Universidad Zaragoza

Trabajo Fin de Grado

Distribuidor de carga para SMS GW
Load balancer for SMS GW

Autor
Óscar Palacín Grasa

Director
Fernando Orús Morlans

Ponente
Julián Fernández Navajas

Escuela de Ingeniería y Arquitectura
2023

Distribuidor de carga para SMS GW

Resumen

IRIS es la solución para mensajería A2P (Application To Person) de la plataforma SONOC TECHNOLOGY sobre protocolo SMPP v3.4 (https://smpp.org). A2P es un servicio de gran demanda en la actualidad, debido a nuevas regulaciones de seguridad en pagos electrónicos y aplicaciones bancarias y al incremento de las notificaciones vía SMS en todo tipo de aplicaciones. Un SMS Gateway es un sistema que permite enviar y recibir mensajes de texto (SMS) entre dispositivos móviles y aplicaciones a través de una red de telecomunicaciones. Actúa como un intermediario entre las aplicaciones y los operadores de telecomunicaciones. Dentro de la arquitectura de IRIS el SMS gateway se conoce como IRISGW.

IRISGW proporciona una interfaz HTTP para recibir mensajes SMS de clientes y redirigirlos a los proveedores capaces de enviarlos a los terminales de los destinatarios finales. Se propone anteponer a IRISGW un balanceador de carga (IRISLB). Este módulo permitirá desacoplar la lógica interna IRIS relacionada con la facturación, enrutamiento, etc… de la gestión de las sesiones con los clientes y la recepción y envío de mensajes.

Para solucionar las problemáticas en cuanto a fiabilidad (si se presenta un único punto de fallo) y escalabilidad (cuando se alcanza un número determinado de mensajes por segundo y la plataforma llega a su límite de capacidad), se propone desplegar IRISLB dentro de un grupo elástico de instancias en clúster Kubernetes.

El protocolo SMPP es el utilizado por las aplicaciones para enviar y recibir mensajes SMS desde y hacia dispositivos móviles. SMPP requiere del establecimiento de una conexión (BIND) entre las dos partes antes de poder enviar un mensaje. IRISLB debe mantener esas asociaciones hasta el envío con éxito de los mensajes de manera transparente. El protocolo SMPP (Short Message Peer-to-Peer) es un protocolo abierto diseñado para proporcionar una interfaz de comunicaciones de datos flexible para la transferencia de datos de mensajes cortos entre entidades externas de mensajes cortos (ESME), entidades de enrutamiento (RE) y centros de mensajes (MC).
Las funciones principales de IRISLB serían:
-Ingress: Interfaz única para los clientes de la plataforma.
-Gestión sesiones SMPP: Autenticación y control de las sesiones establecidas por clientes.
-Adaptación entre protocolos SMPP y HTTP.
Este trabajo se desarrollará en un contexto de colaboración con SONOC (www.sonoc.io/) una empresa líder del sector de las telecomunicaciones internacionales.

Índice de contenidos

Índice de figuras

Glosario de términos

CPU. Central Processing Unit
HTTP. Hypertext Transfer Protocol
JSON. JavaScript Object Notation
IP. Internet Protocol
SONOC. System One Noc & Development Solutions, S.A.
TCP. Transport Control Protocol
TLS. Transport Layer Security
YAML. YAML Ain't Markup Language
A2P. Application to person
P2P. Person to person
jSMPP: Mobile messaging for JAVA
GW. Gateway
API. Application programming interface
SMPP. Short Message Peer-to-Peer
SMS. Short Message Service
ESME. External Short Message Entities
SMSC. Short Message Service Center
RE. Routing Entities
MS. Message Centres
PDU. Protocol Data Unit
Wholesale. Al por mayor
Retail. Al por menor
Tenant. Inquilino (Identificador de un cliente de plataforma en los recursos compartidos de la nube privada de SONOC)
Carrier. Operador
Bind. Enlace
DLR. Delivery Receipt

1. Introducción

1.1 Introducción y motivación

La tecnología de SMS (Short Message Service) es un servicio disponible en los teléfonos móviles que permite el envío de mensajes cortos (con un límite de caracteres) entre teléfonos móviles a través de una amplia variedad de redes, incluidas las redes 4G. Aprovechando el éxito de SMS, como se ve en [BAN], "GSMA (Global System for Mobile Communications Association) ha intentado fomentar múltiples tecnologías para el envío de mensajes MMS (Multimedia Message Service) que no han conseguido imponerse en el mercado". Sin embargo, SMS sigue teniendo demanda gracias a su ubicuidad e interoperabilidad (SMS es una tecnología universal que se puede utilizar en cualquier red

móvil y entre diferentes operadores. Los mensajes SMS no requieren una conexión a Internet y se pueden enviar y recibir de manera rápida).

Por otro lado, la principal competencia en la actualidad son las redes over the top (OTT) que son aquellas que utilizan Internet para ofrecer servicios de transmisión de contenido de audio, video y datos directamente al consumidor final, sin necesidad de intermediarios tradicionales como las compañías de cable o las estaciones de televisión. Algunos de los ejemplos más conocidos son la plataforma de streaming para series Netflix y el servicio de streaming de música spotify.

Las redes OTT tienen un gran problema: no son interoperables, puesto que no son capaces de comunicarse e intercambiar información con otras redes semejantes, lo que se debe principalmente a factores como el control del servicio, la competencia y diferenciación, los modelos de negocio y los acuerdos comerciales. Las redes OTT son operadas por empresas o proveedores de servicios específicos que tienen control total sobre la plataforma y el contenido ofrecido. Cada proveedor de servicios OTT tiene sus propias aplicaciones, interfaces y estándares técnicos, lo que dificulta la interoperabilidad con otros servicios. Las empresas de servicios OTT compiten entre sí para atraer y retener a los usuarios. Para diferenciarse de la competencia, desarrollan características y funciones únicas en sus aplicaciones y plataformas. Por ello no se ha demostrado que haya un mercado para funcionalidades interoperables más avanzadas.

1.1.1 SMS A2P

SMS A2P (Application-to-Person) se refiere al envío de mensajes SMS desde una aplicación o plataforma automatizada a un usuario o receptor final. En este caso, "aplicación" puede referirse a una amplia gama de servicios o sistemas automatizados, como notificaciones de servicios, alertas, recordatorios, confirmaciones de transacciones, códigos de verificación, entre otros. El servicio A2P ha experimentado una alta demanda debido a las nuevas regulaciones de seguridad en pagos electrónicos y aplicaciones bancarias, así como al aumento en el uso de notificaciones vía SMS en diversas aplicaciones.

A diferencia de los mensajes P2P (Person-to-Person), que son enviados y recibidos directamente entre usuarios individuales, los mensajes SMS A2P son generados por sistemas o aplicaciones con el propósito de ser entregados a múltiples usuarios o destinatarios.
El uso de SMS A2P es común en diversas industrias y sectores, como servicios financieros, comercio electrónico, atención al cliente, servicios de entrega y logística, salud, entretenimiento, entre otros. Estos mensajes A2P son enviados a través de pasarelas de mensajería, que son plataformas de software que permiten la conexión entre la aplicación o sistema automatizado y los operadores de telefonía móvil, facilitando la entrega de los mensajes de texto. Las empresas y proveedores de servicios suelen utilizar servicios de mensajería A2P para gestionar y enviar estos mensajes de manera eficiente y masiva.

1.1.2 Wholesale SMS (A2P)

Los wholesale carriers son compañías que ofrecen servicios de telecomunicaciones a otras empresas, en lugar de a consumidores finales. Estas empresas compran capacidad de transmisión de datos a proveedores de redes y luego venden esa capacidad a otras empresas que necesitan grandes volúmenes de tráfico de datos. Los wholesale carriers desempeñan un papel importante en la industria de las telecomunicaciones al facilitar la conectividad entre redes y asegurar que haya suficiente capacidad para satisfacer la demanda de los consumidores finales.

Wholesale SMS A2P se refiere al proceso de enviar mensajes SMS a través de un proveedor mayorista. Los proveedores mayoristas proporcionan servicios de envío de mensajes SMS a gran escala a empresas y organizaciones. Estos proveedores mayoristas a menudo tienen conexiones directas con operadores de redes móviles en todo el mundo, lo que les permite enviar mensajes SMS a través de múltiples redes y países. El envío de mensajes SMS a través de proveedores mayoristas a menudo es más rentable que enviar mensajes individualmente a través de las redes móviles, lo que lo hace una opción popular para empresas que necesitan enviar grandes volúmenes de mensajes.

