se utilizarán para configurarlos solo con los argumentos y atributos realmente requeridos. Esto ayuda a que se entienda mejor la solución de infraestructura.
10. Entender correctamente el funcionamiento de los objetos de Kubernetes, esto en el caso del ingress controller, ya que hubiera facilitado su implementación.
11. Entender claramente las dependencias de los recursos en la nube para comprender la secuencia en que se deberían crear los objetos. Por ejemplo, a nosotros nos pasó que las instancias no estaban descargando las imágenes de Docker Hub porque no tenían acceso a Internet a pesar de que también se había creado un NAT Gateway en el ambiente. Esto fue porque primero se creaban las imágenes y después el NAT Gateway, lo cual causaba que cuando las instancias EC2 intentaban jalar las imágenes, no tenían acceso hacia afuera por Internet por lo que fallaba la ejecución. Tuvimos que agregar la dependencia `depends_on` en las instancias privadas EC2 con el NAT Gateway para resolver este problema. Así, primero se creaba completamente el NAT Gateway y después se levantaban las instancias, lo cual garantiza que las imágenes tuvieran acceso a Internet para descargar las imágenes.
12. Se vuelve muy útil tener una forma de acceder a las instancias en AWS, por lo que se recomienda configurar acceso por llaves SSH. Nos pasó muchas veces que el acceso al Router o a las instancias públicas fallaba porque no servía el EC2 Instant Connect. Además, esta opción no estaba disponible para las instancias privadas, lo cual implicaba que esas eran completamente inaccesibles. Fue muy difícil comprobar que las imágenes de Docker estuvieran corriendo sin poder acceder a las instancias. Por lo tanto, fue muy útil configurar el acceso por SSH para el router, y las instancias privadas. Para acceder a las instancias privadas, primero había que copiar la llave a una instancia pública, conectarse por ssh a la instancia pública y después conectarse por ssh a la privada. Este proceso al inicio no es tan intuitivo,

pero fue extremadamente útil para depurar la condición de las imágenes, porque sí tuvimos muchos problemas para lograr que las instancias privadas corrieran las aplicaciones correctamente.

13. Si alguna imagen no funciona correctamente porque el script da error por cambios de línea, eso debe ser por la diferencia en el manejo de cambios de línea entre Linux y Windows. Para arreglar el problema, se ejecuta el comando `dos2unix script.sh`.

10. Conclusiones

1. La implementación de Helm, Docker y los scripts de automatización redujo significativamente la carga manual en la configuración y despliegue del proyecto. Además de reducir el tiempo y la complejidad reduce la cantidad de errores humanos optimizando los tiempos de instalación.
2. La división en namespaces public y private facilita la organización y seguridad de los servicios, asegurando así que los componentes internos permanezcan debidamente protegidos y los externos puedan ser expuestos sin ningún riesgo. Esto es más relevante en el ambiente de AWS, donde la infraestructura está expuesta a todo el tráfico de Internet.
3. Una debida definición de reglas NAT y filtrado de paquetes en el router garantiza una correcta comunicación entre los servicios y la red externa. En el ambiente de Kubernetes, esto es relevante porque hay que tomar en cuenta la encapsulación de las redes virtuales que implica usar WSL y Docker. Es importante reconocer estas relaciones para identificar la dirección IP correcta para usar. Esto puede determinar la diferencia si se usa el `localhost` o el IP de WSL. Esto fue necesario para el correcto funcionamiento de Asterisk en el ambiente local.
4. Utilizar estructuras como nuestra implementación de un router junto con un ingress controller ayuda a tener un buen control sobre el tráfico y poder asegurar que este se distribuya de manera eficiente. Esta arquitectura basada en capas ayuda a proteger de posibles ataques, como de DoS o DDoS. Implementar una barrera, como un Firewall, antes de los servicios de aplicación nos permite detener tráfico maligno en capas más bajas como 3 o 4 antes de que llegue a Capa 7 y se deban gastar más recursos computacionales, como file descriptors, CPU y memoria, en hacer el filtrado en la capa de aplicación.
5. La estructura basada en contenedores permitió un diseño modular en el que cada servicio puede escalar independientemente. En Kubernetes, esto se puede lograr fácilmente por medio de ReplicaSets. Se puede incrementar la cantidad de servidores corriendo y el plano de control gestiona la cantidad de contenedores que están corriendo, cómo escalar y cómo recuperarse ante pérdidas. Igualmente, en AWS se podría realizar algo similar por medio de recursos como Auto Scaling Groups.
6. Utilizar los servicios de cloud providers facilita la conexión entre dispositivos. Con estos pudimos realizar llamadas entre dos dispositivos diferentes. Por lo tanto, la infraestructura en la nube sí tiene esta ventaja que la implementación de Kubernetes. Con Kubernetes, se estaba limitado a una sola red local. Con AWS, las llamadas en Asterisk se pueden realizar desde diferentes redes ya que se usa el mismo IP público.
7. Al tener una estructura base en Kubernetes, se logró una infraestructura independiente del entorno físico, por lo que facilitó la migración del sistema local a un servicio de cloud. No obstante, la migración a la nube no fue trivial. Originalmente, habíamos pensado que pasarlo a un cloud provider era simplemente descargar Kubernetes en una instancia EC2 y ejecutar los helm charts, igual que como se realiza en la computadora local. No obstante, esto no sirvió por limitaciones computacionales de las instancias del “Free Tier”. Por lo tanto, se decidió automatizar el despliegue en la nube por medio de Terraform. Después de la curva de aprendizaje inicial, el desarrollo con Terraform fue mucho más

intuitivo. Es importante haber tenido un acercamiento previo a usar cloud providers para acelerar la comprensión de la herramienta de Terraform.

8. El uso de herramientas de pruebas y simulación como Postman facilita probar el rendimiento del sistema mediante solicitudes HTTP. Adicionalmente, el uso de herramientas como Wireshark nos ayuda a hacer pruebas sobre el tráfico de red en protocolos SIP y VoIP, dándonos una **observabilidad** más detallada sobre la pérdida de paquetes, latencia y posibles problemas de enrutamiento. Wireshark fue extremadamente útil para resolver un problema en el cual las llamadas no colgaban cuando un extremo colgaba. Además, las llamadas se cortaban solas a los 30 segundos. Gracias a Wireshark, pudimos determinar que fue por causa de que no se lograba establecer una conexión segura y los clientes de teléfono no recibían la señal de "BYE" del protocolo SIP. Esto nos llevó a incorporar las opciones de `external_media_address` y `external_signaling_address` en la configuración, las cuales correctamente resolvieron el problema.
9. El uso del Ingress Controller junto con los recursos de Ingress en Kubernetes automatiza el enrutamiento del tráfico, simplificando la configuración y gestión del sistema. Esto facilita la implementación de servicios, reduce la complejidad y permite administrar más fácilmente el tráfico dentro del clúster.
10. Por capacidad de configuración flexible, NGINX permite un control preciso del tráfico dentro de un clúster, mejorando la seguridad y optimizando el acceso a los servicios. Esto se refleja en su capacidad para definir rutas personalizadas, aplicar restricciones de seguridad y gestionar conexiones seguras. Por el lado del Ingress Controller, este componente nos ayuda a compartir una única dirección IP y poder redireccionar a varios servicios. En lugar de tener dos IPs públicas para el Apache1 y el Apache2, podemos tener una sola con el proxy de Nginx y redireccionar el tráfico internamente. Este proceso es transparente para el usuario.