Universidad Zaragoza
1542

Trabajo Fin de Grado

Análisis de escenarios de despliegue con Kubernetes de módulos de procesamiento de señalización SIP

Analysis of deployment scenarios of SIP signaling processing modules with Kubernetes

Autor

Eduardo Sánchez Sánchez

Director

Fernando Orús Morlans

Ponente

Julián Fernández Navajas

Escuela de Ingeniería y Arquitectura

2021

Análisis de escenarios de despliegue con Kubernetes de módulos de procesamiento de señalización SIP

Resumen

El uso de las tecnologías que sustentan los servicios de telefonía está sujeto a las condiciones personales de los clientes que las utilizan y, por tanto, el nivel de carga que requiere una arquitectura capaz de proveer de estos servicios es igualmente cambiante. Esto se acentúa en situaciones como días festivos, o en un caso más reciente y generalizado, la pandemia de COVID-19, durante la cual el uso de estos servicios ha aumentado considerablemente.

Este Trabajo de Fin de Grado aborda la problemática que reside en el uso de una plataforma capaz de adaptarse automáticamente a estos cambios, con el objetivo de proveer de los servicios necesarios a un gran número de usuarios sin hacer un uso innecesario de los recursos disponibles. Para ello, se va a trabajar sobre un modelo de procesamiento de llamadas diseñado por la empresa System One Noc & Development Solutions, S.A. (en adelante SONOC), que hace uso de las tecnologías de voz sobre IP (VoIP), de forma alternativa a la red telefónica convencional debido a su gran capacidad de integración y costes reducidos. Esta tecnología surgió con el estándar H.323 para el manejo de llamadas, pero en la actualidad el protocolo de uso más extendido (y el que vamos a utilizar) es SIP, que permite el establecimiento, mantenimiento y finalización de sesiones de voz y vídeo entre varios usuarios. SIP es un protocolo que normalmente se utiliza sobre el protocolo de transporte UDP, que no ofrece garantías sobre entrega de mensajes y no guarda el estado de las conexiones establecidas ya que SIP tiene mecanismos propios de mantenimiento de la sesión. Por otro lado, aunque menos extendido, se puede utilizar sobre TCP, que sí ofrece dichas características.

Tras el análisis previo de las tecnologías y aplicaciones que vamos a utilizar se ha implementado una primera aproximación a la arquitectura sin el uso de mecanismos de adaptación a los cambios y que, en la mayoría de casos, provoca un uso innecesario de los

recursos disponibles. A continuación, se procederá a la implementación de la arquitectura en el entorno de despliegue de contenedores Kubernetes, el cual es capaz de controlar el despliegue de las aplicaciones para hacer un uso sostenible de los mismos. Para poder realizar pruebas sobre el sistema desarrollado, utilizaremos un generador de tráfico SIP llamado SIPp, en el cual realizaremos escenarios de tráfico cambiante.

Finalmente se recogen conclusiones respecto al uso de Kubernetes y al uso de señalización SIP sobre UDP y TCP en el mismo, destacando las diferencias principales, con respecto al despliegue del sistema y aportando razones para hacer uso de uno u otro en un escenario de aplicación real, así como posibles líneas futuras de este Trabajo de Fin de Grado.

Índice de contenidos

Índice de figuras

Glosario de términos

ARP. Address Resolution Protocol

AWS. Amazon Web Services

B2BUA. Back to Back User Agent

CLI. Command-Line Interface

DNS. Domain Name Server

CIDR. Classless Inter-Domain Routing

CPU. Central Processing Unit

DTMF. Dual-Tone Multi-Frequency

EC2. Elastic Compute Cloud

GPU. Graphics Processing Unit

HTTP. Hypertext Transfer Protocol

HPA. Horizontal Pod Autoscaler

JSON. JavaScript Object Notation

NAT. Network Address Translation

NGCP. ng Control Protocol

IP. Internet Protocol

IPVS. Internet Protocol Virtual Server

QoS. Quality of Service

RTP. Real Time Protocol

SFTP. Secure File Transfer Protocol

SIP. Session Initiation Protocol

SONOC. System One Noc & Development Solutions, S.A.

SSH. Secure Shell

SQL. Structured Query Language

TCP. Transport Control Protocol

TLS. Transport Layer Security

UA. User Agent

UAC. User Agent Client

UAS. User Agent Server

UDP. User Datagram Protocol

URI. Uniform Resource Identifier

URL. Uniform Resource Locator

VoIP. Voice over Internet Protocol

XML. Extensible Markup Language

YAML. YAML Ain't Markup Language

1. INTRODUCCIÓN

1.1. INTRODUCCIÓN Y MOTIVACIÓN

El constante avance tecnológico no sólo ha supuesto un cambio en la forma en la que se ofrecen los servicios de comunicaciones, sino además una continua necesidad de evolución de estos servicios, dando lugar a un compromiso entre ofrecer aquello que un usuario es capaz de entender con facilidad y aquello que tecnológicamente ofrece mejores prestaciones para el proveedor.

De entre todos los medios de comunicación actuales, la red de telefonía se ha mantenido vigente como estándar de comunicación, debido a su facilidad de uso. En sus orígenes (y en parte, en la actualidad), la red de telefonía pública consistía en centrales de conmutación que comunicaban los extremos mediante una red de circuitos. Esto sin duda era muy costoso y poco eficiente. Debido a esto, surgió Voice over IP (VoIP), que permite el establecimiento de llamadas de voz a través de Internet sin necesidad de circuitos reales, lo cual abarató los costes y permitió integrar los sistemas de comunicación de voz con el resto de servicios que ofrece Internet. Pese a ello, el coste de mantenimiento de las plataformas hardware dedicadas que ofrecen estos servicios, (y no sólo los de telefonía, sino de todo Internet) sigue siendo elevado. Para solucionar este problema, los sistemas evolucionaron a utilizar máquinas virtuales, lo cual abarataba el coste, aunque seguía siendo elevado. En los últimos años, están surgiendo alternativas al uso de máquinas virtuales que permiten un uso más eficiente de los recursos de hardware: es el caso de la arquitectura de microservicios, y en su extensión, del paradigma de contenedores de software.

La arquitectura de microservicios consiste en la construcción de aplicaciones complejas mediante la separación en aplicaciones que ejecutan procesos más sencillos e independientes y que se comunican mediante mecanismos ligeros. Este concepto está muy ligado con el de contenedor de software, que permite alojar los microservicios en entornos aislados haciendo un uso mínimo de los recursos físicos. Es el caso de Docker, que implementa contenedores de software haciendo uso de las funcionalidades del kernel y el aislamiento de recursos para crear una versión simplificada de un sistema operativo que

ejecuta procesos de forma independiente y que se pueden comunicar entre sí haciendo uso de redes virtuales. El uso de Docker se ha generalizado en la actualidad debido a la versatilidad de las aplicaciones que implementa. En particular, el uso de Docker sobre Kubernetes, que es una herramienta que facilita el uso de estos contenedores.

En este contexto, resulta útil gestionar de forma automática estos contenedores, creando o eliminando réplicas (llamadas instancias) de los mismos en un conjunto de equipos hardware que formen parte de una misma arquitectura, llamada clúster. Esta función está implementada por los denominados *"orquestadores":* elementos capaces de gestionar las interacciones entre contenedores de software e integrar los servicios que implementan, a la vez que permite la automatización en el despliegue de los contenedores que implementan un microservicio, de forma que en cada momento sólo se utilice una cantidad de recursos adecuada a las necesidades del servicio, en nuestro caso el establecimiento y liberación de comunicaciones VoIP.

La herramienta Kubernetes permite esta orquestación de contenedores, automatizando el despliegue, ajuste de escala y manejo de aplicaciones implementadas mediante Docker. Para aclarar este concepto, en la *Figura 1. Ejemplo de orquestador para servicios web* se puede observar un ejemplo de funcionamiento de Kubernetes como herramienta de orquestación aplicada a un servicio web. En ella, una población indeterminada de equipos externos desea acceder a los servicios web mediante peticiones HTTP, que alcanzan al plano de control de Kubernetes, que realiza la toma de decisiones para cada petición, enviándola a cada contenedor existente. Además, el plano de control realiza medidas constantes sobre el uso de recursos de cada contenedor, pudiendo así controlar la escala del despliegue de la aplicación.

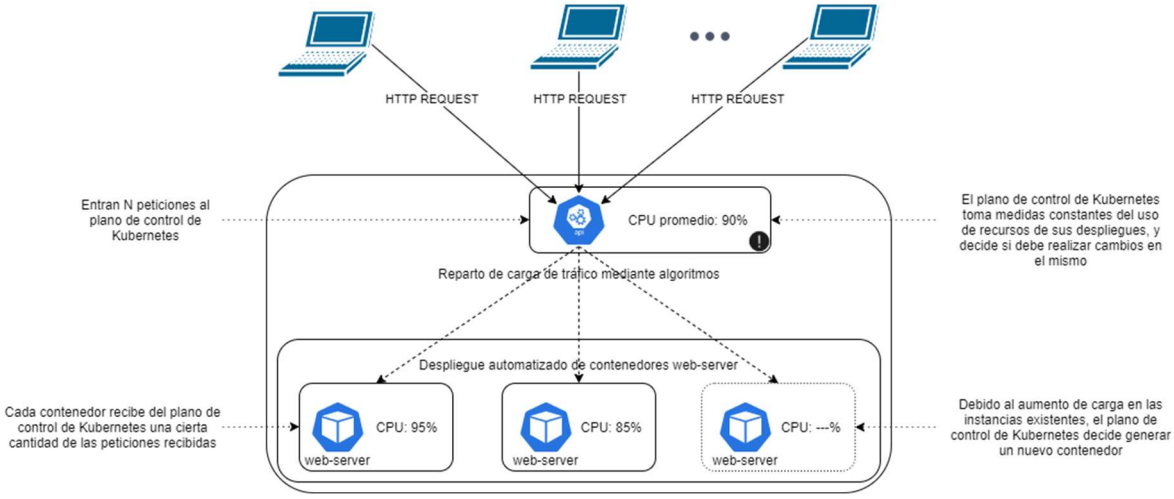

Figura 1. Ejemplo de orquestador para servicios web

Para finalizar, diremos que este Trabajo de Fin de Grado se realiza en el contexto de trabajo de la empresa SONOC, que se encarga, entre otras cosas de proveer servicios de VoIP. Además de esto, SONOC implementa soluciones de mejora de los servicios que ofrecen, como herramienta de realce de la voz, funcionalidades mediante marcado de tonos DTMF, etc. El objetivo de este proyecto es la implementación de una arquitectura de servicios VoIP que optimice el uso de recursos sin comprometer la calidad de los servicios que ofrecen.

1.2. OBJETIVOS

El objetivo principal de este Trabajo Fin de Grado es la comparativa entre las distintas posibilidades planteadas para el diseño de una arquitectura capaz de proveer servicios VoIP masivos, destacando las ventajas y desventajas de cada una, teniendo en cuenta aspectos como la escalabilidad, el uso eficiente de los recursos hardware y las posibles problemáticas que pueden surgir del uso de cada una de ellas.

Así pues, las exponemos a continuación. En primer lugar una implementación tradicional (con cada elemento localizado en equipos independientes, que pueden ser indistintamente máquinas reales o virtuales). En segundo lugar, una implementación

utilizando contenedores de software (Docker con Kubernetes) sin orquestar. Por último propondremos una implementación que permita la orquestación utilizando Kubernetes.

En todos los casos, la arquitectura a desarrollar será similar, variando únicamente el enfoque necesario para adaptar cada implementación al entorno de desarrollo. Esta arquitectura común ha sido diseñada por SONOC y hace uso de señalización SIP para el establecimiento, control y finalización de llamadas simples. En este proyecto, únicamente nos vamos a centrar en los aspectos fundamentales que son necesarios para ofrecer un servicio de telefonía simple y que sea comercial, con el fin de llegar a conclusiones sobre la viabilidad del uso de Kubernetes sobre la implementación actual.

Finalmente, analizaremos el uso de Kubernetes diferenciando la señalización SIP mediante los protocolos de transporte TCP y UDP, que ofrecen prestaciones distintas en un contexto de uso sobre contenedores orquestados y, por tanto, volátiles (es decir, cada contenedor puede ser creado o eliminado en base a una alta o baja demanda de servicios, respectivamente). En concreto, la problemática principal que plantea una implementación de servicios de VoIP sobre contenedores orquestados es el mantenimiento de las conexiones VoIP sobre UDP (sólo haremos esta comparativa en la arquitectura orquestada, que es la que plantea este problema). Una baja demanda de servicios puede implicar la eliminación de réplicas existentes que se encuentran eventualmente sirviendo conexiones UDP a un número bajo de clientes y de los que no existe constancia, provocando el cierre abrupto de las conexiones y molestias tanto a cliente como a proveedor. A su vez, existe la problemática de mantener contenedores activos durante largo tiempo sirviendo una vez más a un número bajo de clientes.

Para el desarrollo se usarán herramientas gratuitas y open-source, a excepción de los servicios de computación de Amazon Web Services, que albergarán las aplicaciones *proxy* SIP, *proxy* RTP, *softswitch* y servidor de rutas.

1.3. ORGANIZACIÓN DE LA MEMORIA

La memoria se compone de los siguientes capítulos:

- Capítulo 1: Introducción y objetivos del proyecto.
- Capítulo 2: Análisis previo al desarrollo del trabajo, en el que se introduce los elementos básicos que vamos a utilizar, como protocolos, lenguajes de programación y aplicaciones.
- Capítulo 3: Implementación de la arquitectura sin contenedores. Se describen los elementos utilizados y la arquitectura de red que presenta.
- Capítulo 4: Implementación de la arquitectura con contenedores. Se describen las diferencias de implementación respecto al capítulo anterior, y las ventajas que se observan.
- Capítulo 5: Pruebas de comunicación y de carga de tráfico sobre los escenarios planteados. Se describen ventajas y desventajas de cada uno y posibles soluciones a los problemas que plantea cada uno de ellos.
- Capítulo 6: Conclusiones finales y posibles líneas de trabajo futuras en base al trabajo realizado.

Por último, se incluirá una bibliografía con las fuentes utilizadas y anexos que complementan la información sobre el trabajo realizado.

2. ANÁLISIS PREVIO

2.1. PROTOCOLOS, LENGUAJES DE PROGRAMACIÓN Y HERRAMIENTAS UTILIZADAS

En el siguiente apartado vamos a enumerar los protocolos, lenguajes de programación y herramientas utilizadas durante el desarrollo del trabajo. En el *Anexo L: Descripción de las herramientas utilizadas* se encuentra una descripción breve de las mismas y que sirve para la comprensión superficial del trabajo realizado. Aquellos elementos que requieran una comprensión más en profundidad cuentan con su propio anexo que será referenciado cuando sea necesario.

Protocolos

- SIP (Session Initiation Protocol)
- RTP (Real-Time Protocol)
- NGCP (ng Control Protocol)

Lenguajes de programación

- XML (Extensible Markup Language)
- YAML (YAML Ain't Markup Language)

Herramientas utilizadas

Servidores

- Kamailio
- Freeswitch
- RTPEngine

Aplicaciones

- Docker

- Kubernetes
- SIPp
- Zoiper
- Wireshark/TCPDump
- WinSCP
- Github
- Dockerhub

2.2. PLANTEMIENTO DEL PROBLEMA

Durante la introducción se han mencionado los tres enfoques posibles del problema: una solución consistente en la separación de los elementos hardware que implementan el servicio, otra solución basada en el uso de contenedores mediante Docker y Kubernetes, pero sin hacer uso de la orquestación, y una solución análoga a la anterior, pero usando orquestación. Estas arquitecturas deben ser capaces de establecer llamadas entre múltiples usuarios a través de la red VoIP utilizando el protocolo SIP para la señalización de llamadas y RTP para el tráfico de audio de la llamada.

El punto de partida de este trabajo es la *Figura 2. Arquitectura previa,* en la que se describen los elementos de conmutación utilizados por la empresa. El objetivo será crear sendas arquitecturas de pruebas que incluyan los elementos de procesamiento para la señalización y contenido de las llamadas.

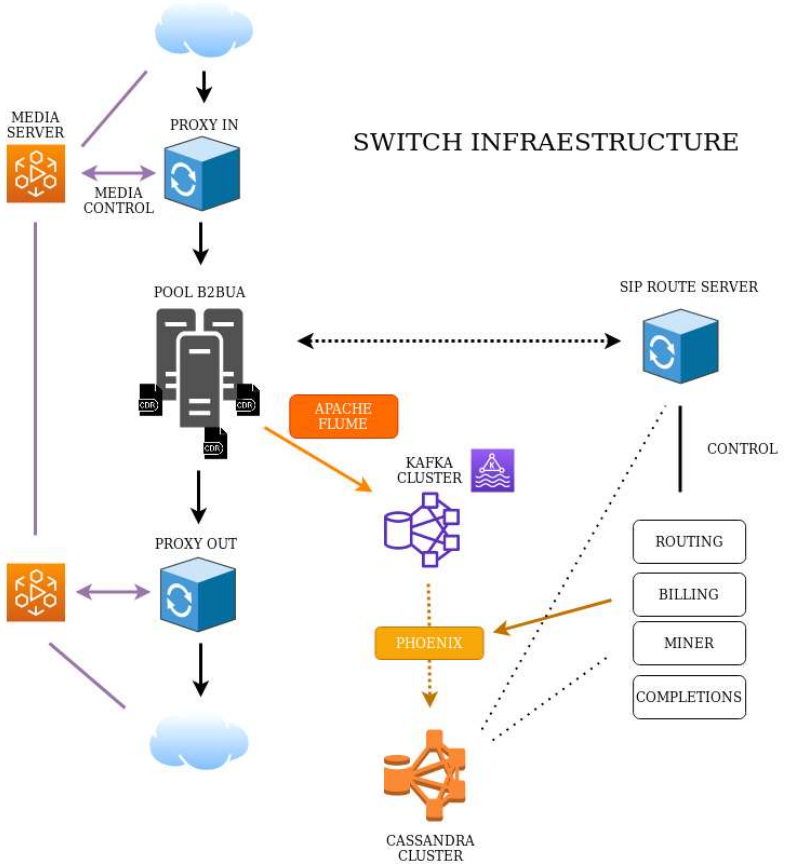

Figura 2. Arquitectura previa

De esta forma, los elementos que vamos a crear en nuestra arquitectura son: PROXY IN, MEDIA SERVER, POOL B2BUA, SIP ROUTE SERVER y PROXY OUT; elementos encargados del establecimiento de llamadas mediante SIP y envío de paquetes de datos mediante RTP, siendo el resto de elementos únicamente necesarios para fines de facturación y obtención de estadísticas. Finalmente, realizaremos una comparativa entre los protocolos TCP y UDP según su viabilidad para su uso en un entorno de contenedores orquestados, siendo necesaria, para ello, la configuración de cada elemento de la arquitectura.