Wholesale SMS es un modelo de negocio en el que una empresa compra grandes cantidades de mensajes SMS a un proveedor y luego revende esos mensajes a sus propios clientes a precios más bajos que el que ofrecen directamente las operadoras. Este modelo se utiliza comúnmente por empresas que ofrecen servicios de SMS a sus clientes, como proveedores de servicios móviles, proveedores de software de mensajería, proveedores de servicios de SMS y otros intermediarios. La compra de mensajes SMS a granel permite a las empresas obtener precios más bajos por mensaje, lo que les permite ofrecer precios competitivos a sus clientes y aumentar su margen de beneficio. Además, las empresas pueden personalizar los mensajes SMS que envían a sus clientes y gestionar el envío de manera más eficiente.

El modelo de negocio de Wholesale SMS también es beneficioso para los proveedores de SMS, ya que les permite vender grandes cantidades de mensajes SMS a un solo cliente, reduciendo sus costos de administración y aumentando su volumen de ventas.

SONOC, la empresa con la que se colabora en el desarrollo del presente TFG, es líder en el mercado de software para operadores mayoristas de telefonía (wholesale carriers). En este contexto, SONOC desarrolla su plataforma IRIS como solución para gestionar mensajería A2P al por mayor sobre protocolo SMPP (Short Message Peer-to-Peer). En este esquema se observa el funcionamiento de envío de mensajes a través de la plataforma IRIS.

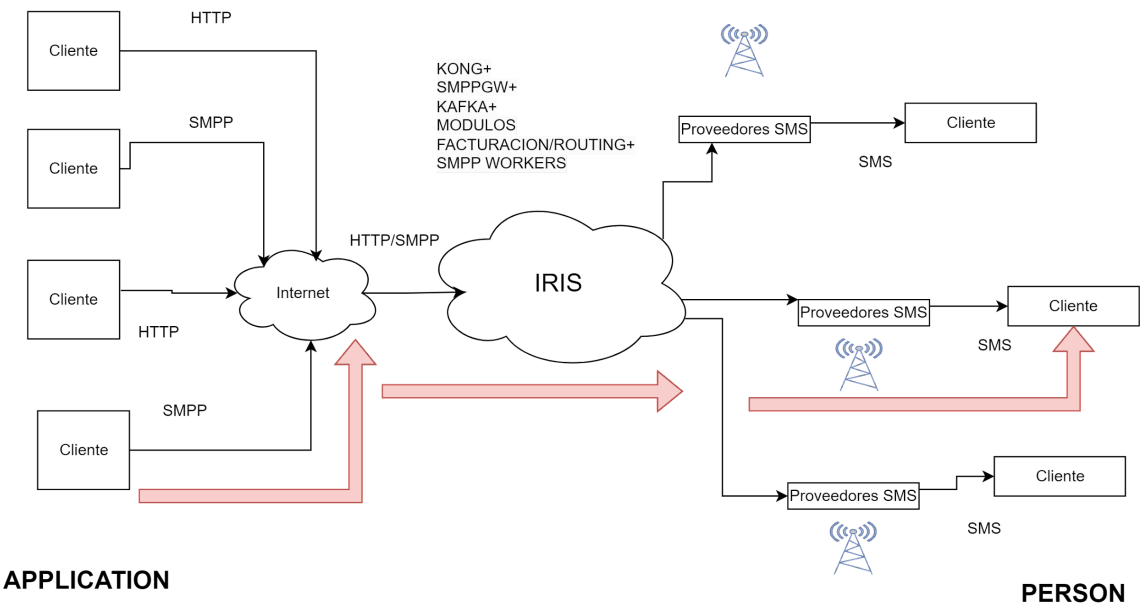

Figura 1. Esquema uso IRIS

En el esquema se muestra un ejemplo de uso de la plataforma IRIS para el envío de mensajes. Los clientes de la plataforma pueden utilizar dos protocolos para el envío de mensajes SMPP y HTTP. Estos mensajes son enviados a IRIS que los distribuye al proveedor SMS adecuado para completar el envío al destinatario final. Inicialmente IRIS tenía la posibilidad de recibir mensajes a través de SMPP y de HTTP, sin embargo solo se tenía implementado un balanceador de carga llamado KONG para el protocolo HTTP. Para SMPP simplemente se habían realizado pruebas de envío de mensajes SMS utilizando un único cliente Jasmin. En este trabajo se desarrollará un balanceador que cumpla una función similar a la de KONG pero para el protocolo SMPP que permita ponerlo en explotación.

Como se ha mencionado anteriormente cuando se envían mensajes P2P (Person-to-Person) los mensajes se envían habitualmente a través de las redes de telecomunicaciones móviles. Sin embargo en nuestro caso en el envío de mensajes A2P (Application-to-Person) existen dos mundos: el primero es el mundo IP con la aplicación enviando los mensajes a través de internet y la posterior recepción por los proveedores. Luego comienza el mundo no IP con la entrega de los mensajes a través de las redes de comunicaciones móviles a los terminales móviles personales.

1.2 Objetivos

Se tiene 3 objetivos fundamentales en este trabajo: obtener la máxima capacidad de procesamiento con los recursos que tenemos, procesar los mensajes SMS para optimizar su envío y gestionar las conexiones SMPP de los clientes. A continuación se procede a analizar en profundidad cada uno de los objetivos.

Para obtener la máxima capacidad de procesamiento con los recursos disponibles se ha decidido utilizar virtualización mediante máquinas virtuales o contenedores. En el caso de

los contenedores, estos utilizan sólo los recursos necesarios para ejecutar la aplicación, sin desperdiciar recursos en la duplicación de sistemas operativos completos como ocurre con las máquinas virtuales. Esto permite maximizar su uso y reducir los costos. Debido a que nuestra plataforma tiene clientes alrededor del mundo, será necesario desplegar múltiples instancias de esta, y para ello, los contenedores nos ofrecen mayor facilidad a la hora de moverse y ser ejecutados en diferentes plataformas. Otra ventaja clave es la posibilidad de utilizar orquestadores de contenedores que nos permiten escalar la aplicación en función de las necesidades, lo que significa que se pueden agregar o eliminar contenedores en función de la carga de trabajo. Esto permite que la aplicación propuesta sea eficiente con los recursos utilizados, algo muy importante para un sistema que tendrá múltiples picos y valles de trabajo en cada parte del mundo dependiendo de la hora local.

En paralelo, para optimizar el envío y manejo de un gran flujo de mensajes SMS se ha desarrollado un módulo balanceador de carga para el protocolo de SMPP. Su función será enviar y repartir los mensajes SMS a un servicio encargado del envío de mensajes.

Dentro de la gestión de sesiones se debe autenticar a los usuarios, gestionar que la conexiones cumplen los límites establecidos con SONOC y que cumplan la tasa de mensajes permitida. Gracias a esta gestión se conseguirá desacoplar la lógica interna IRIS relacionada con la facturación, enrutamiento, etc ... de la gestión de las sesiones y la recepción de mensajes a través del protocolo SMPP.

1.3 Estructura de la memoria

La memoria se compone de los siguientes capítulos:

- ➢ Capítulo 1: Introducción y objetivos del proyecto.
- ➢ Capítulo 2: Herramientas y protocolos utilizados para el desarrollo del proyecto.
- ➢ Capítulo 3: Explicación del contexto previo al desarrollo del proyecto.
- ➢ Capítulo 4:Solución al problema planteado.
- ➢ Capítulo 5: Implementación de la solución en un entorno Kubernetes.
- ➢ Capítulo 6: Pruebas de concepto y de carga realizadas sobre el escenario anterior.
- ➢ Capítulo 7:Conclusiones y líneas futuras.
- ➢ Referencias utilizadas para la redacción de la memoria y anexos complementarios.

2. Herramientas y protocolos

2.1 Protocolos

2.1.1 Protocolo TCP

TCP (Transmission Control Protocol) es un protocolo de nivel de transporte, es decir; que se encarga de la transferencia de datos entre los dos extremos de la comunicación libre de

errores. TCP es una especificación de la funcionalidad de la capa de transporte que permite el envío de paquete mediante el establecimiento de conexiones (realizado mediante un proceso llamado three-way handshake o negociación en tres pasos). Estas conexiones se crean para puertos determinados. TCP garantiza la llegada correcta y en orden de los paquetes emitidos, así como la monitorización del flujo para evitar problemas de congestión, a cambio del tiempo de procesado de la información que se necesita para ofrecer estas funcionalidades.

2.1.2 Protocolo HTTP

Según [HTT] “HTTP (Hypertext Transfer Protocol) se trata de un protocolo sin estado orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor. Los mensajes HTTP son en texto plano, lo que lo hace más legible y fácil de depurar”. HTTP define una serie de métodos de petición (algunas veces referido como "verbos") que pueden utilizarse. Cada método indica la acción que desea que se efectúe sobre el recurso identificado. Lo que este recurso representa depende de la aplicación del servidor. En nuestro trabajo únicamente se utilizan métodos POST: Envía datos para que sean procesados por el recurso identificado en la URL de la línea petición. Los datos se incluirán en el cuerpo de la petición. A nivel semántico está orientado a crear un nuevo recurso.

2.1.3 Protocolo SMPP

SMPP (Short Message Peer-to-Peer) es un protocolo de comunicación utilizado para enviar y recibir mensajes SMS (Short Message Service) entre dispositivos móviles y servidores de SMS.

El funcionamiento del protocolo SMPP se basa en un modelo cliente-servidor, donde el cliente SMPP se conecta al servidor SMPP a través de una conexión segura y envía mensajes SMS al servidor SMPP. El servidor SMPP procesa los mensajes y los envía a través de la red móvil al destinatario adecuado. La comunicación entre el cliente y el servidor se realiza a través de un canal dedicado y seguro, lo que garantiza la entrega confiable y segura de los mensajes SMS.
El protocolo SMPP permite la entrega de mensajes SMS en tiempo real, lo que significa que los mensajes se entregan al dispositivo móvil del destinatario casi al instante. También permite la entrega de mensajes en masa, lo que permite enviar mensajes a muchos dispositivos móviles de forma simultánea. SMPP es un protocolo muy eficiente y escalable que se utiliza en todo el mundo para el envío y recepción de mensajes SMS. Es compatible con diferentes lenguajes de programación y se puede integrar con sistemas existentes, lo que lo hace muy flexible y fácil de usar. Todas estas virtudes hacen necesario tener la posibilidad de utilizar SMPP para enviar mensajes a través de IRIS y por tanto desarrollar un balanceador de carga para el protocolo.

El protocolo SMPP consta de varios componentes:

- Cliente SMPP - ESME (External Short Message Entities) : el componente que envía los mensajes SMS al servidor SMPP. El cliente SMPP se conecta al servidor SMPP a través de una conexión TCP/IP segura.

- Servidor SMPP - SMSC (Short Message Service Center): es responsable de almacenar, reenviar y entregar mensajes SMS. Suele pertenecer a los proveedores de comunicaciones y actúa como intermediario entre los ESME y los dispositivos móviles.
- SMPP Gateway: Es una pasarela que actúa como un intermediario entre los ESME y los SMSC. Proporciona la funcionalidad de traducción y enrutamiento de mensajes entre diferentes protocolos.
- Centros de mensajes SMS - MS (Message Centres): los centros de mensajes SMS son los sistemas que procesan los mensajes SMS que se envían a través de SMPP. Estos centros de mensajes SMS pueden ser propiedad de los proveedores de servicios móviles, de los proveedores de servicios de SMS o de otras empresas.
- Enrutadores de SMS - RE Routing Entities: los enrutadores de SMS son sistemas que dirigen los mensajes SMS entrantes y salientes a través de la red móvil. Los enrutadores de SMS también pueden convertir los mensajes de un formato a otro y garantizar la entrega confiable de los mensajes.
- Mensajes SMPP: los mensajes SMPP son mensajes de texto que se envían a través del protocolo SMPP. Cada mensaje SMPP consta de un encabezado y un cuerpo. El encabezado contiene información como el identificador del mensaje, el número del remitente y el número del destinatario, mientras que el cuerpo contiene el texto del mensaje.

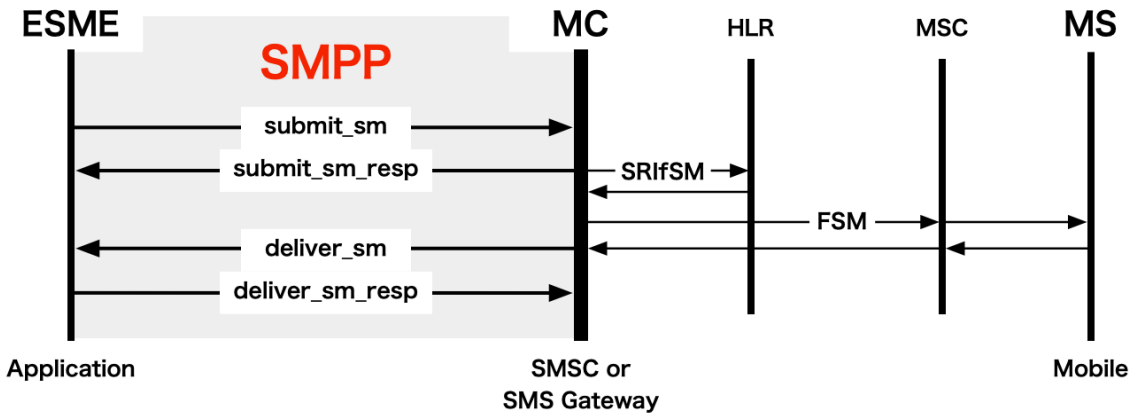

Figura 2. Escenario protocolo SMPP
Figura tomada de https://smpp.org/

SMSC (Short Message Service Center), Routing Entities y Message Centres pueden parecer lo mismo, pero en realidad representan diferentes elementos dentro de la arquitectura de la red. En general, Routing Entities y Message Centres se refieren a los sistemas de gestión de mensajes de texto que se utilizan para enrutar y procesar los mensajes de texto a través de la red, mientras que SMSC es un componente central del sistema, que se encarga de recibir, almacenar y reenviar mensajes de texto a través de la red de telefonía móvil.

Hay tres formas de iniciar sesión por el ESME:

- Transmisor (TX): cuando se autentica como transmisor, una ESME puede enviar mensajes cortos a la MC para su posterior envío a las estaciones móviles (MS).

- Receptor (RX): una sesión de receptor permite que una ESME reciba mensajes de una MC. Estos mensajes normalmente se originan en dispositivos móviles.
- Transceptor (TRX): una sesión TRX es una combinación de TX y RX, de modo que se puede usar una sola sesión SMPP para enviar mensajes hacia dispositivos móviles y recibir mensajes originados en dispositivos móviles.

Como se indica en la web [SMM] “El protocolo SMPP es un conjunto de operaciones, cada una de las cuales adopta la forma de una unidad de datos del protocolo (PDU) de solicitud y de respuesta. Por ejemplo, si una ESME desea enviar un mensaje corto, envía una PDU de submit_sm a la MC. El MC responde con una PDU submit_sm_resp, que indica el éxito o el fracaso de la solicitud. Del mismo modo, si un SMSC desea entregar un mensaje a una ESME, puede enviar una PDU deliver_sm a una ESME, que a su vez responde con una PDU deliver_sm _resp como forma de confirmar la entrega”.