Por último, en todas las aproximaciones al problema planteado, se creará una arquitectura de pruebas, con los elementos más o menos diferenciados, y en todos los casos la arquitectura se encontrará perfectamente localizada en la nube de Amazon Web Services. La terminación de las llamadas en el escenario real serán distintos proveedores de servicios de VoIP, que actúan como pasarelas para la conmutación de llamadas en regiones distintas. En nuestro trabajo y por simplicidad, trabajaremos con un único proveedor de servicios que simplemente conteste todas las llamadas. De esta forma, en la *Figura 3. Escenario previo,* podemos ver cómo quedaría a grandes rasgos la arquitectura que planteamos en un escenario de aplicación real.

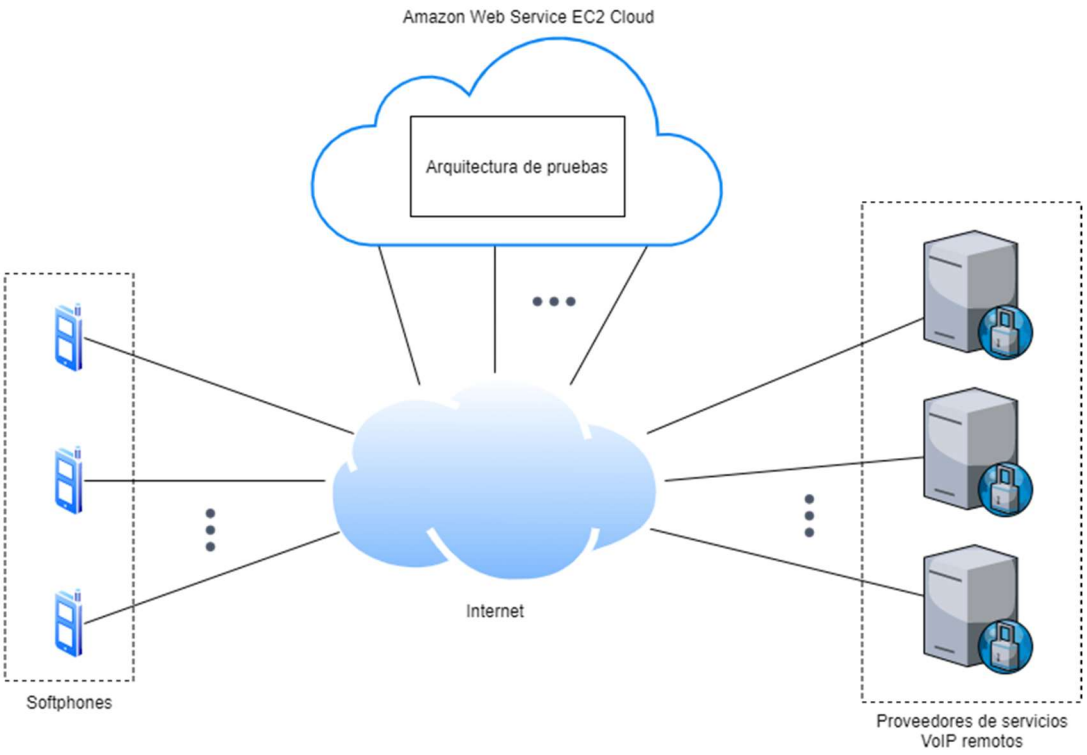

Figura 3. Escenario previo

3. IMPLEMENTACIÓN VIRTUALIZADA SIN CONTENEDORES

3.1. PLANTEAMIENTO GENERAL

La primera aproximación a este problema consiste en establecer una arquitectura de provisión de servicios VoIP (o switch) donde las diferentes aplicaciones que hemos descrito en el *Capítulo 2.2. Plantemiento del problema* se encuentran completamente separadas, es decir; que cada uno de los elementos esté en equipos diferenciados, de forma que cualquier *softphone* o elemento compatible con las redes VoIP a través de los estándares SIP y RTP pueda completar llamadas. Por tanto, la comunicación entre los elementos se hará a través de las interfaces de red externas de cada máquina.

La arquitectura más simple que podemos plantear consistiría en sendos elementos que sirvan como proxy SIP y RTP entre dos usuarios. Para ello, utilizamos Kamailio y RTPEngine respectivamente. La función de estos proxys es simplemente la de reenviar los paquetes generados por un extremo hacia el otro. Este acercamiento, que queda representado en la *Figura 4. Esquema simple de plataforma VoIP,* es muy limitado y no ofrece ningún mecanismo de gestión de llamadas, pudiendo únicamente comunicar usuarios registrados en un mismo dominio. La comunicación entre ambos *proxy* se hace mediante el protocolo propio de Sipwise, NGCP. Este protocolo sirve para hacer conocedor a RTPEngine de las llamadas establecidas por Kamailio y de los usuarios que participan en ellas, a fin de establecer conexiones RTP con ellos.

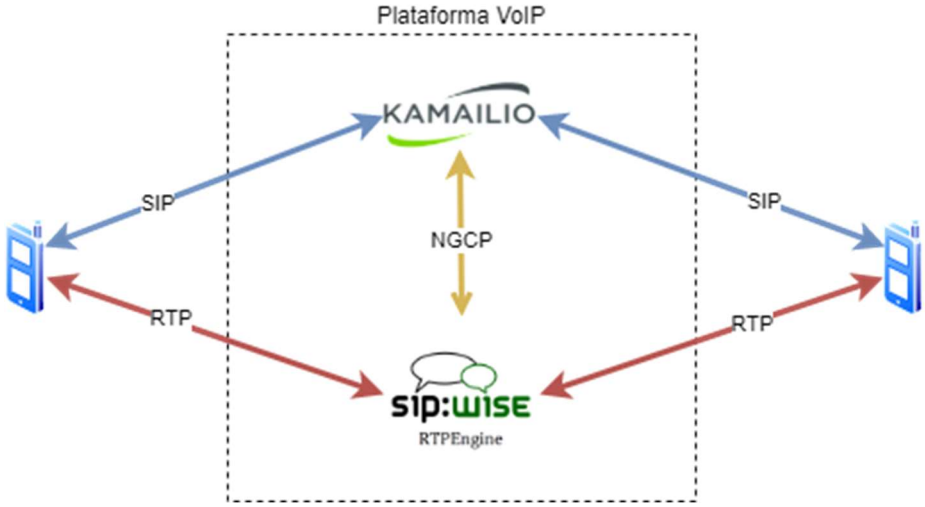

Figura 4. Esquema simple de plataforma VoIP

Este esquema, si bien puede servir como prueba, sólo ofrece la funcionalidad de establecimiento de llamadas. Es por ello que en nuestra plataforma añadiremos un *softswitch*, que como hemos descrito anteriormente, realizará las funciones de integración, de forma que no haya problemas entre usuarios que utilicen clientes VoIP diferentes, así como las diversas funciones adicionales que se quieran incluir (marcación de tonos, buzón de voz, etc.). La inclusión de este *softswitch* implica la creación de un segundo *proxy* SIP, que junto al que ya teníamos servirán, además, de filtrado de tráfico externo, impidiendo llamadas no deseadas y maximizando los recursos que ofrece el *softswitch* a las llamadas aceptadas. Todo esto queda representado en la *Figura 5. Esquema de plataforma VoIP con softswitch.* El tráfico NGCP en este caso se generará desde el Kamailio que actúa como *proxy* de salida, puesto que usando la información de las cabeceras es capaz de transmitir la dirección en la que se encuentran los usuarios que participan de la llamada, permitiendo al *proxy* RTP establecer las conexiones con los extremos.

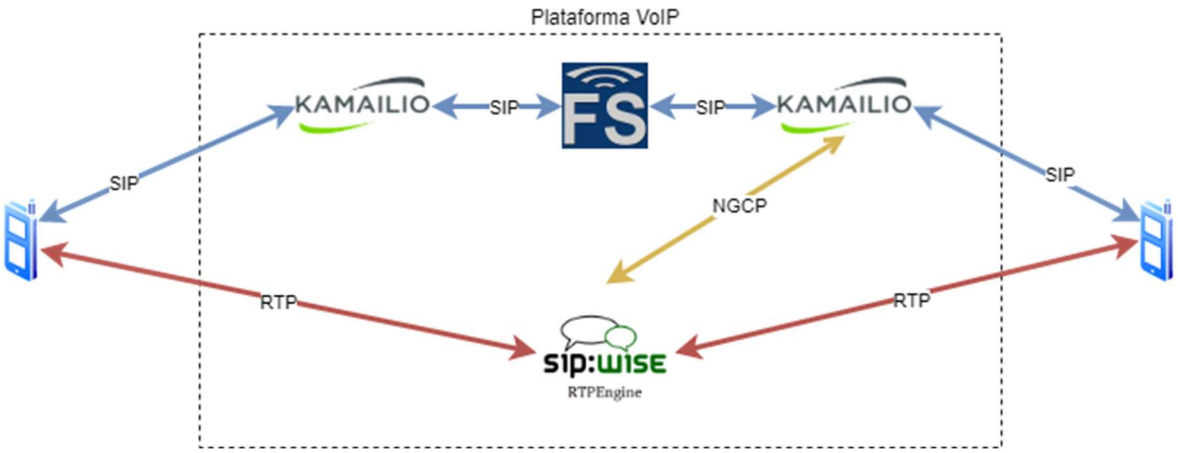

Figura 5. Esquema de plataforma VoIP con softswitch

Esta arquitectura nos permite crear sesiones entre usuarios, y además aporta las funcionalidades de un *softswitch*. Si bien esto ya es perfectamente funcional, sólo permite llamadas entre usuarios residentes en un dominio limitado. Esto significa que usuarios de distintas compañías telefónicas no podrán comunicarse entre sí. Por ello, nuestra plataforma de VoIP deberá poder enviar las llamadas a distintos proveedores de servicios VoIP remotos (en distintos lugares y sirviendo a otros usuarios) que se encargarán de terminar las llamadas usando sus propios *switch*, y cuyas direcciones obtendremos de un servidor de rutas basado en reglas mediante Kamailio. En la *Figura 6. Esquema de plataforma VoIP con servidor de rutas,* observamos este nuevo acercamiento, que permite el establecimiento de llamadas con los proveedores conocidos. Las peticiones de ruta se harán en el *softswitch,* y la información obtenida del servidor se llevará desde el *softswitch* hasta el proxy de salida, que encaminará la llamada al proveedor correspondiente, y a la vez será capaz de comunicar esta información al *proxy* RTP, que establecerá conexiones tanto con el usuario llamante como el llamado.

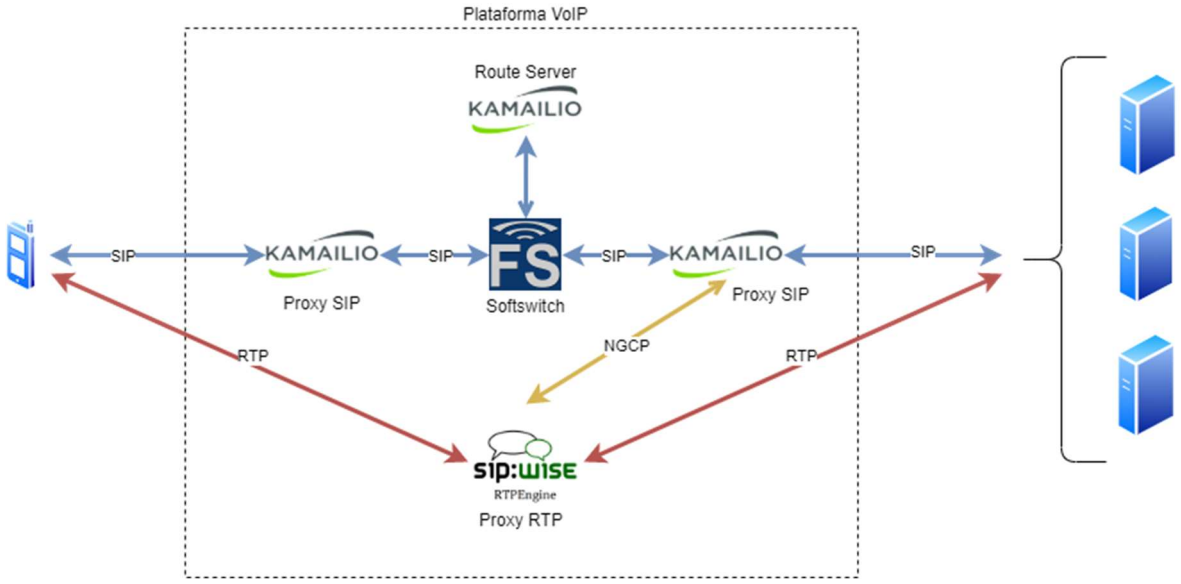

Figura 6. Esquema de plataforma VoIP con servidor de rutas

Por último, como no es necesario acceder a proveedores de servicios en explotación, se ha simulado uno utilizando Freeswitch. Este nuevo elemento servirá como única ruta para nuestras llamadas y tendrá la función de contestar las llamadas entrantes y reproducir un fichero de audio. Debido a que la existencia o no de este elemento no es importante para el desarrollo del trabajo, en consiguientes apartados y figuras no se mencionará este aspecto.

3.2. INFRAESTRUCTURA DE RED

Visto el planteamiento del problema con respecto a los servidores que vamos a utilizar, el siguiente paso es ubicar dichos servidores en un entorno en el que estos puedan comunicarse entre sí. Para ello vamos a utilizar la plataforma AWS. En concreto, de la plataforma de cómputo en la nube EC2, que nos permite crear máquinas virtuales con recursos variables. En el *Anexo B: Amazon Web Services* se da más información sobre la configuración de instancias EC2 y cómo permitir la comunicación entre ellas.

Toda la arquitectura, a excepción de la implementación de RTPEngine y los *softphones* que actúen como usuarios finales; se localizará sobre equipos virtualizados, por lo que la conexión entre los elementos no será directa, al no pertenecer a redes privadas compartidas.

Las instancias que crearemos correrán sobre un kernel de Linux, en concreto utilizarán el sistema operativo Debian 9.13, que ofrece compatibilidad con las herramientas que vamos a utilizar.

En un escenario real, la capacidad de cómputo de múltiples llamadas simultáneas de una instancia con RTPEngine es limitada debido a la necesidad de procesado de flujos RTP (equivalente a cientos de paquetes por segundo), por lo que la máquina (en realidad conjunto de máquinas) donde se aloje será proporcionada por la empresa y cuyo dimensionado no será objeto de estudio en el presente trabajo. En la *Figura 7. Escenario de red sobre EC2* podemos observar el escenario descrito, donde se diferencian los elementos en distintos equipos localizados en la nube EC2 y en un equipo físico con GPU.

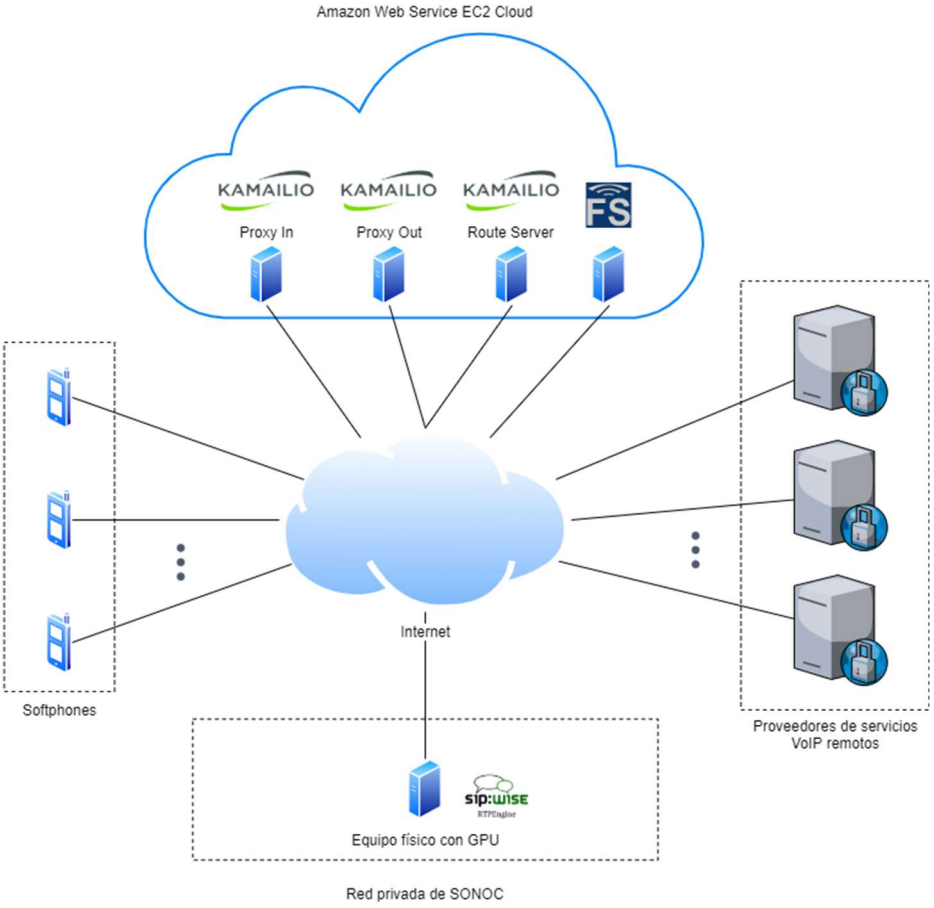

Figura 7. Escenario de red sobre EC2

3.3. COMUNICACIÓN DE SERVIDORES

Como hemos dicho anteriormente, los servidores que formarán parte de nuestro *switch* se encuentran alojados en servidores remotos alojados en AWS, por lo que es necesario configurar correctamente dichos equipos para obtener un funcionamiento síncrono y correcto, capaz de servir múltiples llamadas simultáneas.