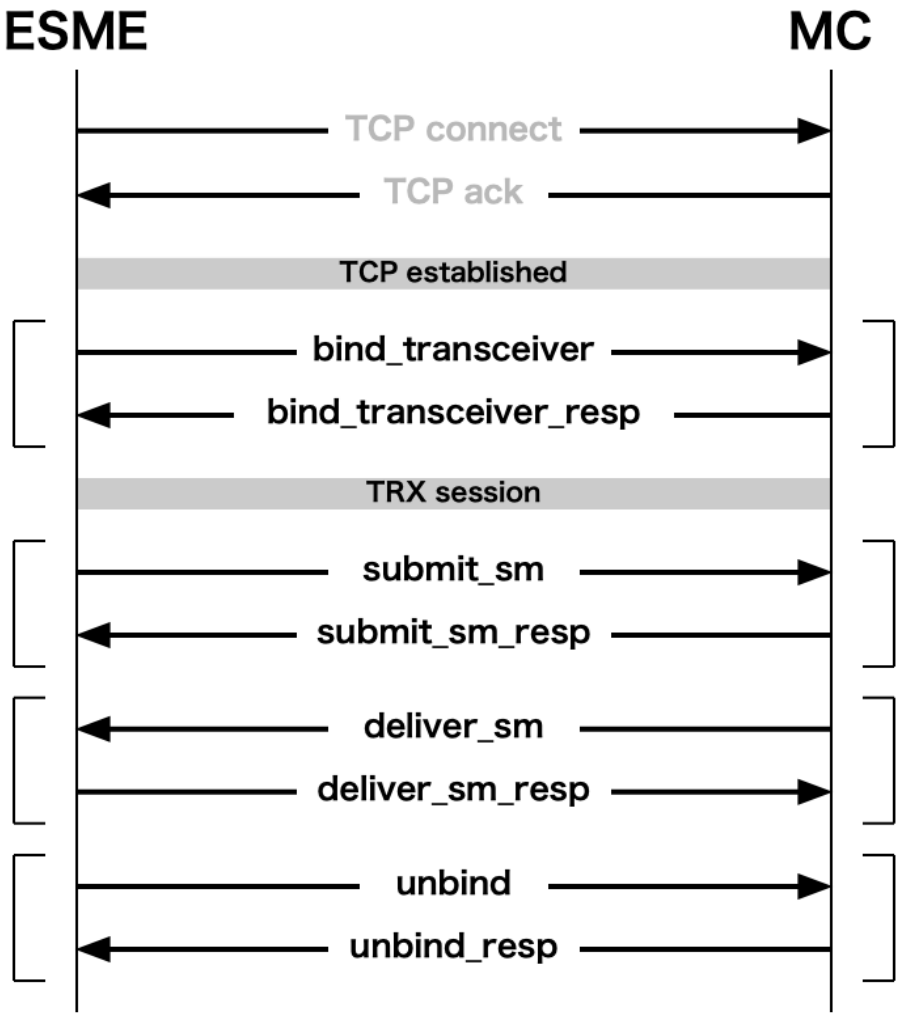

Figura 3. Intercambio de mensajes en una sesión SMPP
Figura tomada de https://smpp.org/

Para hacer uso del Protocolo SMPP, se debe establecer una sesión SMPP entre ESME y el MC o SMSC, según corresponda. La sesión establecida se basa en una conexión

TCP/IP de capa de aplicación. La conexión suele ser a través de Internet y puede usar SMPP sobre TLS o sin él. En el contexto de pruebas para el protocolo SMPP, los clientes SMPP actúan como ESME y el módulo IRISGW actúa como un SMS Gateway que se comunica con los centros de mensajes de los proveedores.

2.2 Herramientas

2.2.1 Java

Java es un lenguaje de programación orientado a objetos, independiente de la plataforma y seguro, con una variedad de bibliotecas y herramientas. Su capacidad para construir aplicaciones escalables y su amplio uso en diversas áreas, lo convierten en uno de los lenguajes más populares y ampliamente adoptados para el desarrollo de software. Para el desarrollo del trabajo se ha decido utilizar ya que posee la biblioteca JSMPP que permite desarrollar con el protocolo SMPP, además es el lenguaje que se utilizó en la empresa para el desarrollo del IRISGW por lo que era la mejor opción disponible.

2.2.2 Gitlab

Gitlab se trata de un servicio web de control de versiones y devops basado en git. GitLab permite gestionar, administrar, crear y conectar los repositorios con diferentes aplicaciones y hacer todo tipo de integraciones con ellas, ofreciendo un ambiente y una plataforma en cual se puede realizar las varias etapas del ciclo de vida del software y DevOps.
Permite la compartición de repositorios de código de forma pública o privada, además de características como validación de modificaciones, Integración continua y entrega continua, gestión de permisos granular, herramientas de revisión de código avanzadas y edición y visualización de archivos en el navegador.

2.2.3 Jasmin SMS Gateway

Jasmin es una plataforma SMS Gateway de código abierto diseñada para el envío y recepción de mensajes SMS. En el contexto del protocolo SMPP puede actuar tanto como un emisor (ESME) o como un receptor (SMSC) de mensajes SMS. En el trabajo se ha utilizado para simular clientes enviando mensajes SMS a través de SMPP hacia el balanceador de carga.
Para poder utilizar Jasmin primero debemos configurar un conector que comunique la plataforma con una red de proveedores de servicios SMS o SMSC que se utilizará para el envío y recepción de mensajes. Cuando se configura un conector en Jasmin, se establecen los detalles de conexión como el nombre de usuario, la contraseña, la dirección IP y el puerto del proveedor, entre otros detalles. En el anexo B se recoge la configuración del conector utilizado.

2.2.4 Kong

Kong es una plataforma de gestión de API de código abierto que actúa como un intermediario entre cliente y servidor HTTP. Incluye características como el balanceo de

carga de las peticiones a través de HTTP, además debido a las necesidades de autenticación y gestión puede contener algunos plugin personalizados para ampliar su funcionalidad. Un primer plugin es iris authentication que autentica al usuario y maneja que no se sobrepase el TPS (Transactions Per Second), también se encarga de enriquecer las cabeceras HTTP que han sido autenticadas. Otro plugin es DLR (Delivery Receipt) FORWARD PROXY Plugin que se encarga de reenviar la solicitud a un proxy que completará el envío del DLR al End Point fuera de la red SONOC para que este DLR llegue al cliente y pueda confirmar el estado de su mensaje enviado.

2.2.5 Docker

Como se indica en [DOC] “Docker es un proyecto de código abierto que permite el despliegue de aplicaciones en contenedores, los contenedores son una unidad de ejecución aislada y portátil que encapsula una aplicación junto con todas sus dependencias y configuraciones, lo que permite su despliegue y ejecución en diferentes entornos”. La principal ventaja que ofrece Docker es su gran comunidad de usuarios, siendo posible encontrar versiones contenerizadas de la mayoría de aplicaciones open-source del mercado. Los contenedores de Docker se despliegan a través de ficheros de configuración sencillos llamados Dockerfile o Docker-compose escritos en texto plano y con una estructura propia.

2.2.6 Kubernetes

Como se indica en la web [KUB] “Kubernetes es un proyecto de código abierto que permite la automatización, ajuste de escala y despliegue de aplicaciones en contenedores, también conocido como orquestador”. Su funcionalidad se basa en el uso de un clúster que coordina la comunicación entre varios nodos (máquinas físicas) en los que se encuentran los contenedores, los cuales se pueden comunicar entre sí y con el exterior para ofrecer servicios de todo tipo. La configuración de los objetos creados en un entorno de Kubernetes se realiza mediante el uso de imágenes de contenedor (como las de Docker) y ficheros YAML para definir la configuración. Se ha decido utilizar la versión completa de Kubernetes (K8s) antes que su versión más ligera K3s, ya que se pretende trabajar con un número elevado de nodos.

La unidad mínima de abstracción en Kubernetes es el pod, que consiste en uno o más contenedores de software que comparten recursos de hardware definidos. Además, no poseen una dirección IP estática, por lo que se requieren de otros elementos para enviar y recibir peticiones desde otro pod. Para la puesta en marcha de los pods hay que decidir qué tipos de controladores de ejecución en Kubernetes es más adecuado, dadas las necesidades del servicio. Existen varios tipos de controladores de pods que te permiten definir cómo se ejecutan y se gestionan las aplicaciones. Se han estudiado los tipos de controladores y cuales son sus ventajas y desventajas para nuestras aplicaciones:

- ReplicaSet: Un ReplicaSet asegura que un número especificado de réplicas de un pod se esté ejecutando en el clúster de Kubernetes en todo momento. Si se detecta que un pod no está funcionando correctamente, el ReplicaSet lo reemplaza automáticamente por otro. Una ventaja de ReplicaSet es que te permite definir fácilmente el número deseado de réplicas, lo que proporciona alta disponibilidad

para tus aplicaciones. Sin embargo, si necesitas actualizar la definición de un pod, debes actualizar manualmente el ReplicaSet para que se repliquen las actualizaciones.

- Deployment: Un Deployment es similar a un ReplicaSet, pero con la capacidad adicional de actualizar las aplicaciones sin tiempo de inactividad. Cuando se realiza una actualización en la definición del pod, un Deployment crea un nuevo conjunto de pods con la nueva definición y los reemplaza gradualmente con los pods antiguos. Una ventaja de Deployment es que proporciona una actualización sin tiempo de inactividad para tus aplicaciones, lo que minimiza la interrupción del servicio. Sin embargo, la actualización gradual puede llevar más tiempo en comparación con el reemplazo inmediato de todos los pods de un ReplicaSet.
- StatefulSet: Un StatefulSet es un controlador de pods que se utiliza para aplicaciones que requieren identificadores de red únicos y estables. Cada pod en un StatefulSet tiene un nombre único y un orden establecido. Una ventaja de StatefulSet es que proporciona una identidad estable para cada pod, lo que es necesario para aplicaciones de bases de datos y otras aplicaciones que requieren almacenamiento persistente. Sin embargo, la creación y actualización de pods en un StatefulSet puede llevar más tiempo que en un ReplicaSet o Deployment. Como en nuestro caso no se necesita almacenamiento persistente en las máquinas descartamos esta opción.

Finalmente para el despliegue de ambas aplicaciones se optó por usar deployment que asegura un conjunto de pods establecidos se esté ejecutando en el clúster de Kubernetes, y maneja la creación y la actualización de estos pods. En un mismo clúster pueden existir diferentes deployment, cada uno implementando la funcionalidad de los contenedores definidos en un único pod.

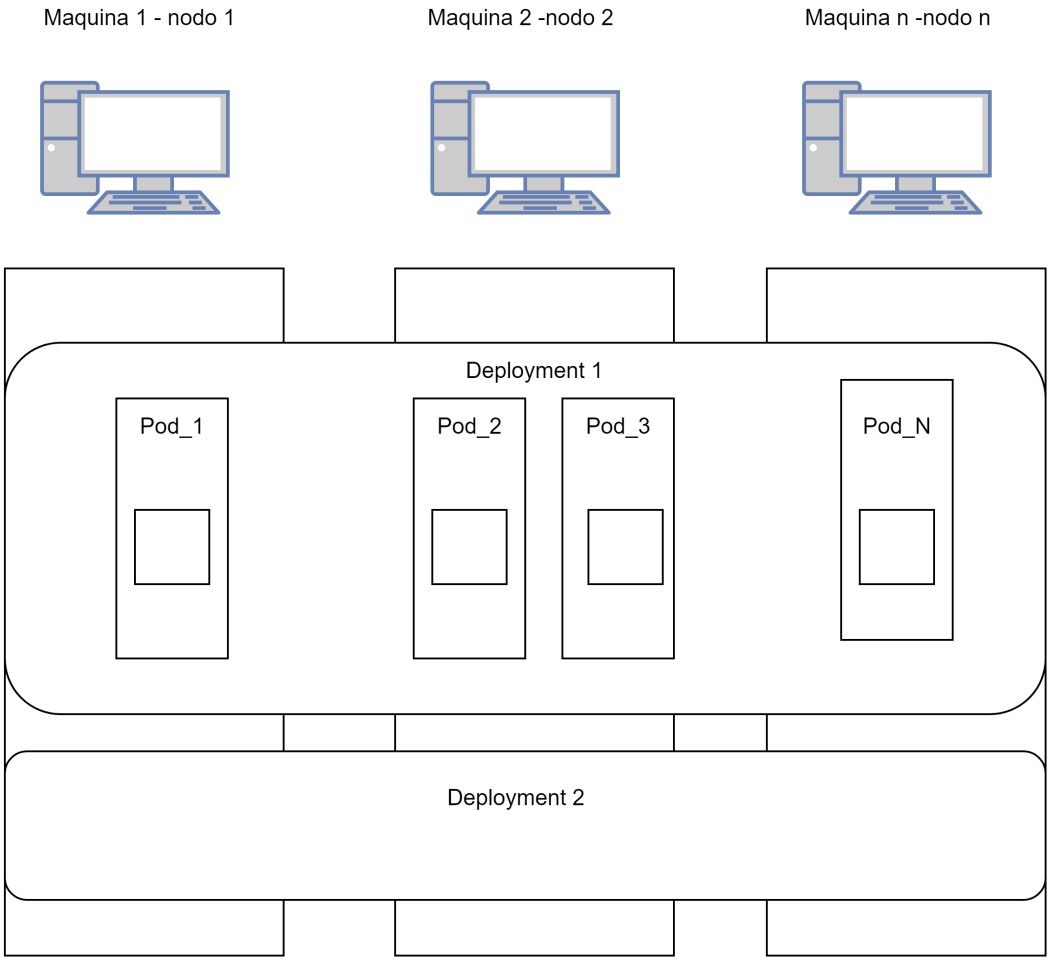

Figura 4. Abstracción de aplicaciones en kubernetes

En la figura 4 se observa un ejemplo de cómo se distribuyen las aplicaciones en contenedores en un clúster de Kubernetes con múltiples nodos. 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.

2.2.7 Prometheus

Prometheus es un proyecto de código abierto y una herramienta de monitoreo y visualización de métricas en entornos distribuidos. Prometheus está diseñado para recopilar métricas de diversos servicios y sistemas, lo que permite monitorear y analizar el rendimiento de las aplicaciones y los recursos. Puede recolectar métricas en tiempo real, almacenarlas en una base de datos de series temporales y generar alertas cuando se superan ciertos umbrales. Estas características le hacen una herramienta de monitorización muy popular en entornos desplegados con kubernetes.

3. Escenario previo

3.1 Funcionamiento IRIS

Ahora mismo ya existen algunos módulos de la plataforma que permiten las siguientes funcionalidades. Toda la información relacionada con IRIS se ha obtenido de la documentación interna de SONOC para el proyecto.

Funciones:

1. Para el protocolo de comunicación HTTP
    a. Autenticación de mensajes entrantes.
    b. Control de transacciones por segundo (TPS), tanto globales como por cliente/proveedor.
    c. Control de transacciones activas, tanto globales como por cliente/proveedor.
2. Logging
3. Exponer métricas
    a. TPS Totales.
    b. TPS por cliente/proveedor.
    c. Mensajes activos totales en la plataforma.
    d. Mensajes activos por cliente/proveedor.
4. API de configuración
5. Insertar en la cola de mensajería:
    a. Mensajes entrantes vía HTTP API.
    b. PDU.
    c. DLR.
    d. Alertas.
6. Consumir de la cola de mensajería
    a. Mensajes rechazados
    b. DLR
7. Lógica de re-enrutamiento
    a. Aviso si un proveedor está caído o disabled
    b. Aviso si un proveedor ha alcanzado el límite de TPS
    c. Aviso si un proveedor rechaza el mensaje