Dado que las máquinas alojadas en AWS forman parte de redes virtuales privadas protegidas con *firewalls* (lo que supone grupos de seguridad), en un escenario por defecto, el único puerto activo para acceder a los recursos del equipo es el 22 de TCP, usado para conexiones SSH, y cualquier otra conexión que realicemos será rechazada, por lo que el primer paso será configurar el *firewall* de cada equipo para permitir conexiones entrantes y salientes a través de los puertos correspondientes. Todas las aplicaciones que usamos pueden configurarse para utilizar los puertos que se deseen para todas las conexiones, pero nosotros, por conveniencia, haremos uso de los puertos por defecto de cada protocolo.

- En el caso de SIP, el puerto por defecto es el 5060 de TCP o UDP. Utilizaremos este último. Adicionalmente, si se usa TLS, el puerto por defecto es el 5061.
- En el caso de RTP, el *proxy* abre dos conexiones, una con cada extremo de la conexión a través de un puerto aleatorio entre el 16384 y el 32768 de UDP, por lo que los abriremos todos a fin de servir tantas llamadas como sea posible.
- En el caso de NGCP, el puerto por defecto es el 22222 de UDP.

En el *Anexo B: Amazon Web Services* se ofrece información sobre los firewalls de red y como abrir los puertos indicados. Para comprobar que un determinado puerto se encuentra abierto podemos hacer uso del siguiente comando desde una terminal de Linux conectada por SSH al equipo remoto (para más información sobre como realizar este acceso ver *Anexo H: Creación de par de claves y acceso SSH*):

- `nmap <-sT,-sU> localhost -p <port>`

En la *Figura 8. Comprobación de puertos abiertos usando nmap* se observa un ejemplo de ejecución de dicho código para comprobar que el puerto 22 de TCP (que claramente está abierto porque hemos abierto una conexión SSH) se encuentra abierto.

```
Seleccionar esanchez@ip-10-3-0-120: ~
root@ip-10-3-0-120:/home/esanchez# nmap -sT localhost -p 22

Starting Nmap 7.40 ( https://nmap.org ) at 2021-09-07 14:57 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00059s latency).
PORT   STATE SERVICE
22/tcp open  ssh

Nmap done: 1 IP address (1 host up) scanned in 0.23 seconds
root@ip-10-3-0-120:/home/esanchez#
```

Figura 8. Comprobación de puertos abiertos usando nmap

Una vez permitidas las conexiones entre equipos a través de los puertos, únicamente hace falta instalar los servidores correspondientes en los diferentes equipos (ver *Anexo D: Instalación y configuración de servidores* para más información) y posteriormente realizar las modificaciones pertinentes en los ficheros de configuración de cada servidor. Estas modificaciones se harán usando una conexión SSH remota desde un equipo local utilizando WinSCP, un cliente que presenta una interfaz gráfica para la transferencia y edición de ficheros, perfecto para lo que necesitamos. En el *Anexo H: Creación de par de claves y acceso SSH* se describe como utilizar este programa. Una vez hecho esto, podemos iniciar cada aplicación mediante sus servicios, haciendo que estas empiecen a escuchar tráfico en los puertos correspondientes y a gestionarlo de forma que se ofrezca el servicio que deseamos. La explicación sobre los ficheros de configuración que vamos a desarrollar se encuentra en el *Anexo C: Desarrollo del sistema*.

Por último, vamos a replicar el comportamiento deseado para una llamada cualquiera (esta explicación no entra en detalle sobre el procesamiento o los mecanismos propios de SIP para asegurar el establecimiento de las conexiones, lo cual se verá más adelante):

En primer lugar, un usuario envía una petición de llamada al *proxy* SIP de entrada con un mensaje “SIP INVITE”, usando una URI compuesta de un nombre de usuario, y el par

IP/puerto (5060) donde se aloja el *proxy* SIP. El destinatario de la llamada es otra URI con un nombre de usuario y el mismo par IP/puerto, lo cual vemos en la *Figura 9. Envío de mensaje INVITE a proxy SIP.*

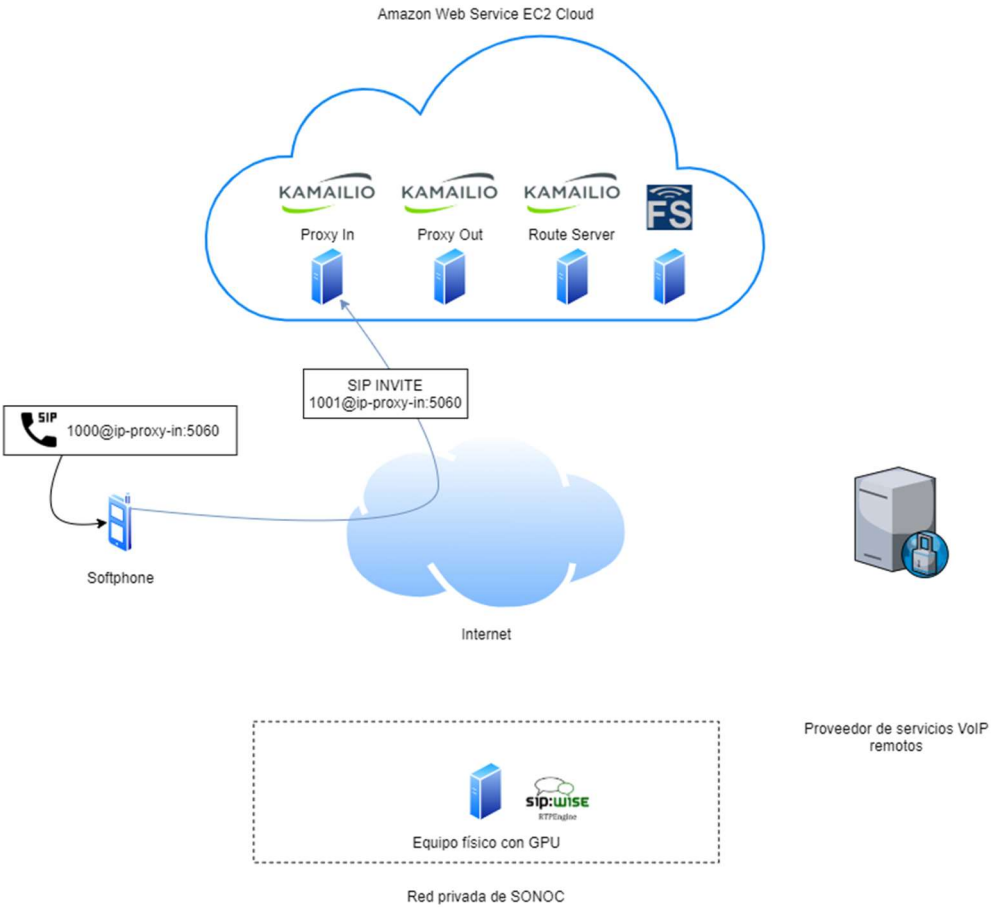

Figura 9. Envío de mensaje INVITE a proxy SIP

A continuación, el *proxy* SIP habiendo filtrado mensajes indeseados, envía el mensaje sin modificar al *softswitch*, como se ve en la *Figura 10. Reenvío de mensajes al softswitch*, y este se encarga de todos los aspectos de procesamiento e integración, así como de las funciones específicas que tienen que ver con los elementos que no procesan señalización SIP, como se vio en el *Capítulo 2.2. Plantemiento del problema.*

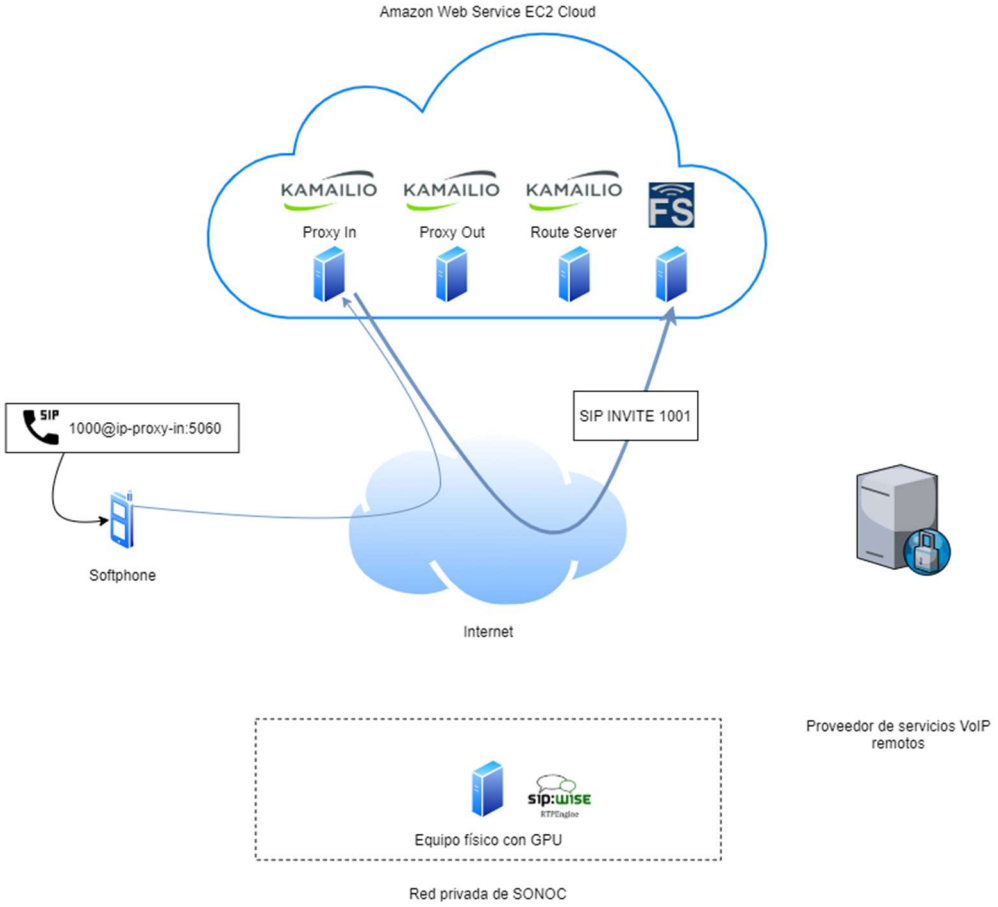

Figura 10. Reenvío de mensajes al softswitch

A continuación, el mensaje se reenvía al servidor de rutas, que lee los diferentes campos del mensaje INVITE y mediante una serie de reglas preestablecidas, indica al *softswitch* dónde se encuentra el proveedor de servicios VoIP al que hay que contactar para terminar el establecimiento de la llamada con el usuario indicado en la URI. El mensaje a través del cual envía esto es un "302 moved", al cual se le añade una cabecera con la dirección IP del proveedor. Podemos observar este comportamiento en la *Figura 11. Petición de ruta.*

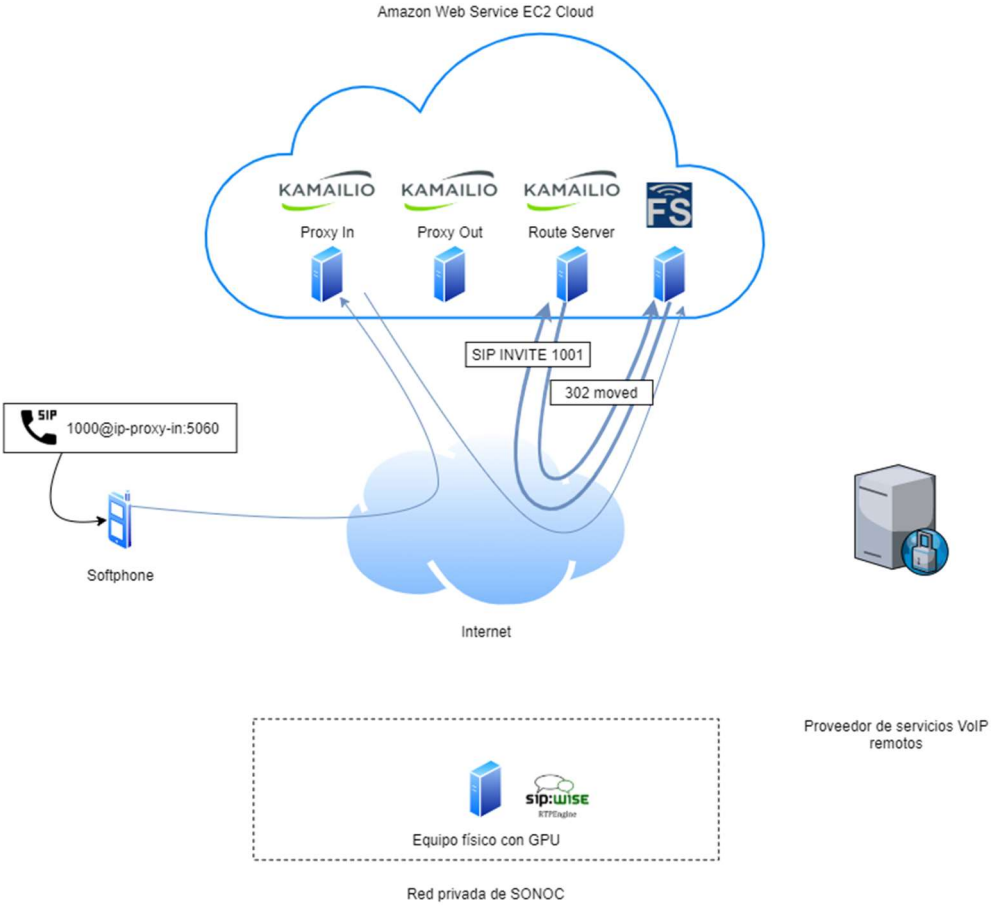

Figura 11. Petición de ruta

A continuación, la información de la cabecera es enviada por el *softswitch* al *proxy* de salida, y este, usando la cabecera correspondiente que hemos obtenido del servidor de rutas, la envía al proveedor correspondiente, que se encarga de establecer la conexión con el usuario SIP que corresponda, fuera del alcance de nuestro switch. Podemos observar este comportamiento en la *Figura 12: Finalización del establecimiento de llamada.*

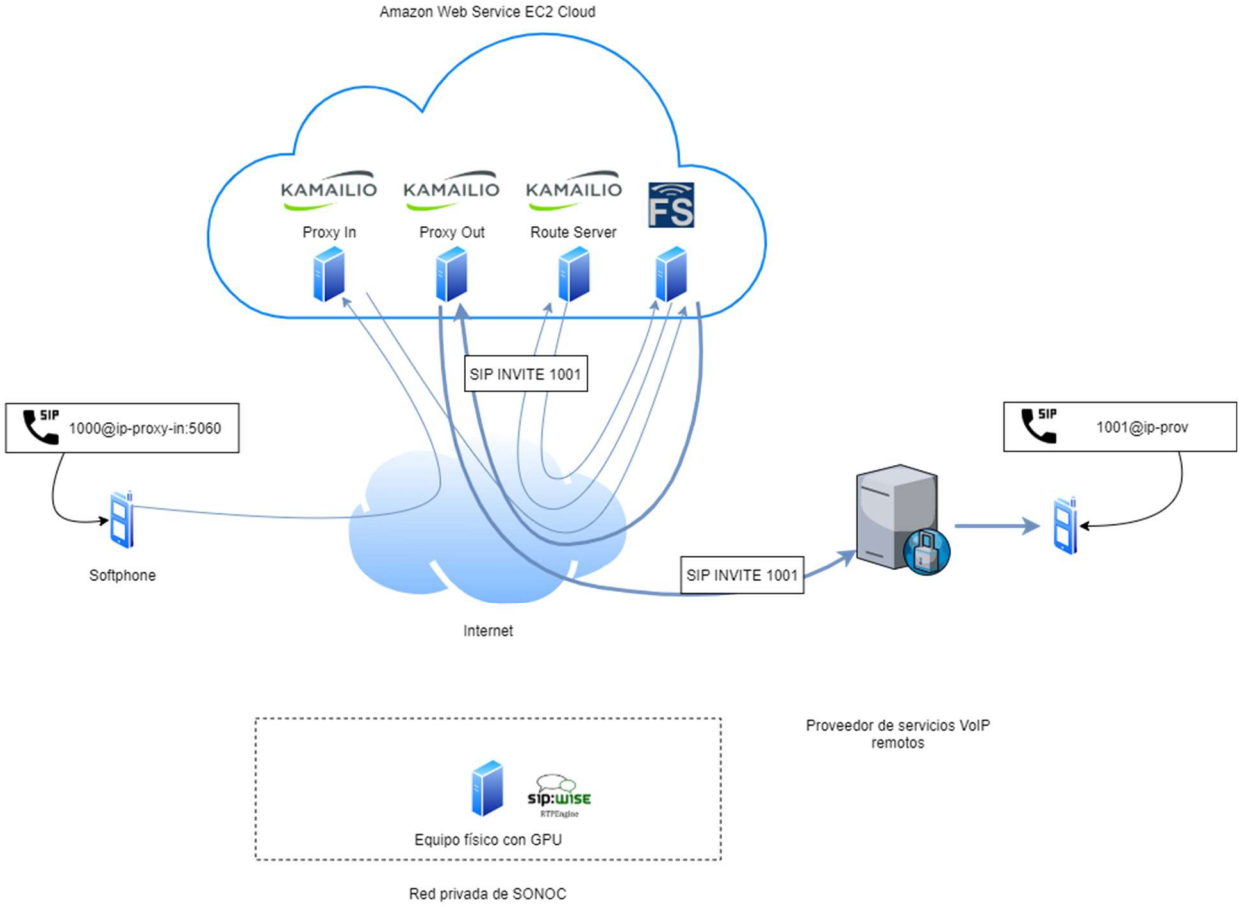

Figura 12: Finalización del establecimiento de llamada

Una vez completado el establecimiento, se envían los mensajes correspondientes para informar de que la llamada ha sido aceptada, junto con información sobre los usuarios finales y se informa de que se va a empezar a cursar. Cuando el *proxy* de salida obtiene dicha información, contacta con el *proxy* RTP a través del protocolo NGCP, indicando las direcciones IP de los extremos. Por último, el *proxy* RTP, utilizando esta información abre dos conexiones de datos, una con cada usuario que envía su propio tráfico de voz y, tras ser procesado se envía al otro extremo. Podemos observar este comportamiento en la *Figura 13. Establecimiento de conexiones RTP*

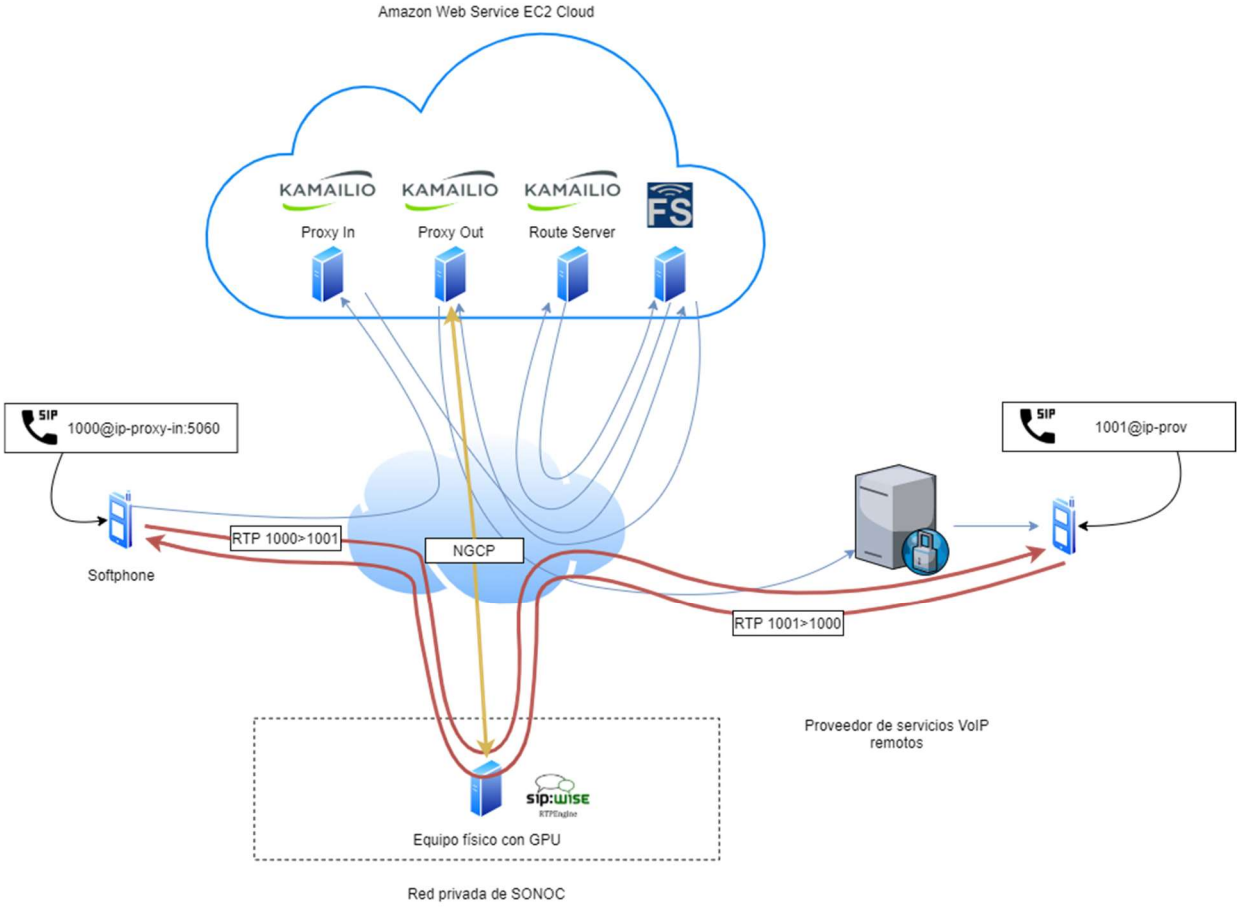

Figura 13. Establecimiento de conexiones RTP

En este momento, se dejan de enviar mensajes SIP, en cambio, únicamente se observan dos flujos constantes de tráfico RTP a través del *proxy* de este mismo protocolo, que durará hasta la interrupción o finalización de la llamada.

Finalmente, cuando un usuario desea terminar la llamada, envía un mensaje "SIP BYE", que recorre todos los elementos de procesado SIP del switch (a excepción del servidor de rutas), informándoles de la liberación de la conexión, hasta finalmente llegar al usuario en el otro extremo, que confirma esta liberación. Dada esta liberación, los usuarios finales cerrarán las conexiones que mantienen con el *proxy* RTP, el cual liberará los puertos usados en esas conexiones para servir nuevas llamadas. En este sentido, la liberación de las conexiones desde el punto de vista de RTPEngine no es explícita.

Con esto, la configuración del escenario (implementación tradicional) está completo y estaría listo para recibir llamadas. En el *Anexo J: Pruebas de llamada con Zoiper* y el *Anexo K: Pruebas de llamada con SIPp* se encuentra más información sobre la puesta en marcha de cada una de las aplicaciones, así como de la forma en la que cada una de ellas gestiona los mensajes. Adicionalmente, los resultados de dichas pruebas sobre este escenario se encuentran en el *Capítulo 5. Pruebas de comunicación y carga de tráfico.*

4. IMPLEMENTACIÓN VIRTUALIZADA USANDO CONTENEDORES MEDIANTE KUBERNETES

En este capítulo vamos a presentar los dos escenarios en los que vamos a usar la herramienta Kubernetes para el despliegue de aplicaciones en contenedores. En primer lugar, haremos un escenario sin orquestar y en base a ello, evolucionaremos hacia un escenario orquestado, presentando las diferencias entre ambas implementaciones. Los elementos de Docker y Kubernetes que se van a utilizar son comunes entre ambas implementaciones. El punto de partida de este escenario es una máquina virtual en el que ambas herramientas se encuentran instaladas, la información al respecto de esta instalación se encuentra en el *Anexo E: Instalación y configuración de otras herramientas.*

4.1. ELEMENTOS DE DOCKER

El primer paso para la implementación de cualquier aplicación en un entorno de Kubernetes es la creación de las imágenes de contenedor que queremos desplegar. Estas imágenes serán versiones reducidas de un sistema operativo funcionando sobre el kernel de la máquina real que utilizamos (Linux), aunque el sistema operativo que implementen puede ser distinto al de la máquina real.

Las imágenes se crearán a partir de un fichero Dockerfile. Las imágenes que usemos partirán de una ya creada por las desarrolladoras de las respectivas aplicaciones, por lo que no tendremos problemas de compatibilidad, y lo único que tendremos que hacer es realizar los cambios requeridos para que nuestra aplicación funcione como nosotros deseemos, como, por ejemplo, copiar los ficheros de configuración específicos que hemos desarrollado, ejecutar comandos o exponer puertos. Todo esto está explicado en el *Anexo F: Docker.*

Para este escenario se elaborarán imágenes de contenedor para cada uno de los servidores que anteriormente se encontraban virtualizados, para un total de cuatro ficheros Dockerfile, cada uno con la configuración de los elementos equivalentes del escenario anterior, y que luego utilizaremos en la implementación de objetos en Kubernetes, como se muestra en la *Figura 14. Elaboración de ficheros de imagen.*

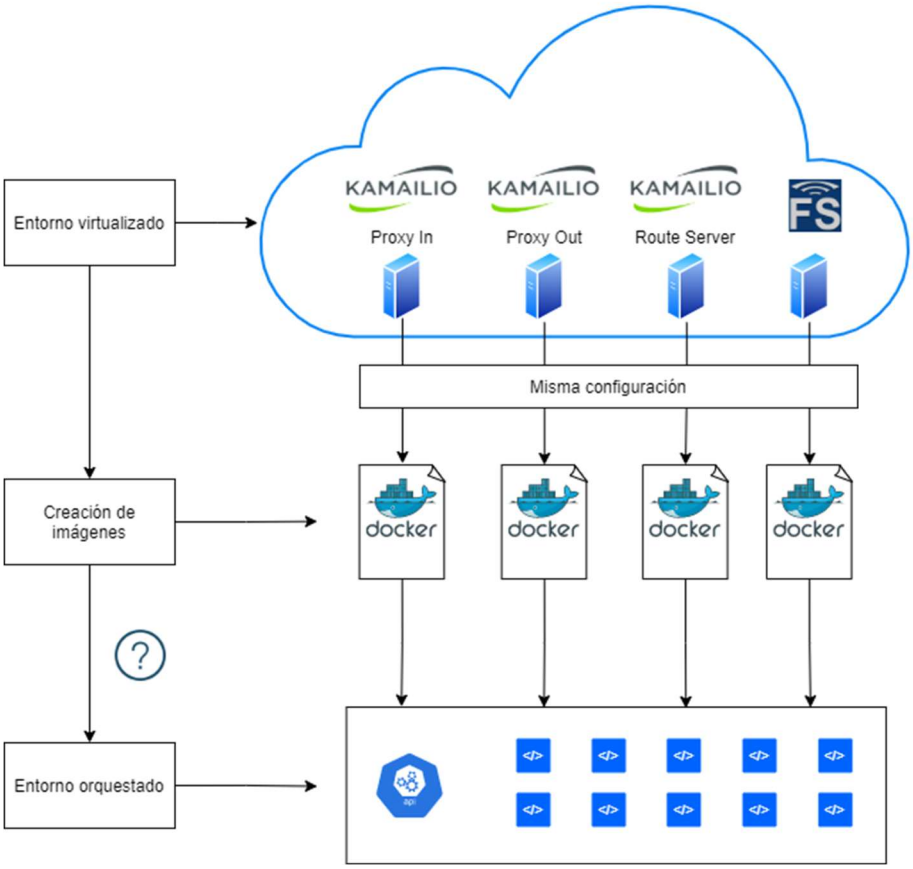

Figura 14. Elaboración de ficheros de imagen

Una vez creado un fichero de imagen, podemos guardarlo localmente utilizando:

- `docker build -t <nombre_de_imagen> <path_dockerfile>`

Y podemos hacer una prueba de funcionamiento de la imagen con el motor de Docker utilizando:

- `docker run -it <nombre_de_imagen> bash`

Lo cual ejecuta un Shell interactivo dentro del contenedor, de forma que podemos ejecutar comandos y realizar una comprobación previa de los elementos básicos de la imagen.

4.2. ELEMENTOS DE KUBERNETES

Una vez creadas las imágenes de contenedor que vamos a utilizar, podemos pasar a su implementación en un entorno de Kubernetes. Como ya se mencionó anteriormente, la estructura de Kubernetes se basa en objetos de distinta naturaleza que implementan funciones que forman parte de un todo. Estos objetos se escriben en formato YAML y tienen una estructura muy concreta. En el *Anexo G: Kubernetes*, se describe con precisión la estructura de los distintos objetos.

El entorno en el que se mueve Kubernetes se llama clúster, que es un conjunto de equipos reales con Kubernetes donde se alojan los recursos de una misma aplicación, de forma que esta no se encuentra localizada en un único equipo, sino que está distribuida a través de múltiples nodos que forman parte de un clúster.

La unidad mínima de abstracción en Kubernetes (en cuanto a recursos que implementan funcionalidades de contenedores) es el *pod*, que consiste en uno o más contenedores de software que comparten recursos de hardware definidos y que tienen una misma dirección IP, y con un ciclo de vida que les da un carácter efímero (pueden ser creado y eliminados mediante diferentes mecanismos). Además, no poseen una dirección IP estática, por lo que se requieren de otros elementos para enviar y recibir peticiones desde otro *pod*.

Por otra parte, un *deployment* es la definición de un conjunto de *pod* iguales, en los contenedores que implementa y que pueden estar distribuidos a lo largo de distintos equipos del clúster. En un mismo clúster pueden existir diferentes *deployment*, cada uno implementando la funcionalidad de los contenedores definidos en un único *pod*. Dada esta definición, en la *Figura 15. Abstracción de aplicaciones en Kubernetes* se encuentra un ejemplo de cómo se distribuyen las aplicaciones en contenedores en un clúster de Kubernetes con múltiples nodos.

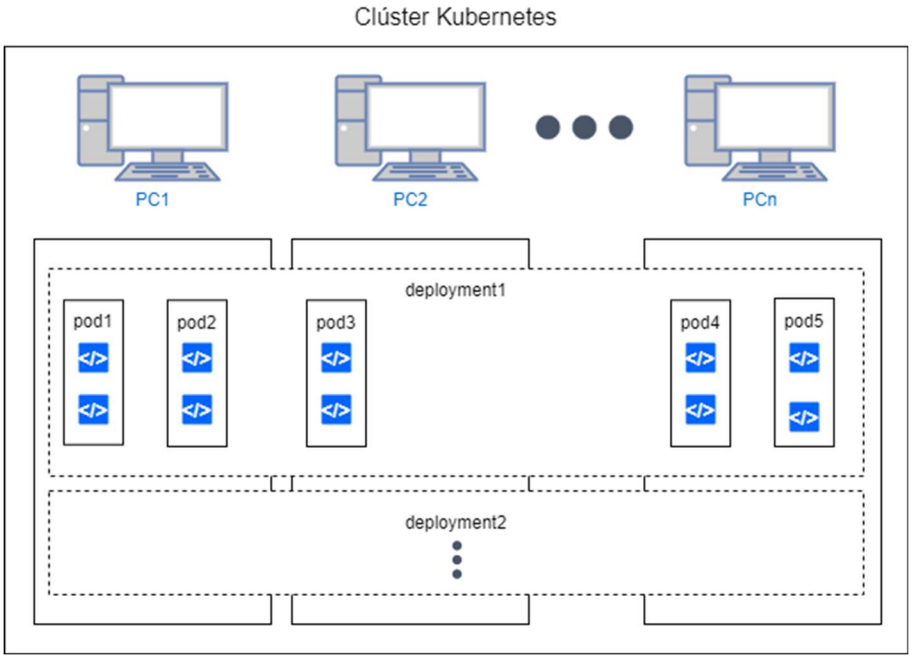

Figura 15. Abstracción de aplicaciones en Kubernetes

Como se ve en la imagen, los *pod* que hay en un *deployment* se distribuyen a lo largo de los diferentes nodos, repartiendo la cantidad que hay en cada uno de ellos. Los *pod* además son iguales, por lo que no hay diferencia entre los servicios que ofrecen.

Para entender la infraestructura de red de Kubernetes, tenemos que hablar de los *service*. Los *service* son elementos de Kubernetes que actúan como *proxy* para acceder a las aplicaciones que implementan los *pod*, alojándose en una dirección IP estática y no efímera. Cada *service* aporta las funcionalidades de balanceo de carga entre *pod*. En la *Figura 16. Servicios en Kubernetes* se muestra un ejemplo de esto.

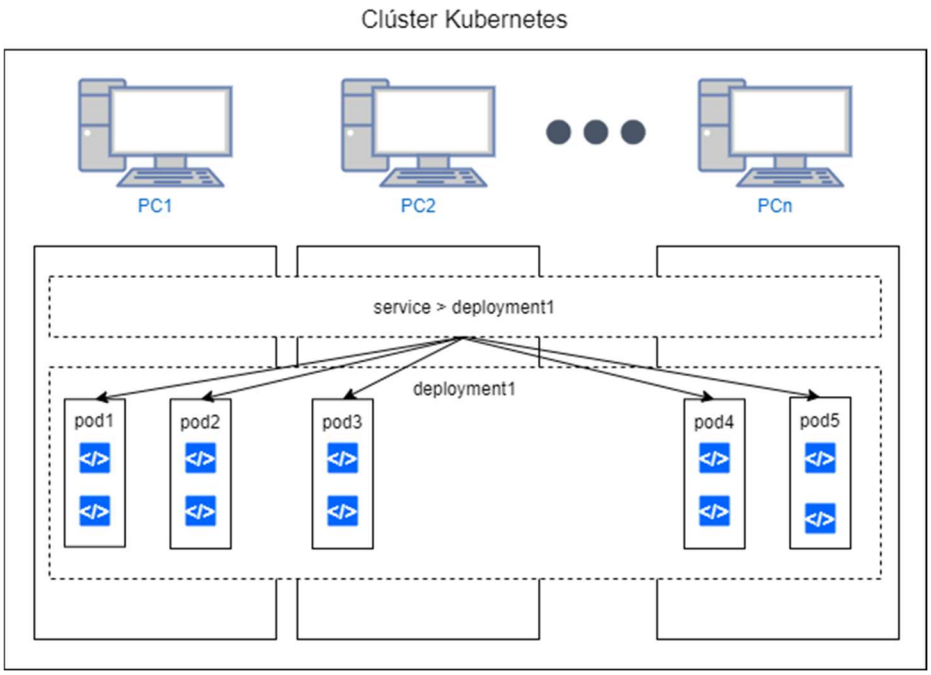

Figura 16. Servicios en Kubernetes

En la figura vemos que el *service* que distribuye la carga entre *pod* se encuentra distribuido en todos los nodos, de forma que cualquier petición, interna o externa al clúster pueda ser alcanzada por el *service*. En el caso de querer servir peticiones se hace uso de un puerto seleccionado de las interfaces externas de los nodos del clúster para recibir paquetes. Si se quieren servir peticiones provenientes del interior del clúster, Kubernetes implementa un servidor de nombres conocido por todos los *pod* y con una dirección IP estática, y de esta forma se puede localizar el *service* sencillamente sin conocer su IP mediante peticiones DNS.

Extendiéndonos más en los aspectos de encaminamiento de tráfico, podemos hablar de cómo se distribuyen estos elementos en el clúster: como ya hemos dicho, cada *pod* y cada *service* tienen una dirección IP, estática o no, y que sirve para el encaminamiento de tráfico entre elementos de una red virtual. De esta forma, podemos situar los elementos de la red virtual en las siguientes subredes:

- Cada *pod* tiene una dirección IP aleatoria de la subred 10.32.0.0/16, pero se priorizan los números más bajos.

- Cada *service* tiene una dirección IP de la subred 10.96.0.0/16, siendo seleccionable por el usuario. La única dirección de la subred que no se puede seleccionar es la 10.96.0.10, donde se aloja el servidor de nombre kube-dns.