3.2 IRIS: visión general

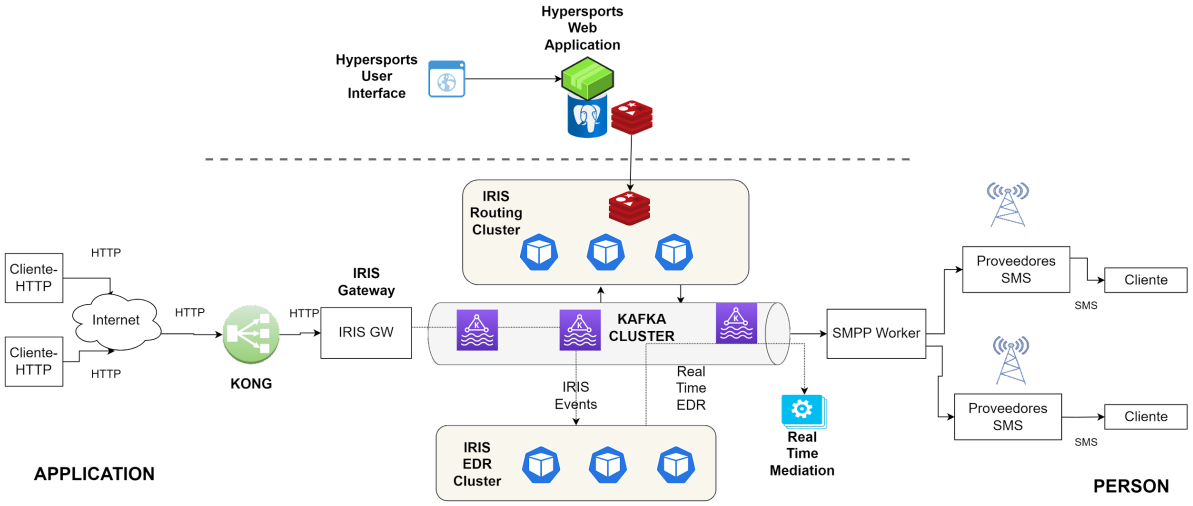

Figura 5. Escenario inicial IRIS

En la parte superior del esquema de la figura 5 se observa que cada cliente de la plataforma IRIS tiene disponible un servicio web llamado hypersports donde puede dar de alta los usuarios a los que envía los mensajes y los proveedores que utiliza, además de configurar las rutas para los envíos, facturación y parámetros básicos como número de binds disponibles para abrir. Esta configuración quedará guardada en una base de datos que será consultada posteriormente por módulos de enrutamiento de IRIS. Todos los usuarios que estén configurados en hypersports compartirán el tenant asignado a nuestro cliente de forma que se tienen siempre identificados.

Actualmente los clientes solo pueden enviar mensajes a través del protocolo HTTP. Los clientes envían peticiones HTTP hacia Kong especificando los datos del mensaje con la forma recogida en el anexo A. Después del proceso de autenticación de Kong se procede a reenviar esa petición HTTP hacia la API del IRISGW.
IRISGW es el componente del sistema que maneja todas las conexiones de clientes y proveedores para un tenant específico. Se accede a el IRISGW a través de una API HTTP, actualmente la API solo nos ofrece un método send para enviar mensajes, sin embargo se está trabajando para ampliar las operaciones disponibles. IRISGW se encarga de revisar las cabeceras de la petición HTTP para comprobar que la petición ha sido verificada previamente por Kong. Posteriormente el IRISGW inyecta los mensajes en una cola de mensajería.

De la misma forma que nosotros podemos limitar la tasa de mensajes por segundo que envían nuestros clientes, los SMSC de los proveedores pueden tener una limitación en el número de mensajes por segundo que puede procesar. Por ejemplo para proveedor de comunicaciones Vonage (https://www.vonage.com.es/) se ha observado que la tasa de mensajes entrantes es unas 10 veces mayor que la que puede procesar el SMSC. La solución a esta limitación se ha resuelto utilizando unas colas de mensajes distribuidas que son consumidas a la tasa que acepta el proveedor. Como solución para la cola de mensajes distribuida se ha elegido Apache Kafka, una solución muy extendida en la industria, que proporciona todas las garantías necesarias para ser el fundamento de la arquitectura y además permite organizar en topics los distintos mensajes.

La arquitectura seleccionada, cuyo objetivo es lograr un elevado nivel de transacciones por segundo, así como la posibilidad de que nuevos módulos sean directamente “enchufables” siempre y cuando se ajusten al protocolo definido, se basa en algunas ideas principales:

1. Se desacopla el procesamiento de bajo nivel relacionado con las sesiones y protocolos de la lógica de negocio.
2. Procesos asíncronos.
3. Persistencia de la información en la cola de mensajería.

Se muestra en mayor detalle en la figura 6 cómo sería el procesamiento de un mensaje.

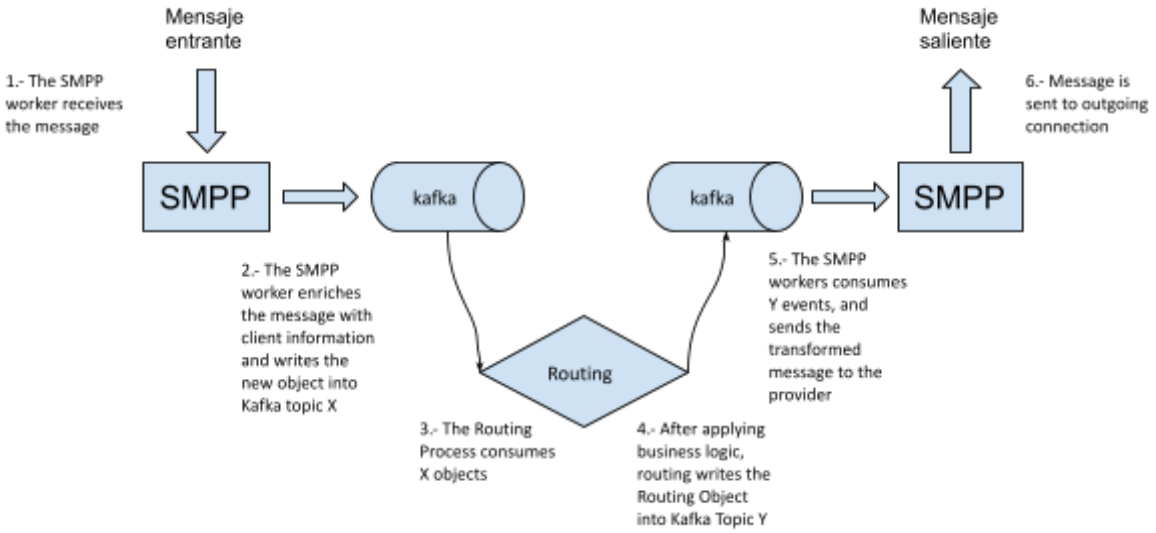

Figura 6. Procesamiento de un mensaje

Figura tomada de documentación interna de SONOC

El cluster de enrutamiento se encarga de establecer el routing que deben seguir los mensajes consultando lo establecido por el cliente a través de Hypersports. El cluster de EDR se encarga de detectar y responder ante posibles errores durante el proceso del envío de mensajes.

Finalmente, los mensajes SMS se envían por el proveedor o grupo de proveedores determinado por la lógica de enrutamiento hacía los clientes finales.

4. Solución

En este capítulo se va a explicar las dos soluciones una inicial y una avanzada desarrolladas para el problema planteado.

4.1 Solución Inicial

Como se puede observar en la figura 7, para que los clientes puedan enviar mensajes SMS a través del protocolo SMPP de una forma eficiente se debe desarrollar un módulo de

balanceador de carga que realice una función similar a Kong pero para el protocolo SMPP. En caso de que los clientes quieran utilizar el protocolo SMPP para el envío de mensajes deberán iniciar previamente una conexión con el Balanceador SMPP. Cuando se reciben los mensajes se realiza una autentificación (Para el SMPP, el mensaje debe provenir de un cliente habilitado en la plataforma. Por ejemplo, si un BIND está activo, pero el cliente se ha bloqueado por un problema de crédito, el mensaje se debe rechazar) y un control de la conexión y de la tasa de mensajes permitida. Posteriormente si se completa correctamente las verificaciones anteriores se envían los mensajes a través de una petición HTTP que invoca la API del IRISGW.