Cada vez que queremos hacer una petición dirigida a un pod tenemos que escoger como dirección IP destino la del *service* (en caso de escribir el nombre de dominio asociado al *service* se acude al kube-dns para resolver la petición). Como todos los elementos del clúster se encuentran en una red privada compartida, la resolución de direcciones se hace a nivel de enlace, por lo que es necesario resolver la dirección MAC asociada a una dirección IP. Esto se hace mediante el protocolo ARP, tras lo cual los paquetes pueden llegar a los *service* correspondientes, que ya conociendo las direcciones IP y MAC de los *pod* a los que deben hacer llegar el tráfico, realizan las funciones de balanceo de carga. Este comportamiento puede servir para comprender el funcionamiento global del enrutamiento de paquetes en el clúster, pero tiene algunas peculiaridades debido al uso de *proxy* configurado por Kubernetes. Esto lo veremos en el *Capítulo 5. Pruebas de comunicación y carga de tráfico.* Por último, tenemos que hablar de un elemento que va a resultar importante en la orquestación de contenedores: el HPA (Horizontal Pod Autoscaler), que es un elemento que toma medidas de uso de los recursos hardware asignados a los *pod* de un *deployment* y los pasa al plano de control de Kubernetes para que este decida si se deben tomar acciones de escalado de la aplicación. La operación que utiliza Kubernetes para decidir si se deben crear o eliminar réplicas es la siguiente:

$$nR = \left\lceil nRA * \frac{mA}{mD} \right\rceil$$

Donde:

- nR es el número de réplicas de *pod* al cual se va a escalar.
- nRA es el número de réplicas de *pod* que existen actualmente.
- mA es la media de medidas de métricas de hardware tomada en nRA *pod*.
- mD es el valor de uso de hardware medio deseado en cada *pod*.

Una vez visto esto, ya hemos abordado todos los elementos importantes en nuestra implementación de aplicaciones en Kubernetes.

4.3. IMPLEMENTACIÓN

En este apartado vamos a hablar de la implementación de los objetos de Kubernetes que son necesarios para nuestra aplicación, diferenciando el escenario orquestado y sin orquestar.

4.3.1. Escenario sin orquestar

Empezando por el escenario sin orquestar, y teniendo claro la función de cada uno de los elementos que hemos descrito en el capítulo anterior, necesitaremos crear cuatro *pod* y cuatro *service*. Como hemos mencionado anteriormente, el *proxy* RTP no se va a encontrar contenerizado, por lo que cada *pod* implementará las funciones de SIP *prox*y in, SIP *proxy* out, route server y *softswitch*, respectivamente. Los ficheros de configuración de los contenedores que se encuentran en cada *pod* se detallan en el *Anexo C: Desarrollo del sistema*. Por lo general la configuración de estos es la misma que en el escenario anterior, pero hay que tener en cuenta los aspectos concretos de una implementación en Kubernetes, donde las direcciones IP de los *pod* no son estáticas y la comunicación entre elementos se realiza mediante *proxys*.

Por otra parte, los cuatro *service* que vamos a implementar permitirán la conexión entre los cuatro *pod* que implementamos. En el caso del *proxy* de entrada, tendremos que hacer uso de un *service* tipo "Nodeport", el cual escucha tráfico en la interfaz externa de los nodos del clúster y redirige el tráfico a un puerto del *pod*. En el resto de casos, usaremos un *service* de tipo "ClusterIP", que tienen una IP estática y un nombre de dominio conocido por un DNS para balancear la carga de tráfico entre los *pod*. En el caso del escenario sin orquestar, dicho balanceo de carga no existirá, pues solamente existe una réplica de cada *pod* para cada servicio. Aun así, es necesario crearlos ya que la IP de los *pod* no es estática. En la *Figura 17. Implementación en Kubernetes. Escenario sin orquestar* se muestra cómo quedaría este escenario en un clúster distribuido.

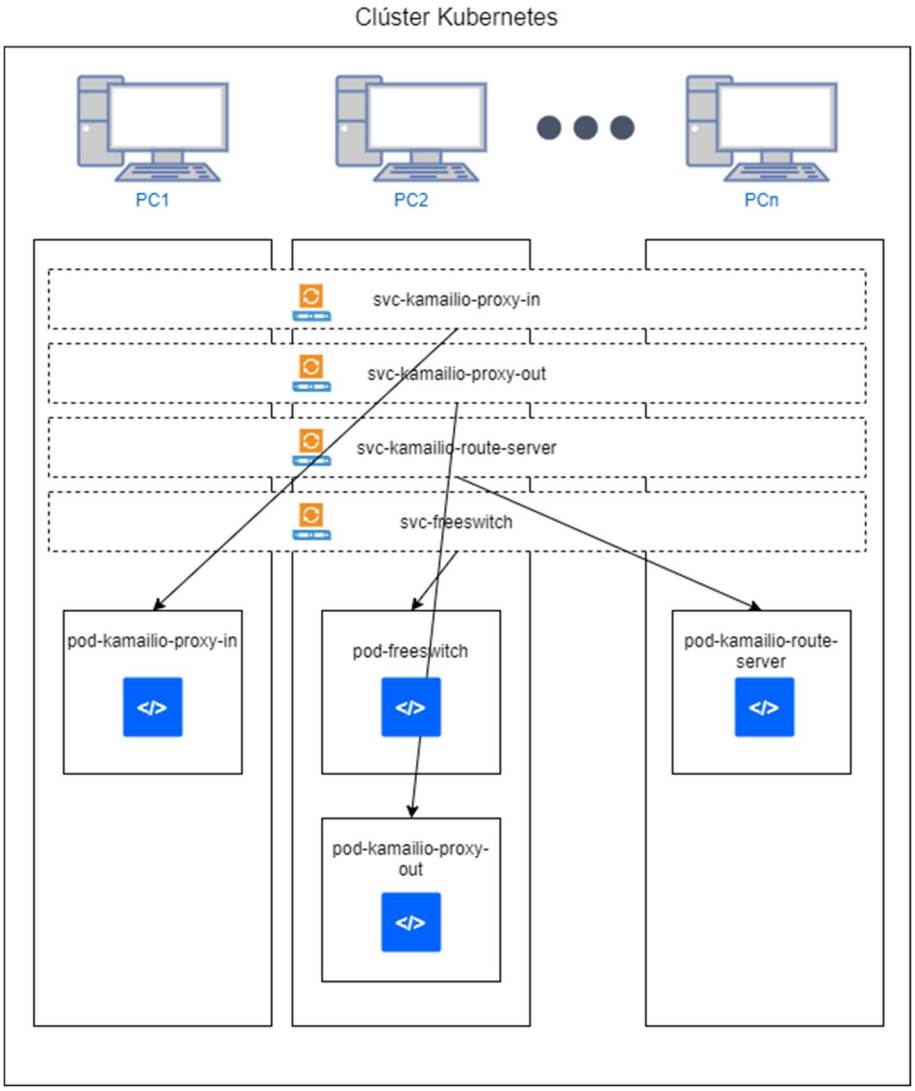

Figura 17. Implementación en Kubernetes. Escenario sin orquestar

En esta figura se observa que cada *pod* se puede encontrar en un equipo distinto, mientras que los servicios se implementan en cada uno de ellos, de forma que cualquier petición proveniente del clúster pueda ser servida.

4.3.2. Escenario orquestado

A continuación, pasaremos a observar el escenario orquestado. Para ello, necesitaremos: 4 *deployment*, 4 *service* y 4 HPA. Como hemos dicho anteriormente, los *pod* que implementa un *deployment* se encuentran distribuidos entre todos los equipos, y los *service* se encuentran simultáneamente en todos ellos para servir peticiones y funcionar como balanceador de carga sobre dichos *pod*. En este caso, además, necesitaremos 4 HPA, que se encarguen de tomar medidas de uso de recursos hardware asignados a los *pod* de un *deployment*, para de esta forma tomar decisiones sobre el escalado de los mismos. En la *Figura 18. Implementación en Kubernetes. Escenario orquestado* se observa el resultado de esta implementación.

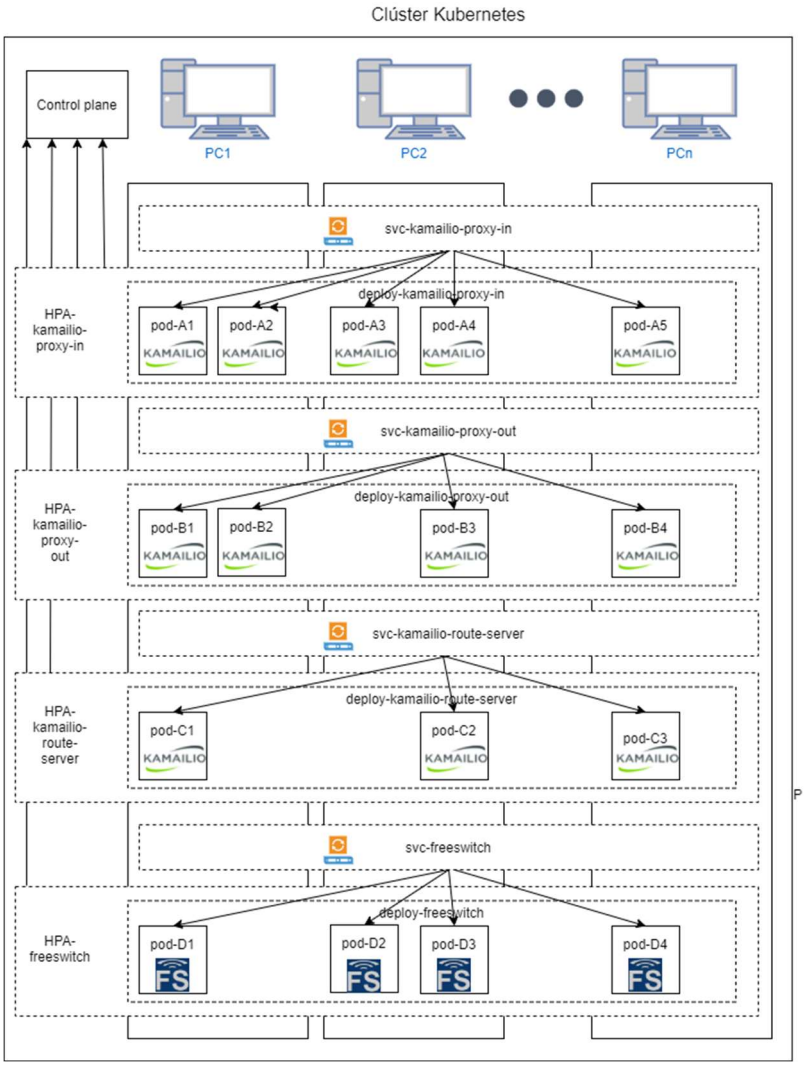

Figura 18. Implementación en Kubernetes. Escenario orquestado

En esta figura se observa, una vez más, cómo los *pod* de cada *deployment* (siendo los *pod* de cada *deployment* iguales entre sí, pero diferentes al de los *pod* de otros *deployment* en cuanto a las funciones que implementan) se encuentran distribuidos por los equipos de forma arbitraria y los *service* se encuentran en cada nodo. Adicionalmente, los HPA toman medidas de cada *pod*, por lo que deben encontrarse también en cada nodo donde haya *pod* del respectivo *deployment*.

4.4. INFRAESTRUCTURA Y COMUNICACIÓN DE POD

Vista la implementación que vamos a hacer en Kubernetes, pasaremos a tener una visión global de la arquitectura. Recordamos de capítulos anteriores que, además de los elementos virtualizados, contamos con un *proxy* RTP situado en un equipo físico, así como de los *softphones* llamantes y un proveedor de servicios que se encarga de terminar las llamadas. Todo esto se ve representado en la *Figura 19. Escenario con Kubernetes.* En ella, se ha representado cada *deployment* como un conjunto de tres *pod* iguales entre sí y diferenciados de los *pod* de otros *deployment*, pero como ya hemos dicho anteriormente, la cantidad de los mismos es arbitraria y se ajusta a las necesidades de procesamiento de cada instante mediante la orquestación.

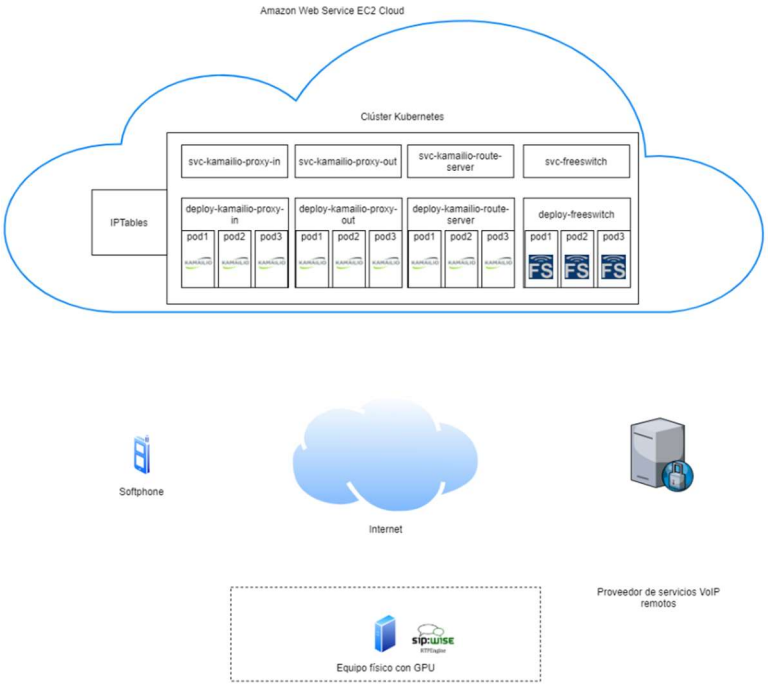

Figura 19. Escenario con Kubernetes

Dadas las restricciones que presenta Kubernetes para captar tráfico externo, el primer paso para conmutar una llamada es que los paquetes SIP correspondientes pasen por una regla de IPTables. El usuario normalmente hará la llamada sobre el puerto 5060, y esta regla la traducirá al puerto 30100, de forma que el *service* que hemos establecido para captar el tráfico externo y enviarlo al *proxy* de entrada reciba esos paquetes. Esto se ve representado en la *Figura 20. Escenario con Kubernetes. Regla* La definición de la regla que hemos creado se encuentra en el *Anexo I: Configuración de IPTables como DNAT.*

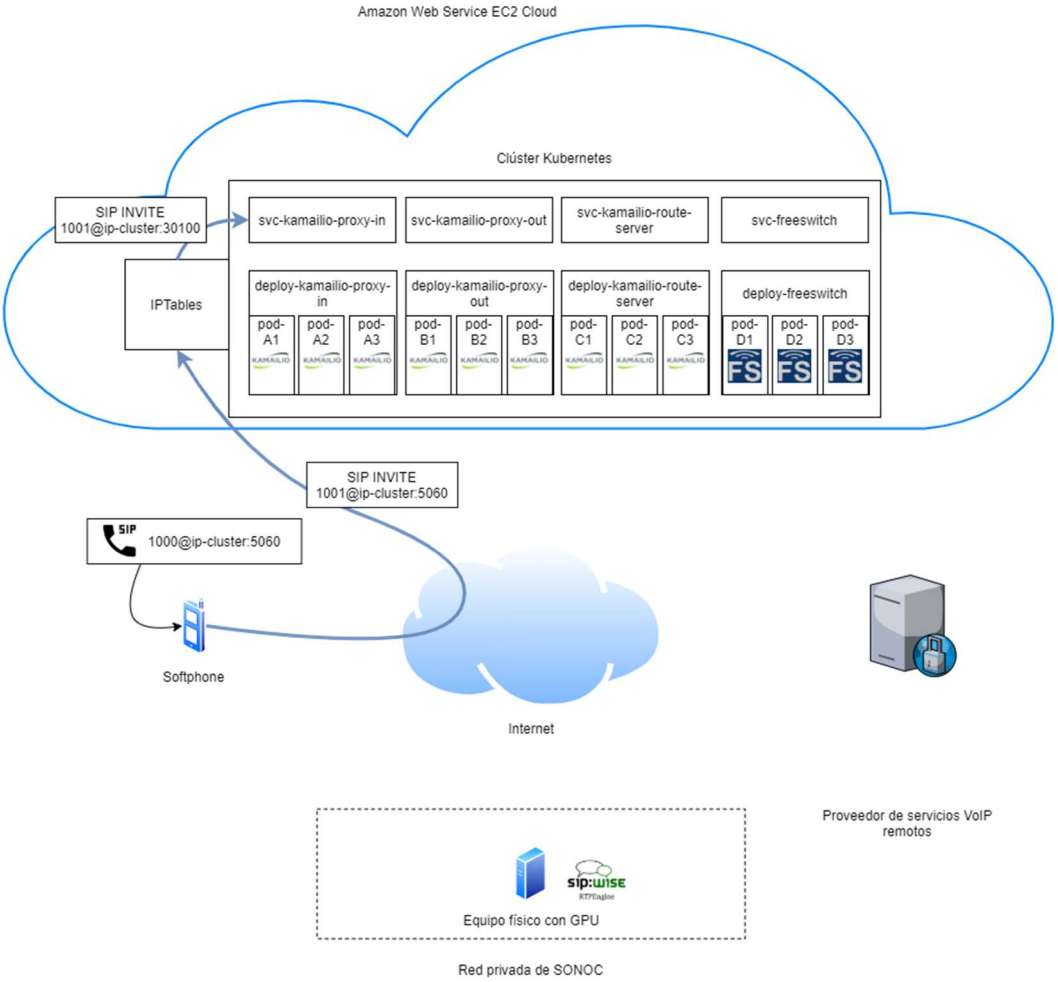

Figura 20. Escenario con Kubernetes. Regla de IPTables

La forma en la que se procesan los paquetes dentro del clúster a partir de este punto es similar al escenario sin contenedores, salvo por la existencia de los diversos *proxys* a los que

se envían inicialmente las llamadas y del reparto de carga que estos realizan. En las siguientes figuras se muestra este proceso, que es análogo a las de las *figuras 9, 10, 11, 12 y 13* que representaban el flujo de tráfico en el primer escenario. Además, en un escenario con un único nodo, las conexiones entre *pod* son locales, por lo que no salen a la red pública de Internet. En el caso de que los *pod,* que deben comunicarse entre sí, se encuentren en nodos diferentes, el contenido de los paquetes se encapsula y se envía a través de un puerto seguro (el 6443 de TCP por defecto). En las siguientes figuras, para mayor claridad vamos a suponer un único nodo.

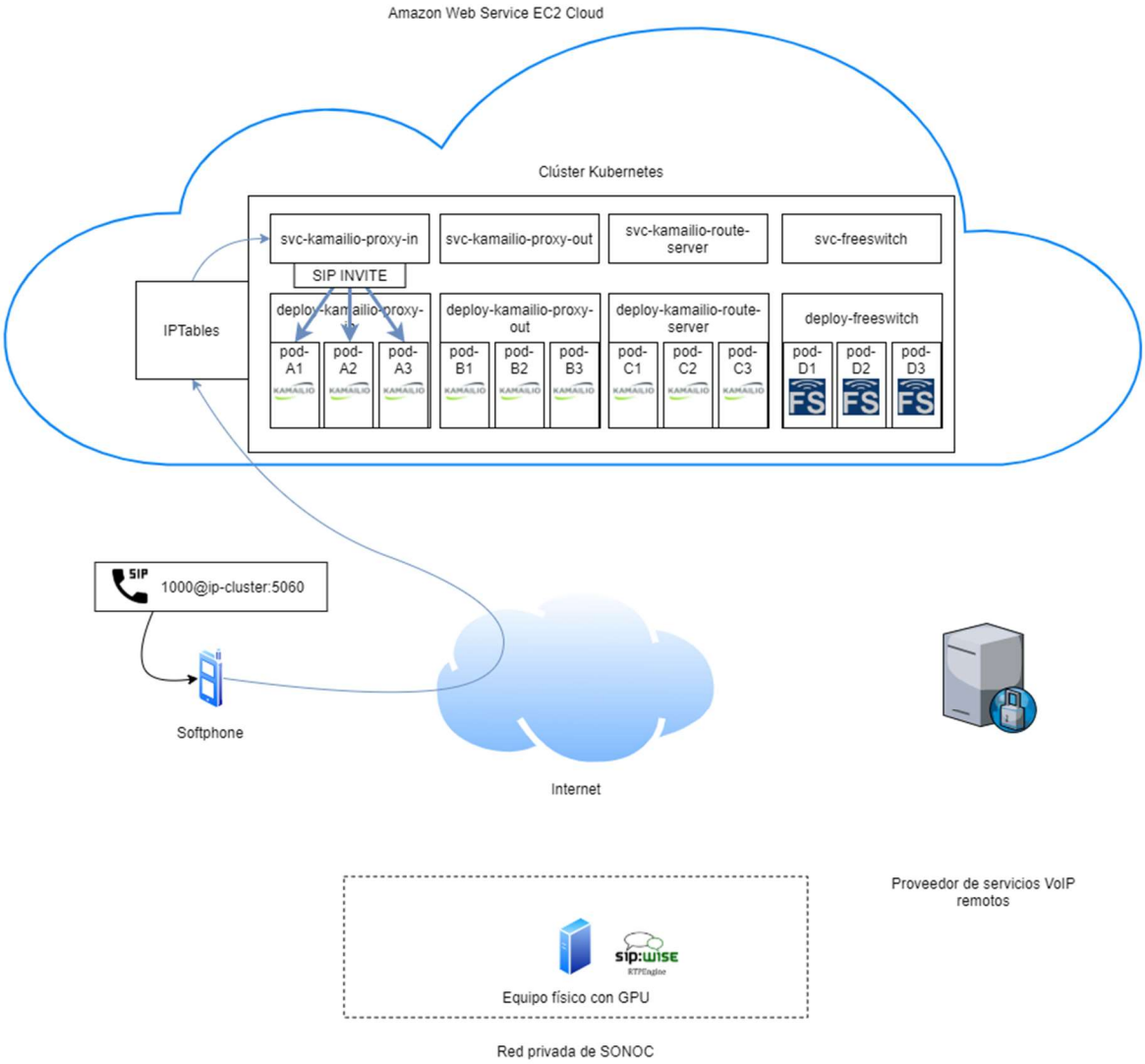

Figura 21. Escenario con Kubernetes. Envío de mensaje INVITE a proxy SIP

En esta figura el *service* que escucha en el puerto 30100 recibe las llamadas de los *softphones*, y distribuye la carga entre los *pod* que implementan los contenedores del *proxy* SIP de entrada, enviándolos al puerto 5060 de cada uno de ellos.

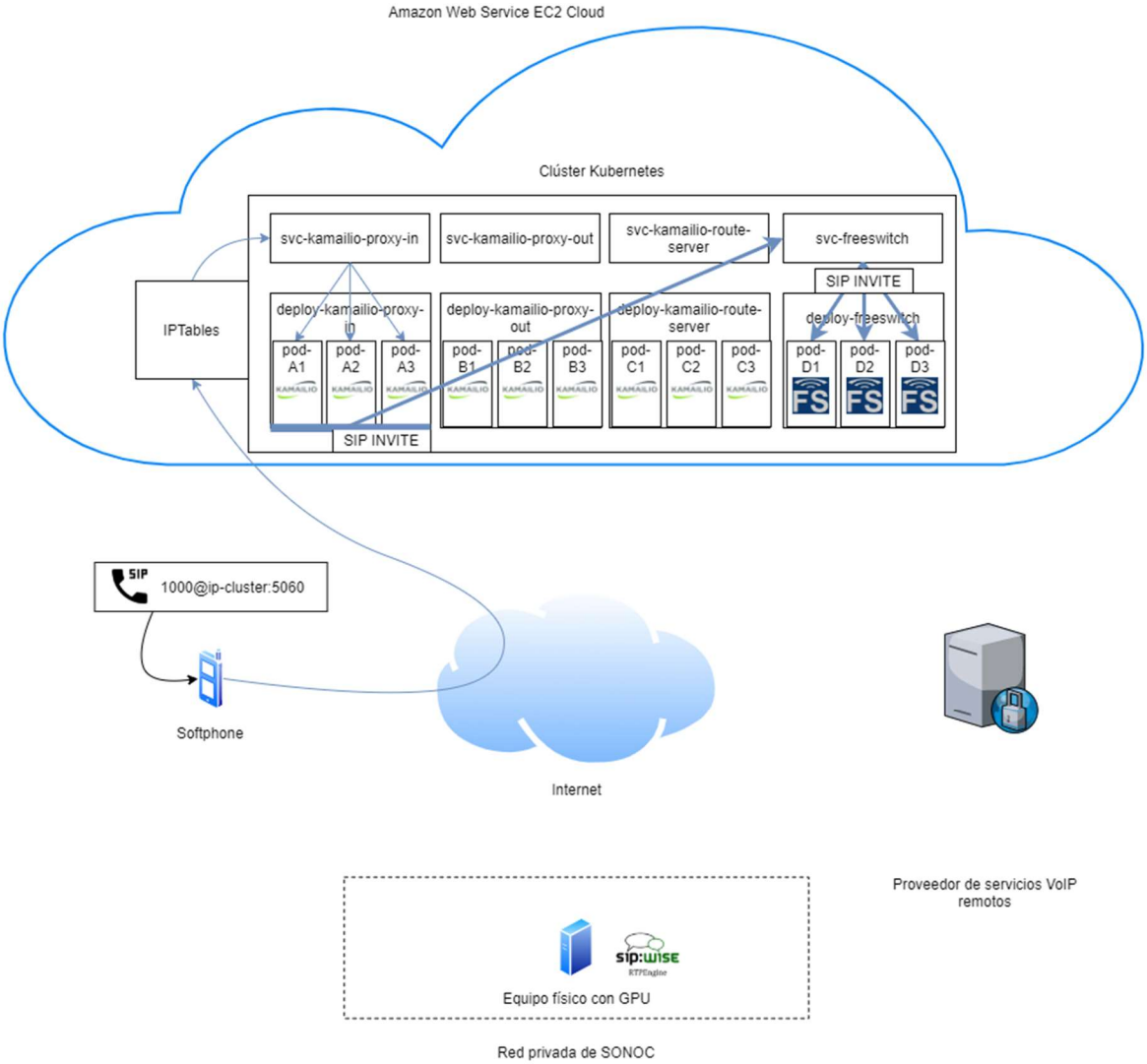

Figura 22. Escenario con Kubernetes. Reenvío de mensajes al softswitch

En esta figura el mensaje SIP INVITE recibido por los *proxys* SIP es reenviado al *service* del *softswitch*, tras lo cual se hace un reparto de carga entre los *pod* existentes.

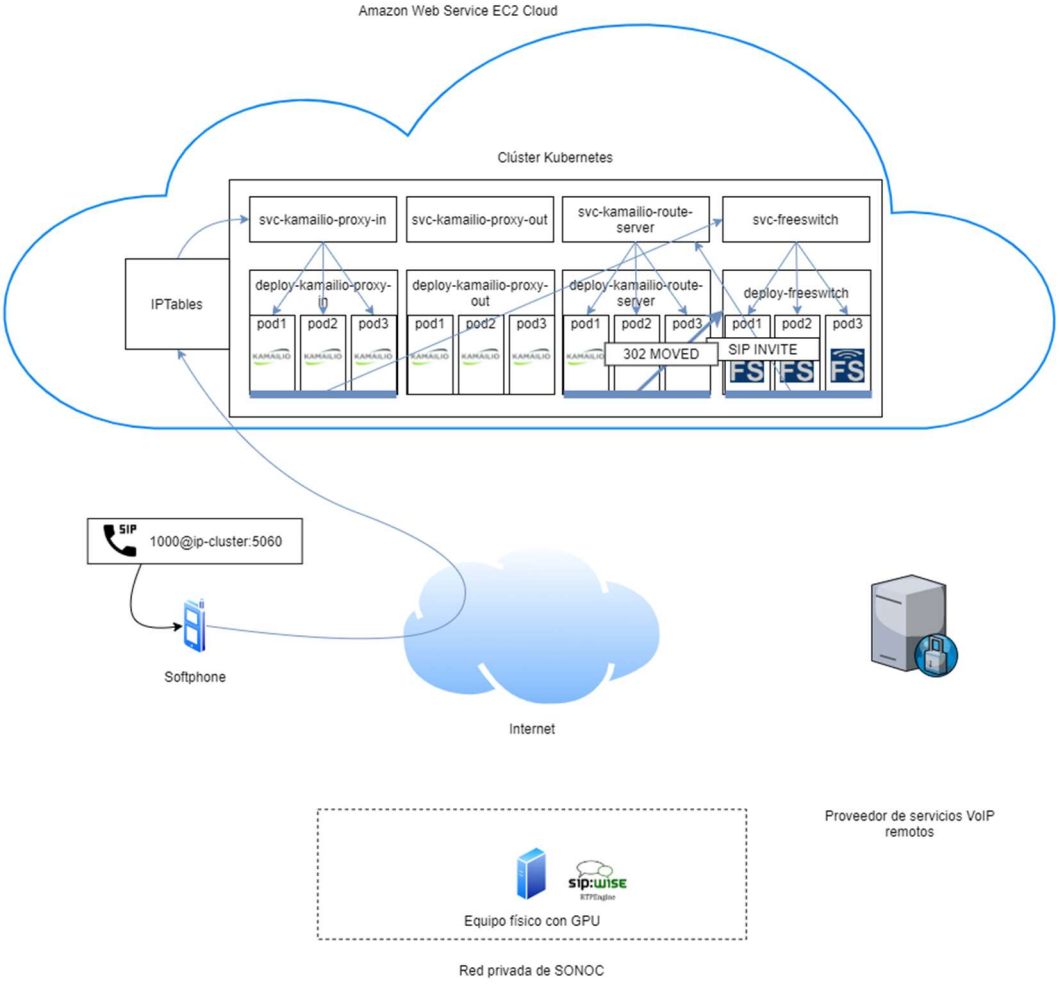

Figura 23. Escenario con Kubernetes. Petición de ruta

A continuación, se realiza la petición de ruta entre el *softswitch* y el route server. Es importante conocer que debido a la estructura de petición-respuesta que tiene el protocolo SIP, los mensajes de respuesta conocen la dirección IP del *pod* que ha pedido la ruta, por lo que estos no deben pasar por el *service*, sino que van directamente al *pod* que la ha pedido. Esto es así en todos los casos en los que se deban generar respuestas a paquetes.

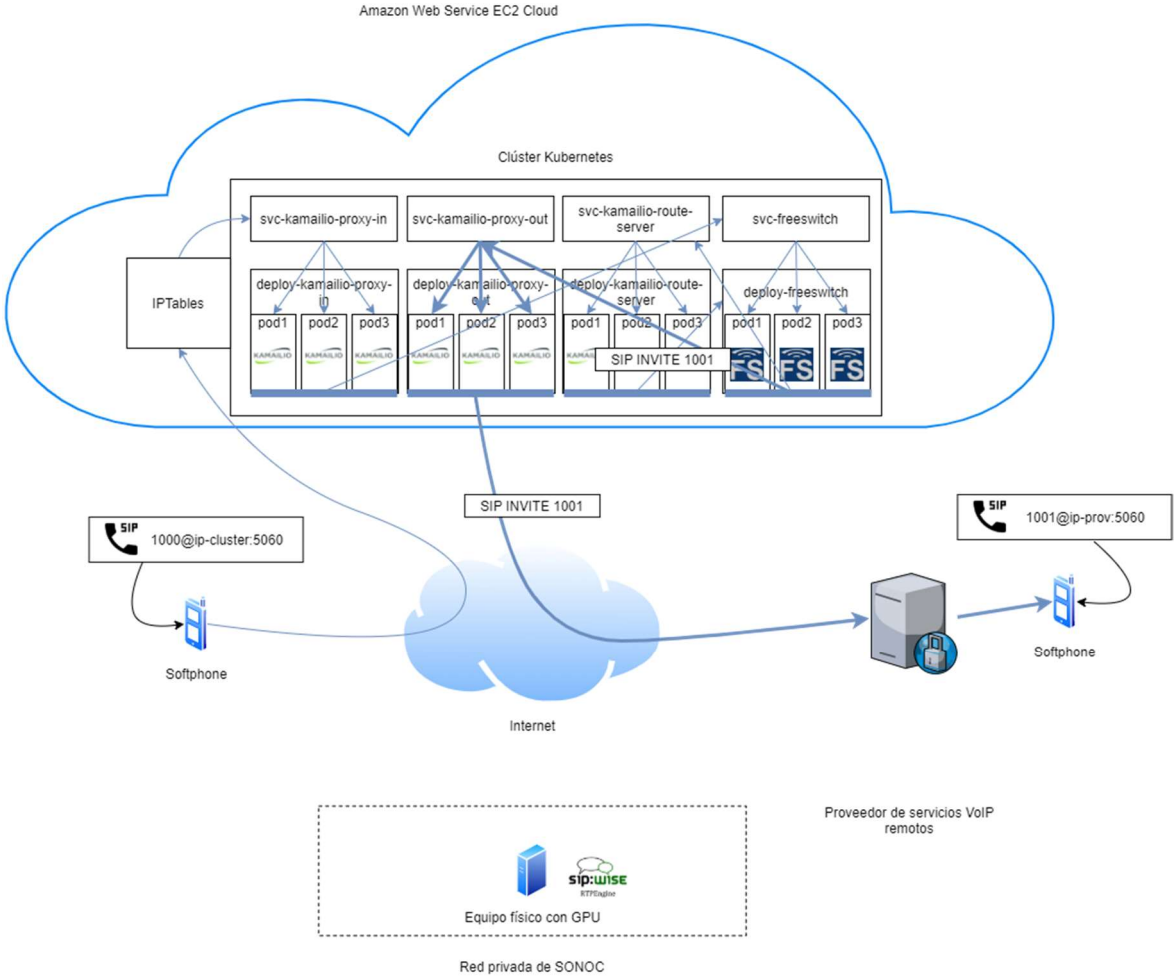

Figura 24. Escenario con Kubernetes. Finalización del establecimiento de llamada

Por último, una vez comunicada la ruta, el *softswitch* envía el mensaje INVITE al *proxy* de salida, que genera un paquete INVITE que sale del clúster con la dirección destino indicada en la cabecera, que se corresponderá con el equipo remoto que se encarga de terminar las llamadas.

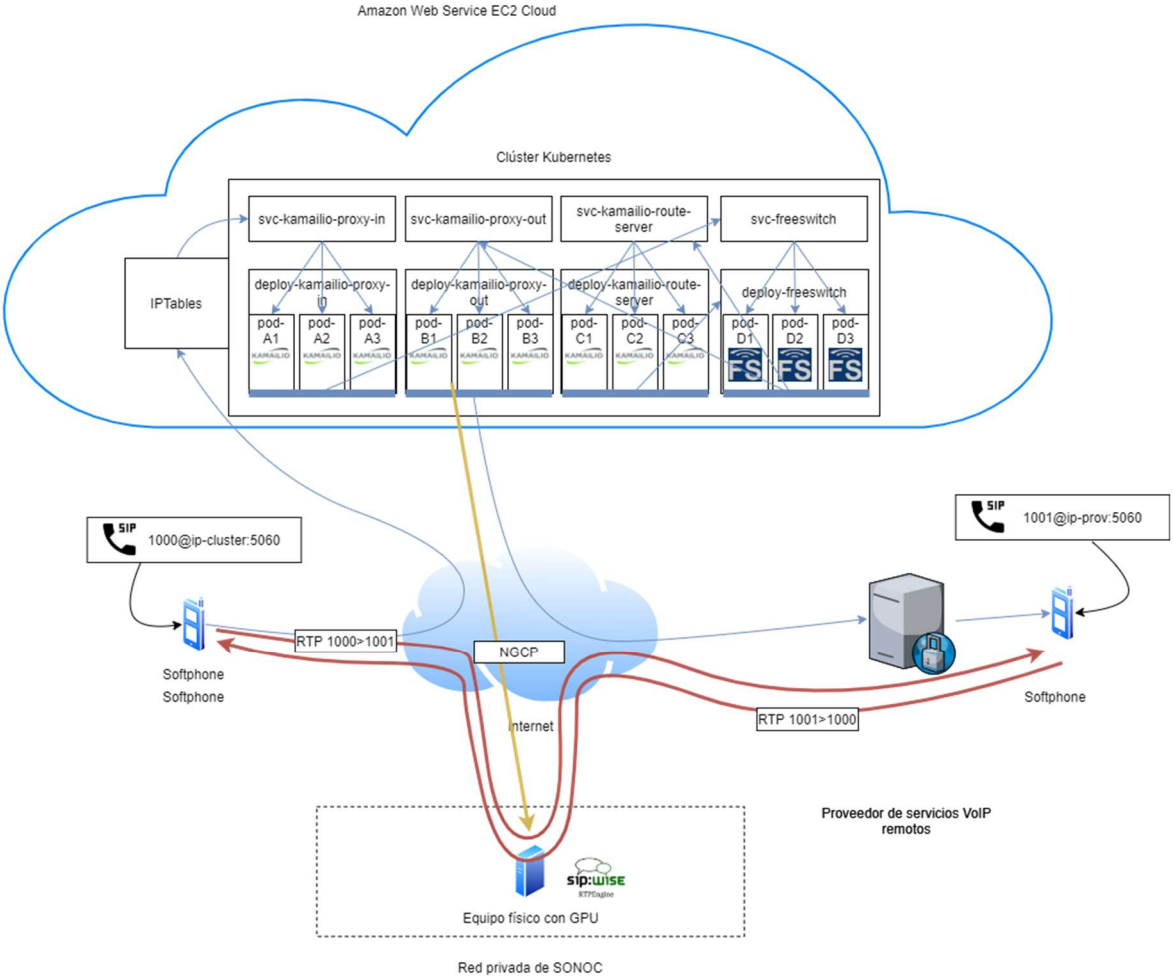

Figura 25. Escenario con Kubernetes. Establecimiento de conexiones RTP

Por último, el establecimiento de la conexión NGCP se realiza desde el *pod* que ha recibido la petición, y monitoriza la llamada en caso de recibir algún mensaje SIP informativo por parte de los extremos. El *proxy* RTP no es consciente de que este *proxy* se encuentra en un clúster de Kubernetes, y la forma en la que establece las conexiones RTP con los extremos es exactamente igual a los escenarios anteriores.