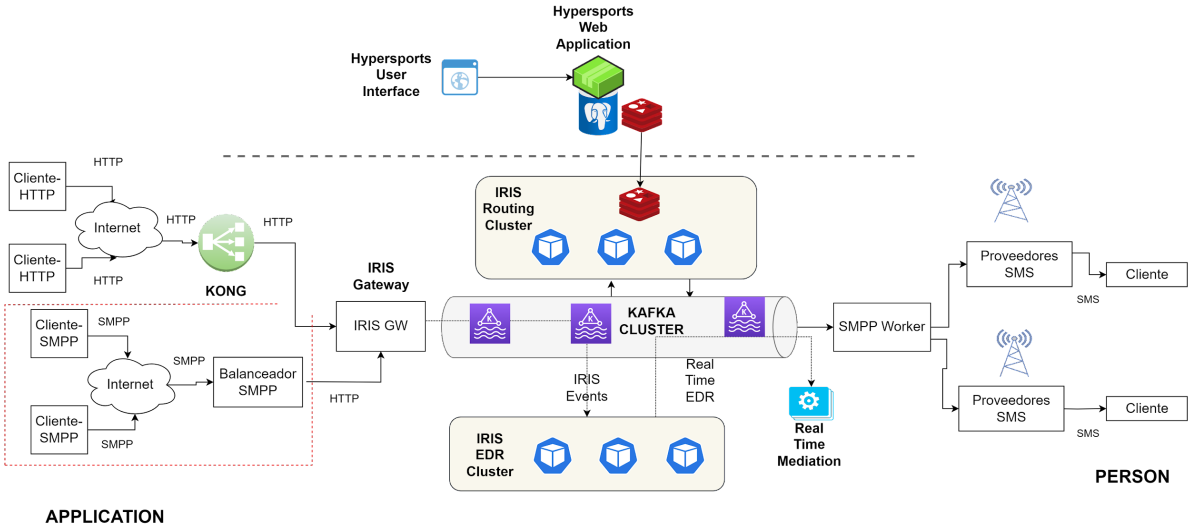

Figura 7. Solución inicial propuesta

El objetivo de desarrollar este módulo es desacoplar la lógica interna IRIS relacionada con la facturación, enrutamiento, etc… de la gestión de las sesiones y la recepción de mensajes. Debido a las decisiones tomadas en el desarrollo del balanceador SMPP el módulo IRISGW solo necesitará la API HTTP, simplificando la gestión de los mensajes y unificando los dos tipos de clientes HTTP y SMPP.

4.1.1 Alternativas de desarrollo

A la hora de realizar el desarrollo se manejaron varias posibilidades. La primera de ellas consiste en que el módulo permita establecer conexiones SMPP con clientes y cuando llegue un mensaje de un cliente se inicie otra sesión SMPP con IRISGW, de forma que se mantengan las dos sesiones abiertas de forma simultánea y se reenvíe los mensajes del cliente al IRISGW. Esta opción se ha descartado porque obliga a que IRISGW maneje dos protocolos el SMPP y HTTP y tenga que diferenciar entre los dos tipos de clientes y también por la dificultad de escalar esta solución.

La alternativa que se ha desarrollado es que el balanceador de carga SMPP establezca conexiones SMPP con los cliente para recibir los mensajes pero que sean enviados al IRISGW utilizando la API disponible con el protocolo HTTP. De forma inversa se mandará la confirmación DLR de vuelta al cliente.

4.1.2 Funcionalidades Distribuidor de carga SMPP

Para crear el módulo se ha utilizado como base el código de ejemplo de SMPPServerSimulator que ya incluye la funcionalidad básica de un servidor de SMPP: establecer conexiones SMPP, recibir mensajes y mandar un DLR automáticamente a estos mensajes. El código de SMPPServerSimulator se puede clonar del siguiente repositorio de Github
https://github.com/opentelecoms-org/jsmpp/blob/master/jsmpp-examples/src/main/java/org/jsmpp/examples/SMPPServerSimulator.java

A esta funcionalidad básica ha sido necesario añadir 3 funcionalidades: autentificación de usuarios y comprobación de que las peticiones se ajusta a lo contratado para el cliente, adaptación del protocolo SMPP al protocolo HTTP de forma transparente para el cliente a través de la creación de peticiones HTTP a partir de los mensajes recibidos y recepción de confirmación de llegada del mensaje al destino final transmitido desde IRISGW a través de HTTP y envío del correspondiente DLR a nuestro cliente a través de SMPP.

Para la autentificación de los clientes se realiza una consulta a una base de datos redis a través del cliente de java jedis de donde se extraen los usuarios registrados, sus contraseñas y el número de binds que tiene disponibles, posteriormente se compara si alguno de ellos coincide con el cliente y contraseña de la sesión que se inicia. Si no coincide no se permite establecer las conexiones SMPP y se informa al cliente, en caso contrario se autentica el cliente y se comprueba que tiene disponibles binds para establecer una sesión. Se ha utilizado Redis ya que es necesario poder realizar modificaciones de forma rápida y dinámica. Para hacer las pruebas del sistema, ya que no se tenían todos los componentes de IRIS funcionando, no se ha utilizado Redis para consultar la información sobre cliente sino que se han utilizado datos incorporados de forma estática a la aplicación.

Cuando se recibe un mensaje desde un cliente autorizado es necesario comprobar que no se está sobrepasando la tasa de mensajes por segundo (TPS) permitida. Además se debe crear una petición HTTP a partir del mensaje recibido por SMPP para que se envíe a través de la API del IRISGW.

Como se ha mencionado anteriormente el protocolo SMPP no tiene garantía de entrega. Por tanto es necesario devolver una respuesta a nuestro cliente SMPP para confirmar si se ha enviado el mensaje correctamente. Cuando un mensaje es recibido por la otra persona la SMSC nos genera un DLR para que sea enviado al cliente, en nuestro caso es el IRISGW que cuando se procesa correctamente la petición HTTP del mensaje nos devuelve un código HTTP 200 y un mensaje de confirmación que se utiliza como DLR para el cliente. En la figura 8 se ilustra el funcionamiento interno del Balanceador SMPP.

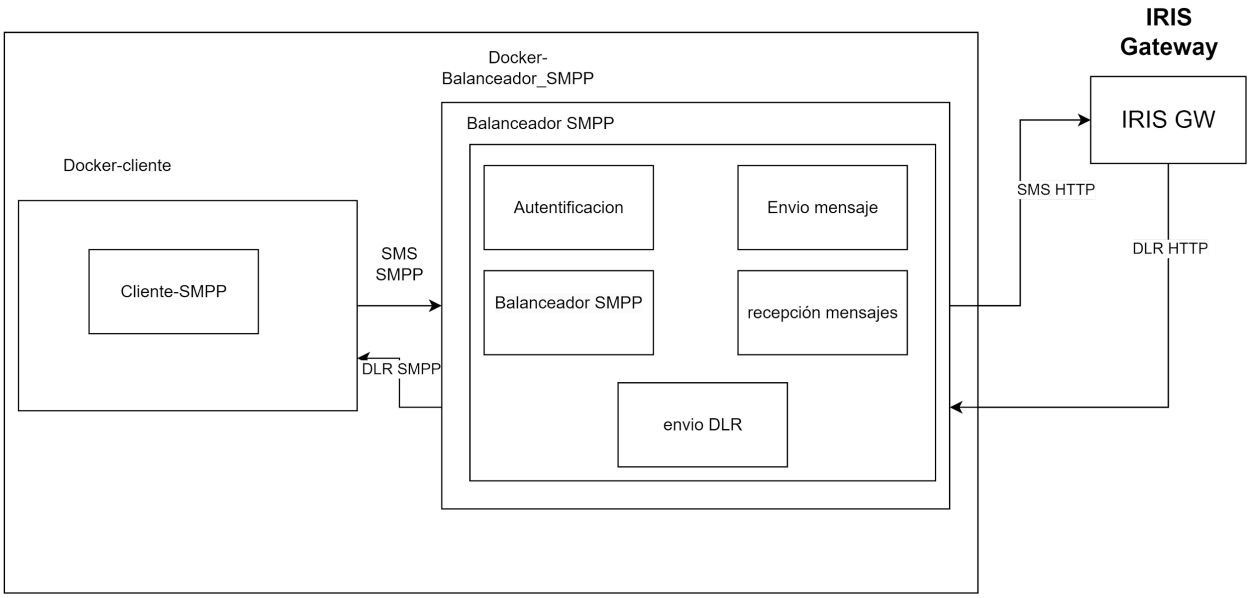

Figura 8. Funcionalidades Balanceador SMPP

El objetivo de esta primera solución era desarrollar y probar todas las funcionalidades que se necesitaban para posteriormente modularizar estas funcionalidades y contenerizarlas (empaquetar una aplicación y todas sus dependencias en un contenedor) buscando desarrollar una arquitectura distribuida que permita una mejora en el rendimiento global de IRIS.

4.1.3 Implementación Distribuidor de carga SMPP

Desde el programa principal se lanza un método que se encarga de establecer un SMPPServerSessionListener que nos permite escuchar las conexiones entrantes y de establecer un MessageReceiverListener para recibir los mensajes de los clientes.
En el contexto de Java, un listener es un componente utilizado para recibir y responder a eventos generados por otros objetos o componentes en un programa.