Tras haber explicado el flujo de mensajes esperado, pasaremos a hablar de la problemática principal que presenta este escenario y que abordaremos en mayor profundidad en el *Capítulo 5. Pruebas de comunicación y carga de tráfico*: la diferencia entre la señalización SIP sobre los protocolos UDP y TCP. La señalización SIP usada en llamadas VoIP comúnmente funciona sobre el protocolo UDP, pero debido a la naturaleza efímera de

los *pod*, no podemos garantizar la existencia de aquellos que están sirviendo una determinada llamada en el momento de su finalización. Esto implica que no se puede crear un registro de llamada y la conversación se quedaría "colgada", al haber flujos constantes RTP, pero no SIP que indiquen la finalización de la llamada. Esto puede ser solucionado haciendo uso del protocolo TCP, que establece conexiones entre los *pod*, impidiendo que estos sean eliminados mientras tengan conexiones activas correspondientes a las llamadas que cursan. Esto genera una nueva problemática: el objetivo de Kubernetes es que los recursos de hardware que se dedican al procesamiento de tráfico (en este caso de llamadas) se ajuste a las necesidades de cada instante. En el caso de uso de TCP para conexiones de larga duración podemos vernos en la situación en la que algún *pod* se encuentre activo debido a una llamada inusualmente larga. Esto genera una pérdida de eficiencia, pues estamos dedicando una cantidad de recursos alta al procesamiento de un tráfico ligero en comparación.

5. PRUEBAS DE COMUNICACIÓN Y CARGA DE TRÁFICO

En el siguiente capítulo realizaremos las pruebas de comunicación o de carga de tráfico pertinentes sobre cada uno de los escenarios que hemos desarrollado. Para establecer las comunicaciones haremos uso de las herramientas Zoiper y SIPp. En el *Anexo J: Pruebas de llamada con Zoiper* y el *Anexo K: Pruebas de llamadas con SIPp,* se encuentran los pasos seguidos para realizar dichas pruebas. En cada escenario indicaremos dónde se captura y cuáles han sido los resultados. La señalización SIP, salvo que se especifique lo contrario, se hará sobre UDP, a fin de que el escenario desarrollado sea lo más parecido posible a un escenario real.

Para capturar el tráfico usaremos la herramienta TCPDump, disponible en nuestras distribuciones de Linux y guardaremos la captura realizada como fichero .pcap, que posteriormente analizaremos haciendo uso de Wireshark. Un ejemplo de uso de TCPDump es:

- `tcpdump -s 0 -i <interface> -w <path_destino>`

Mediante los argumentos que pasamos para la ejecución de TCPDump estamos indicando lo siguiente:

- -s 0: Impide que el paquete capturado se trunque, es decir; que en la captura que realicemos se muestre la información de todos los protocolos que se encuentran encapsulados hasta el nivel de aplicación, que puede ser SIP, RTP o NGCP.
- -i <interface>: Indica la interfaz de la máquina en la cual se realiza la captura. En los escenarios desarrollados se utiliza la interfaz de red primaria de las máquinas virtuales y la interfaz virtual del clúster de Kubernetes.
- -w <path_destino>: Indica el directorio dónde se va a guardar el fichero .pcap que se genera a partir de la captura.

5.1. PRUEBAS DE COMUNICACIÓN: ESCENARIO 1

El primer escenario se corresponde con el desarrollo de la arquitectura virtualizada sin contenedores, visto en el *Capítulo 3. Implementación virtualizada sin contenedores.* Se ha utilizado Zoiper para la generación del tráfico de una única llamada.

Haremos capturas en las siguientes máquinas: *SIP Proxy In, softswitch, SIP Route Server y SIP Proxy Out*. Las comunicaciones entrantes y salientes de cada máquina utilizan la interfaz de red primaria (por lo general se llama eth0, pero ésta puede variar dependiendo de la configuración de red de las máquinas), y en ella hacemos las capturas. Si ordenamos los paquetes capturados en cada máquina, podemos elaborar un diagrama completo del intercambio de mensajes generado, el cual se muestra en la *Figura 26. Escenario 1. Intercambio de mensajes*, y que deberemos comparar con el generado en los escenarios posteriores.

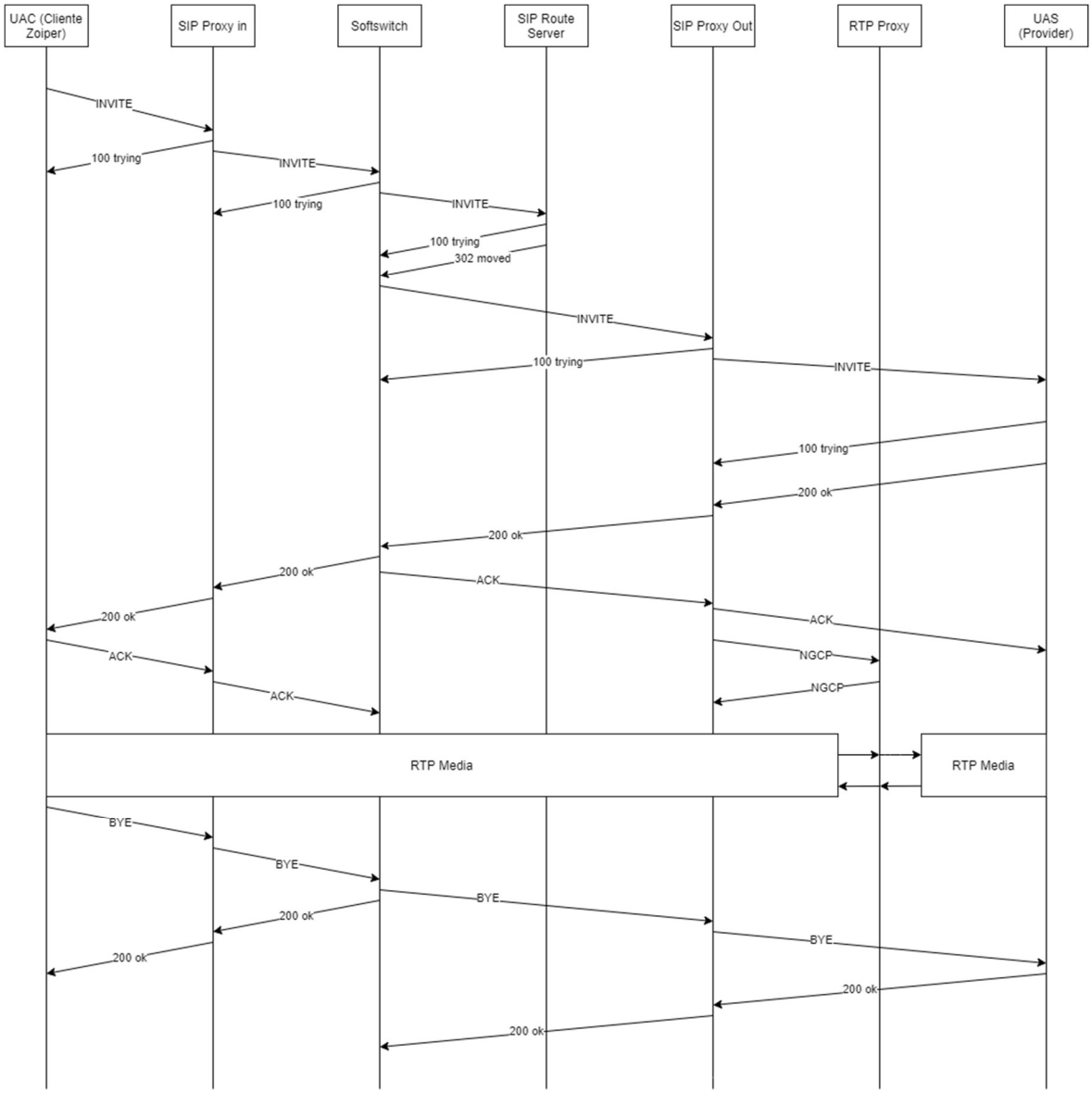

Figura 26. Escenario 1. Intercambio de mensajes

5.2. PRUEBAS DE COMUNICACIÓN Y CARGA DE TRÁFICO: ESCENARIO 2

El segundo escenario se corresponde con el desarrollo de la arquitectura virtualizada mediante contenedores en una red virtual de Kubernetes sin aplicar orquestación (visto en el capítulo *4.3.1. Escenario sin orquestar*. Se ha utilizado Zoiper para la prueba de comunicación que compararemos con el escenario anterior; y SIPp para analizar el volumen de tráfico que puede soportar un contenedor con unos recursos limitados, y que nos servirá para dimensionar las pruebas pertinentes del siguiente escenario.

Prueba de comunicación

En primer lugar, haremos la prueba de comunicación utilizando Zoiper. Para ello, deberemos diferenciar el tráfico generado sobre las aplicaciones que no se encuentran virtualizadas mediante contenedores, es decir, la de los usuarios finales (cliente Zoiper y proveedor remoto, que actúan como UAC y UAS respectivamente) y el proxy RTP; con el generado en las máquinas cuyas aplicaciones lo están: *SIP Proxy In, softswitch, SIP Route Server y SIP Proxy Out.* En el caso de aplicaciones no contenerizadas, el tráfico generado se producirá en las interfaces de red primarias de las máquinas (por lo general eth0), mientras que, en el caso de aplicaciones contenerizadas, dicho tráfico se generará en una interfaz virtual que depende de la configuración de red del clúster, en concreto del "Controlador de Políticas de Red" de Kubernetes. Dicho controlador maneja la red virtual mediante la cual intercambian mensajes los contenedores del clúster, así como los distintos mecanismos de descubrimiento de red. Por defecto, el controlador asignado a Kubernetes es *Weave Net*, cuya interfaz asociada se llama *weave,* y será dónde capturemos en el clúster.

Realizando una captura desde el cliente con Zoiper (que actúa como UAC de la llamada) sobre su interfaz de red primaria obtenemos la siguiente traza:

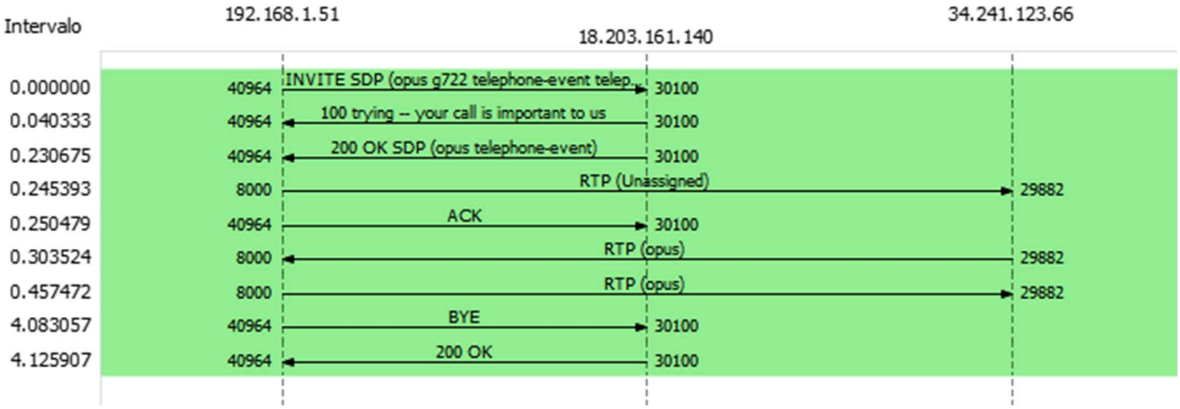

Figura 27. Escenario 2. Intercambio de mensajes. UAC

Como hemos dicho, en el clúster debemos capturar en la interfaz *weave*. Esta interfaz no es propia de un contenedor en concreto, sino que muestra todo el tráfico generado en el clúster. En esencia, la interfaz *weave* conecta virtualmente las interfaces de red propias de cada contenedor. Capturar sobre esta interfaz nos permite observar de forma conjunta el intercambio de mensajes generado, debiendo identificar todos los elementos que intervienen en cada diálogo SIP.

En las siguientes figuras se han identificado las distintas conexiones generadas a partir de una única llamada. Estas conexiones se corresponden con la división generada por la existencia de un B2BUA (que como hemos visto anteriormente, divide la comunicación entre dos usuarios, actuando como UA simulado que se comunica "back-to-back" y de forma independiente con los UA reales [ver *Anexo A: Protocolo de Iniciación de Sesión* para información más detallada al respecto]) y que hemos llamado "*tramo A*" y "*tramo B*" de la comunicación, y de la conexión entre el *softswitch* y el *SIP Route Server*, en la que se produce la petición de ruta.

Las trazas obtenidas se muestran en las siguientes figuras, las cuales se han ordenado en el orden lógico en el que las distintas conexiones se producen: en primer lugar, el *tramo A* de la comunicación entre el UAC y el *softswitch,* seguido de la petición de ruta, y por último el *tramo B* de la comunicación entre el *softswitch* y el proveedor.

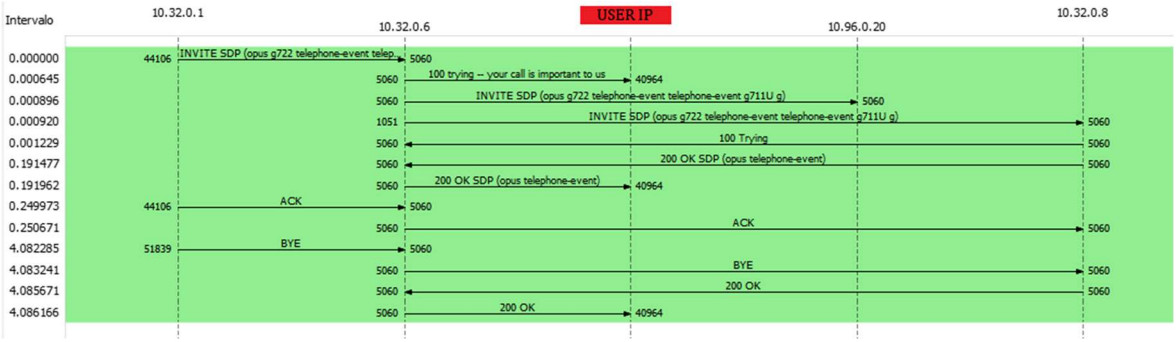

Figura 29. Escenario 2. Clúster tramo A

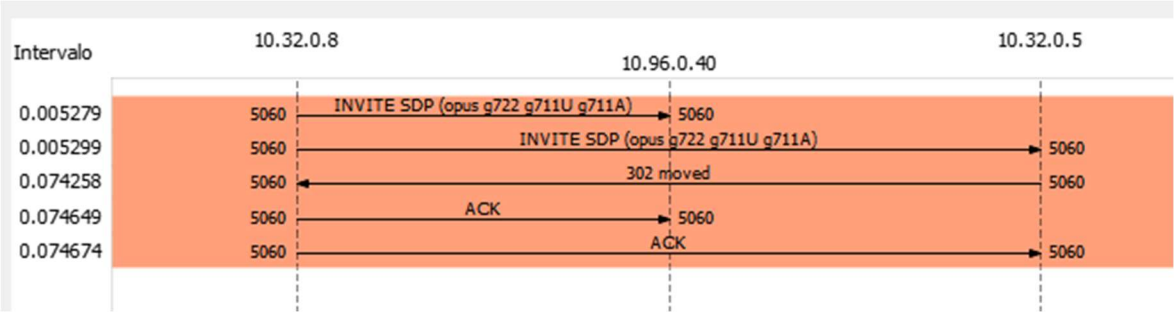

Figura 28. Escenario 2. Intercambio de mensajes. Petición de ruta

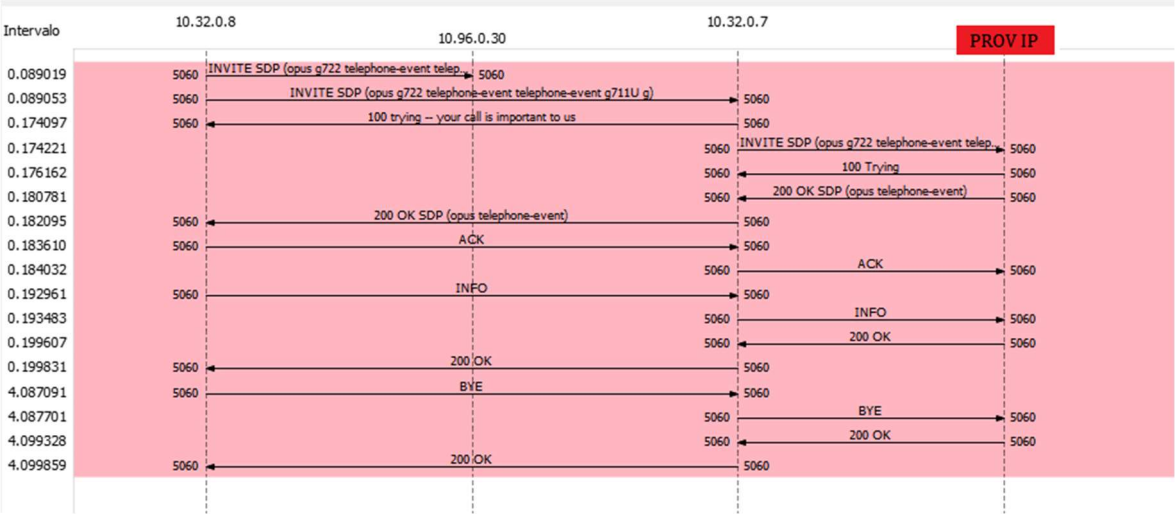

Figura 30. Escenario 2. Clúster tramo B

Para una mejor comprensión de las figuras identificamos las siguientes direcciones IP:

- USER IP y 192.168.1.51: IP del usuario que llama (pública y privada respectivamente).
- PROV IP: IP pública del proveedor de servicios remoto.
- 18.203.161.140: IP pública del clúster.
- 34.241.123.66: IP pública de RTPEngine.
- 10.32.0.1: IP interna del plano de control de Kubernetes que recibe las llamadas en el clúster, y que envía el tráfico al pod Kamailio proxy-in.
- 10.96.0.20, 10.96.0.30, 10.96.0.40: IP internas de los service de Kubernetes que conectan respectivamente con los pod: Freeswitch, Kamailio route-server y Kamailio proxy-out.
- 10.32.0.6, 10.32.0.8, 10.32.0.5, 10.32.0.7: IP internas asociadas de forma dinámicas a los pod: Kamailio proxy-in, Freeswitch, Kamailio route-server y Kamailio proxy-out respectivamente.

Hecha esta aclaración, hemos comprobado cómo el intercambio de mensajes es igual al del escenario anterior a excepción del tráfico generado por la existencia de *service,* que actúan como proxy. Dependiendo de la configuración establecida en kube-proxy se puede observar o no la existencia de los mismos, ya que no son equipos físicos reales (en el contexto del clúster), sino direcciones IP que son resueltas por los algoritmos establecidos en la configuración de kube-proxy (mediante IPTables o IPVS) para hacer llegar el tráfico a los *pod* correspondientes.

Prueba de carga

A continuación, pasamos a usar SIPp para observar el volumen de llamadas que es capaz de soportar el escenario antes de que se produzcan pérdidas de paquetes cuando la configuración de los *pod* establece la siguiente asignación de recursos:

- Memoria: mínimo de 0.5 GB y máximo de 1 GB
- CPU: mínimo de 200 milicores y máximo de 300 milicores

Usando el escenario desarrollado en los anexos, estableceremos una tasa de 1 nueva llamada por segundo, con una duración de 15 segundos cada una, e iremos aumentando dicha tasa hasta que observemos la existencia de errores. Los resultados obtenidos son los siguientes:

- Se obtienen retransmisiones que no afectan al establecimiento y mantenimiento de las llamadas con una tasa de generación de 4 llamadas por segundo.
- Se empiezan a perder paquetes puntuales por exceso de retransmisiones o timeout con una tasa de generación de 9 llamadas por segundo.
- Las pérdidas se empiezan a hacer notables e impiden el establecimiento normal de llamadas con una tasa de generación de 12 llamadas por segundo.

Debido a que los recursos asociados a cada elemento de nuestra arquitectura son iguales, y es sabido que Freeswitch consume una cantidad de recursos mayor que Kamailio (debido a aspectos como tener que mantener el estado de las conexiones activas), es aceptable asumir que dichos errores se producen en este elemento, debido a la falta de recursos CPU. En nuestro proyecto hemos usado una máquina de AWS t2.medium, con 2 CPU virtuales y 4 GB de memoria RAM, por lo que en un despliegue completo usando este tipo de máquinas, haciendo la división entre los recursos de CPU de los que hace uso cada pod (200 milicores o 0.2 CPU) y los recursos totales, el máximo de llamadas nuevas que calculamos de puedan servir será de aproximadamente 65 por segundo y "worker" (cada equipo del clúster de Kubernetes que ofrece sus recursos hardware para el despliegue de aplicaciones en contenedores). Este valor es una aproximación, ya que, en un escenario real, las comunicaciones activas tienen, por lo general, una mayor duración.

5.3. PRUEBAS DE CARGA DE TRÁFICO: ESCENARIO 3

El tercer escenario se corresponde con el desarrollo de la arquitectura virtualizada mediante contenedores en una red virtual de Kubernetes utilizado orquestación. Una vez comprobado que Kubernetes aplica las medidas de ajuste de escala de la aplicación, se pasará

a observar las diferencias observadas entre los ajustes de escala mediante señalización SIP por UDP y TCP. Para ello, haremos uso de SIPp.

En primer lugar, haremos una prueba utilizando una tasa de llamadas por segundo inferior al límite obtenido en el apartado anterior (9 llamadas por segundo), y veremos el grado de utilización de cada *pod* y cómo se crean nuevos debido a la demanda de recursos.

Para ver el grado de utilización utilizamos:

- `kubectl get hpa`

Y para ver el número de *pod* creados:

- `kubectl get pod`

Inicialmente nos encontramos en un estado igual al de la *Figura 31. Escenario 3. Estado inicial*, donde no se han creado más *pod* que las réplicas iniciales.

```
root@ip-10-3-0-120:/# kubectl get hpa
NAME                     REFERENCE                                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-freeswitch           Deployment/deploy-freeswitch             3%/70%     1         10        1          30m
hpa-kamailio-proxy-in    Deployment/deploy-kamailio-proxy-in      10%/80%    1         10        1          31m
hpa-kamailio-proxy-out   Deployment/deploy-kamailio-proxy-out     6%/80%     1         10        1          30m
hpa-kamailio-redirect    Deployment/deploy-kamailio-redirect      4%/80%     1         10        1          30m
```

Figura 31. Escenario 3. Estado inicial

Tras aplicar un tráfico constante de 8 llamadas por segundo, y como se ve en la *Figura 32. Escenario 3. Estado final*, se ha creado automáticamente una nueva réplica de Freeswitch, que es el *pod* que más recursos consume, lo cual corrobora lo explicado en el escenario anterior.

```
root@ip-10-3-0-120:/home/esanchez/pruebas# kubectl get hpa
NAME                     REFERENCE                                TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
hpa-freeswitch           Deployment/deploy-freeswitch             136%/70%    1         10        2          4m21s
hpa-kamailio-proxy-in    Deployment/deploy-kamailio-proxy-in      76%/80%     1         10        1          4m58s
hpa-kamailio-proxy-out   Deployment/deploy-kamailio-proxy-out     16%/80%     1         10        1          4m46s
hpa-kamailio-redirect    Deployment/deploy-kamailio-redirect      6%/80%      1         10        1          4m37s
```

Figura 32. Escenario 3. Estado final

5.3.1. Señalización SIP por UDP

Vamos a comprobar el mecanismo de cierre de *pod* de Kubernetes, para ello, abriremos dos llamadas servidas por dos *pod* distintos, y comprobaremos el cierre de éstos pasado un tiempo. El escenario usado será uno en el que la orquestación sólo se aplique sobre uno de los tipos de *pod*, para comprobar mejor los resultados. Para señalización UDP, como ya dijimos, no detectaría conexiones activas, por lo que se produce dicho cierre generando la pérdida de la comunicación. Para comprobar el nivel de escala de los *deployment* en tiempo real usamos:

- `kubectl describe deploy <my-deployment>`

En la *Figura 33. Escenario 3. Ajuste de escala con señalización UDP* observamos este comportamiento.

```
Pod Template:
  Labels:  app=pod-freeswitch
  Containers:
   cnt-freeswitch:
    Image:       esanchez:freeswitch
    Ports:       5060/UDP, 30100/UDP
    Host Ports:  0/UDP, 0/UDP
    Limits:
      cpu:     300m
      memory:  512Mi
    Requests:
      cpu:        200m
      memory:     256Mi
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   deploy-freeswitch-7bc65db7b7 (1/1 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  8m6s   deployment-controller  Scaled up replica set deploy-freeswitch-7bc65db7b7 to 1 <- Momento de creación del deployment
  Normal  ScalingReplicaSet  7m37s  deployment-controller  Scaled up replica set deploy-freeswitch-7bc65db7b7 to 2 <- Aplicación de tráfico intenso
  Normal  ScalingReplicaSet  13s    deployment-controller  Scaled down replica set deploy-freeswitch-7bc65db7b7 to 1 <- Tiempo después de reducir el nivel de carga
```

Figura 33. Escenario 3. Ajuste de escala con señalización UDP

En la sección “events” podemos ver en una línea de tiempo (cada línea tiene un valor “Age” asociado, que indica el tiempo pasado desde que se produjo el evento hasta la ejecución del comando que los muestra) cómo aumenta el nivel de escala del *deployment* (es decir, que aumenta el número de *pod* del mismo tipo), pero éste disminuye, tiempo después, debido a la bajada de tráfico que observa el clúster. Hay que precisar que, por defecto, las acciones que suponen bajar la escala tardan alrededor de unos 5 minutos en surtir efecto. Este valor puede ser modificado, pero conviene dejar un valor lo suficientemente grande como para que el sistema no responda inmediatamente a picos de tráfico.

5.3.2. Señalización SIP por TCP

Realizaremos la comparación del apartado anterior con el del uso de señalización TCP. El comportamiento del mismo se ve representado en la *Figura 34. Escenario 3. Ajuste de escala con señalización TCP.*

```
Pod Template:
  Labels:  app=pod-freeswitch
  Containers:
   cnt-freeswitch:
    Image:       esanchez:freeswitch
    Ports:       5060/TCP, 30100/TCP
    Host Ports:  0/TCP, 0/TCP
    Limits:
      cpu:     200m
      memory:  512Mi
    Requests:
      cpu:        100m
      memory:     256Mi
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   deploy-freeswitch-db9d4fb67 (2/2 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  9m56s  deployment-controller  Scaled up replica set deploy-freeswitch-db9d4fb67 to 1
  Normal  ScalingReplicaSet  8m8s   deployment-controller  Scaled up replica set deploy-freeswitch-db9d4fb67 to 2
```

Figura 34. Escenario 3. Ajuste de escala con señalización TCP

Como se puede observar, si bien Kubernetes aumenta el nivel de escala del *deployment*, no lo baja debido a que en todo momento hay conexiones activas. De forma nativa, Kubernetes tiene formas de sortear estos problemas, pero las conexiones inusualmente largas no escalan bien en este entorno.

Por último, mencionar que se ha observado que el sistema no es capaz de procesar tantas llamadas simultáneas, en comparación al escenario con señalización UDP antes de realizar medidas de ajuste ajustes de escala. Esto es comprensible debido a la necesidad de abrir conexiones TCP entre *pod* cada vez que llega una llamada, así como el procesamiento de las cabeceras adicionales.

6. CONCLUSIONES Y LÍNEAS FUTURAS

6.1. COMPARACIÓN Y CONCLUSIONES

Vamos a resumir lo aprendido en este Trabajo de Fin de Grado:

En primer lugar, se ha desarrollado una plataforma de comunicaciones VoIP completamente funcional, y para ello se ha hecho uso de las herramientas Kamailio, Freeswitch y RTPEngine, de las cuales hemos aprendido su funcionamiento básico y una configuración simple que permite la comunicación entre las mismas para ofrecer las funcionalidades básicas de un *switch*.

A continuación, se ha pasado a la implementación del escenario desarrollado en un entorno de Kubernetes. En primer lugar, esto ha afianzado los conocimientos obtenidos de las herramientas anteriormente mencionadas, pues hemos sido capaces de configurarlas en distintos entornos para ofrecer la misma funcionalidad. Por supuesto, el mayor conocimiento que hemos obtenido ha sido en el despliegue de aplicaciones en Kubernetes, para lo cual hemos realizado un estudio de los diferentes elementos de un clúster, los hemos valorado y hemos hecho uso de lo necesario para ofrecer las funcionalidades del escenario anterior. Esto también ha supuesto la comprensión del concepto de contenedor y de su desarrollo en el motor de Docker.

Por último, se ha aprendido a hacer uso de SIPp, una herramienta de simulación de tráfico SIP basada en escenarios XML. El desarrollo de los mismos implica el estudio del protocolo SIP, las comunicaciones entre usuarios y el significado de cada mensaje, así como del payload de estos.

Dadas las pruebas realizadas y las diferencias que se pueden observar entre los escenarios planteados, se concluye que la migración de un switch de comunicaciones VoIP a Kubernetes es posible, pero su transición no es directa, por lo que harían falta ciertos ajustes para asegurar la calidad de las llamadas. En este sentido, nos decantamos por una solución orquestada basada en TCP, que escala mucho mejor en el entorno de Kubernetes. Esto, como ya hemos dicho, provoca ciertos problemas en la asignación de recursos y la automatización

del ajuste de escala, pero es mucho más práctico que el uso de UDP, donde surgen problemas serios en un caso de aplicación real, con posibles pérdidas imprevistas de llamadas, dificultad para el registro de las mismas, etc. Todos estos problemas que plantea la señalización UDP se solucionan automáticamente en el traslado a TCP, debido a las diferencias en su comportamiento en el contexto del clúster.

También se han observado problemas para realizar balanceo de carga entre elementos del clúster, lo cual puede deberse a la arquitectura utilizada o a la naturaleza del tráfico generado. En todo caso, Kubernetes no escala muy bien las conexiones de larga duración, debido a que no existe un método completamente automatizado (sin herramientas externas) para trasladar las conexiones entre *pod,* que acaba en pérdidas de eficiencia por tener elementos del clúster con una cantidad de recursos asociada que no se equiparan con la requerida para procesar el tráfico que pasa por ellos, lo cual es otro aspecto sobre el que se tendría que trabajar para llevar el escenario estudiado a uno real.

6.2. LÍNEAS FUTURAS

Una vez realizado el análisis sobre los resultados obtenidos, se presentan las líneas de trabajo futuras que servirían como posibles mejoras al trabajo realizado:

- Integración del escenario desarrollado junto al resto de elementos planteados del *switch*, así como las funcionalidades de los elementos trabajados para que funcionen en ese nuevo entorno.
- Desarrollo e integración de herramientas propias de Kubernetes que solucionen los problemas que han surgido. Entre ellos se encuentra KEDA, que realiza ajustes de escala de aplicaciones mediante eventos,
- Despliegue de la aplicación usando las plataformas como EKS, AKS, GKE u OpenShift, que ofrecen las funcionalidades de un clúster de Kubernetes alojado en la nube y completamente gestionados por la plataforma.

BIBLIOGRAFÍA

[1] Documentación del protocolo SIP

- The Internet Society (2002, junio). RFC3261. https://datatracker.ietf.org/doc/html/rfc3261/
- Banerjee, K. (2005). [Tutorial aplicado sobre el uso del protocolo SIP]. http://www.siptutorial.net/SIP/index.html

[2] Documentación del protocolo RTP

- The Internet Society (2003, julio). RFC3550. https://datatracker.ietf.org/doc/html/rfc3550/

[3] Documentación del protocolo TCP

- Instituto de Ciencias de la Información de la Universidad de California del Sur. (1981, septiembre). RFC793. https://datatracker.ietf.org/doc/html/rfc793/

[4] Documentación del protocolo UDP

- Postel, J. (1980, 28 de agosto). RFC768. https://datatracker.ietf.org/doc/html/rfc768/

[5] Documentación de Kamailio

- The Kamailio SIP Server Project. (s.f.). [Documentación general sobre la instalación y uso de Kamailio]. https://www.kamailio.org/w/documentation/
- The Kamailio SIP Server Project. (s.f.). [Archivo público de correos electrónicos de usuarios de Kamailio en el que se describen problemas conocidos]. https://lists.kamailio.org/
- The Kamailio SIP Server Project. (2017, 19 de octubre). [Repositorio de Kamailio en Dockerhub] https://hub.docker.com/r/kamailio/kamailio/

[6] Documentación de Freeswitch

- Varios contribuidores. (s.f.). [Documentación general sobre la instalación y uso de Freeswitch] https://freeswitch.org/confluence/
- Varios contribuidores. (s.f.). [Documentación de módulos de la API de Freeswitch] https://docs.freeswitch.org/
- Varios contribuidores. (s.f.). [Archivo público de correos electrónicos de usuarios de Freeswitch en el que se describen problemas conocidos]. https://lists.freeswitch.org/

- Better Voice. (2015, 14 de noviembre). [Repositorio de Freeswitch en Dockerhub] https://hub.docker.com/r/bettervoice/freeswitch-container/

[7] Documentación de RTPEngine

- Sipwise. (2007, 29 de junio). [Repositorio en Github con información sobre la instalación y uso de RTPEngine]. https://github.com/sipwise/rtpengine

[8] Documentación de Docker

- Docker, Inc. (s.f.). [Documentación general sobre la instalación y uso de Docker y Docker Hub]. https://docs.docker.com/

[9] Documentación de Kubernetes

- Google, LLC. (s.f.). [Documentación general sobre la instalación y uso de Kubernetes]. https://kubernetes.io/docs/home/
- Kubernetes SIGs. (2017, 24 de mayo). [Repositorio de Github sobre la instalación y uso de servidores de métricas en Kubernetes]. https://github.com/kubernetes-sigs/metrics-server/
- Google, LLC. (s.f.). [Ejemplo práctico para el ajuste de escala automático de pod en Kubernetes]. https://cloud.google.com/kubernetes-engine/docs/how-to/horizontal-pod-autoscaling?hl=es-419
- Docker, Inc. (s.f.). [Ejemplo de despliegue de aplicaciones de Docker en Kubernetes]. https://docs.docker.com/get-started/kube-deploy/

[10] Documentación de SIPp

- Gayraud, R., Jacques, O. y otros contribuidores. (2004). [Documentación general sobre la instalación y uso de SIPp]. http://sipp.sourceforge.net/doc3.3/reference.html
- Bertera, P. (2018, 25 de mayo). [Repositorio en Github con ejemplo de uso de escenarios XML para SIPp]. https://github.com/pbertera/SIPp-by-example/