Cuando un cliente desea establecer una conexión con el balanceador SMPP, primero debe realizar un proceso de enlace (bind) para autenticarse y establecer una sesión SMPP válida. Durante este proceso, el cliente envía un mensaje de enlace (bind) al balanceador y espera una respuesta de confirmación. La clase WaitBindTask es responsable de esperar y manejar la respuesta de confirmación del proceso de enlace entre el cliente y el balanceador SMPP. Para poder confirmar el enlace es necesario comprobar la autenticación mencionada en el apartado anterior. La clase se ejecuta en un hilo separado para permitir que el cliente continúe ejecutando otras tareas mientras espera la respuesta del enlace.

La interfaz MessageReceiverListener define un conjunto de métodos que deben ser implementados para manejar eventos relacionados con la recepción de mensajes SMS desde un servidor SMPP. El método onAcceptSubmitSm se invoca cuando se recibe un mensaje SMS entrante en el servidor SMPP y permite acceder al contenido del mensaje y realizar acciones en función de su contenido o remitente, se han añadido las funcionalidades relacionadas con la recepción y envío de mensajes mencionados en el

apartado anterior además del control de la tasa de mensajes por segundo establecida. Este método también se encarga de invocar la ejecución de DeliveryReceiptTask.

La clase DeliveryReceiptTask es una implementación de la interfaz Runnable que permite crear hilos y se utiliza para recibir y procesar de manera asíncrona los DLR de los mensajes enviados.

4.2 Solución avanzada

Para esta solución el módulo Balanceador SMPP implementado anteriormente se divide en dos módulos (figura 9). El primero llamado proxy se encargaba de la gestión de las conexiones SMPP, autenticación de clientes y control de número de binds, de que se respete la tasa de mensajes por segundo y de la recepción de los mensajes a través de la conexión SMPP. El otro módulo llamado envio_mensaje está dedicado a la gestión del envío de mensajes y posterior recibimiento del DLR del mensaje por parte del IRISGW.

Inicialmente se tenían todas las funcionalidades comentadas anteriormente en un único módulo, sin embargo después de hacer pruebas se buscó desacoplar la gestión y control de las conexiones smpp del envío del mensaje. De forma que se puede dimensionar uno de los servicios y hacer crecer el otro de forma dinámica dependiendo del número de mensaje a enviar mejorando las prestaciones del sistema.
La mejora en el rendimiento se consigue colocando múltiples instancias del servicio de envío de mensaje por cada servicio de proxy. Se observó que como se conoce de antemano el número de clientes de la plataforma IRIS y que abrir y cerrar binds es poco costoso se podía dimensionar el sistema y desplegar un módulo de proxy por cada cliente de forma que no era necesario lanzarlos de forma dinámica como el módulo de envío de mensajes.

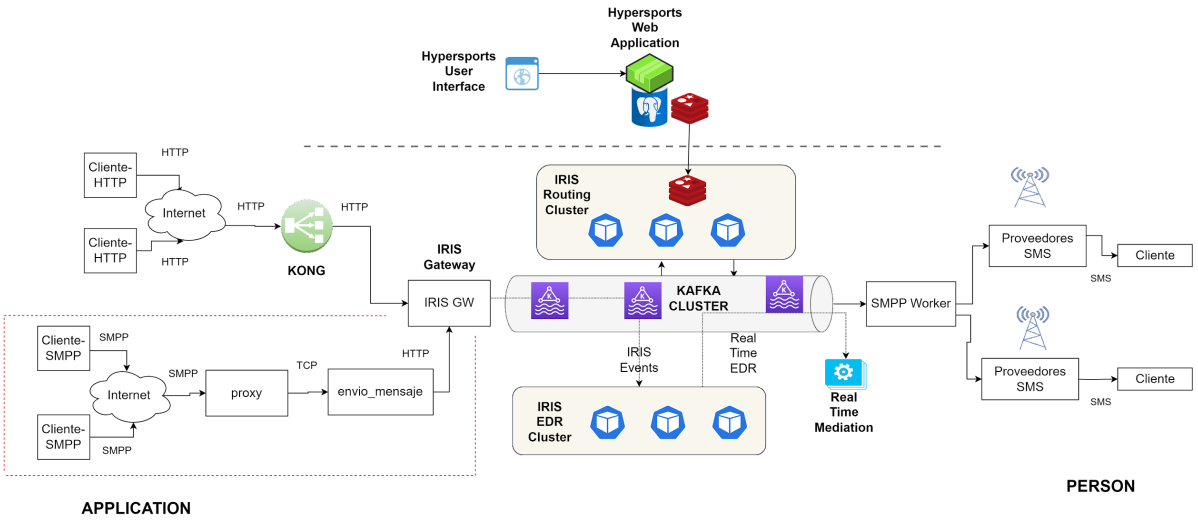

Figura 9. Solución avanzada

En la figura 10, se ve como en nuestra solución se espera que el módulo de envío de mensaje reciba un DLR a través de HTTP en una dirección y puerto establecidos. Este

servicio se encarga de enviar el DLR al módulo LB que le envió el mensaje para que luego sea este el que se lo envíe a través de SMPP a nuestro cliente.

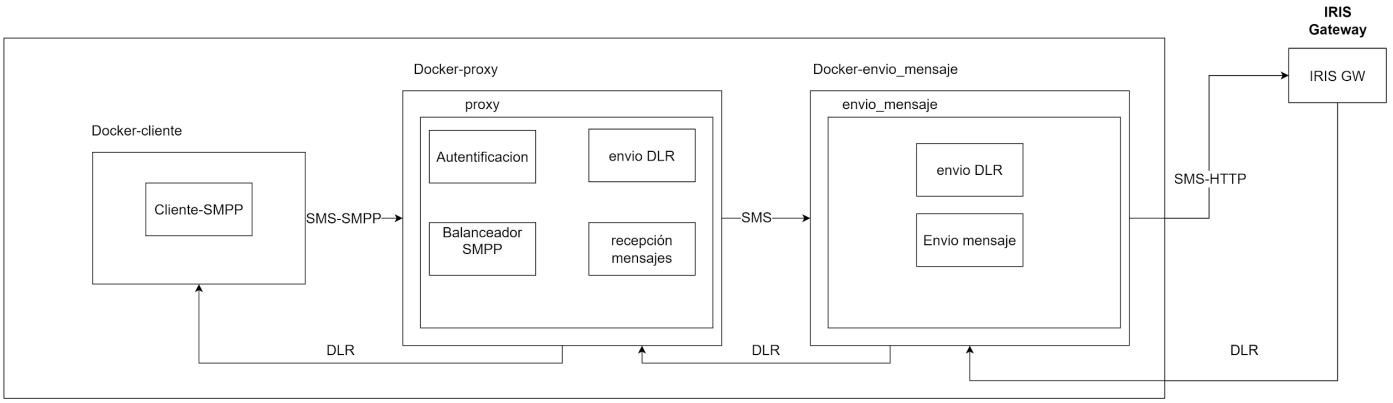

Figura 10. Esquema funcionalidades Proxy y Envío mensaje

También se puede ver cómo el módulo proxy es el encargado de establecer la conexión SMPP con el cliente, después de la autentificación y verificación de la conexión el cliente puede empezar a mandar mensajes. Cuando el proxy recibe un mensaje lo manda a través de una conexión con sockets que tiene con el servicio de envio_mensaje, este servicio es el encargado de repartir entre las réplicas del módulo para que envíen los mensajes al IRISGW.

Para la comunicación entre los módulos se ha utilizado la biblioteca sockets de java que permite crear sockets y mandar mensajes con la forma de protocolo SMPP a través de ellos pero sin la necesidad de establecer conexiones SMPP. Los pods de proxy actúan como clientes que se conectan y envían los mensajes de los clientes a los servidores sockets que serán los pods de envio_mensaje.

5.Despliegue de la solución

A la hora de implementar el despliegue de la solución surgieron dos alternativas: la primera, tener una instancia del balanceador de carga y múltiples instancias del IRIS GW. El balanceador era responsable de recibir los mensajes enviados por los clientes y enviarlos por HTTP a Kong para que los reparta entre las distintas instancias del IRISGW.

La otra alternativa por la que se optó es dejar una única instancia de IRISGW y tener múltiples instancias del distribuidor de carga SMPP que se encargaran de manejar las peticiones de los clientes y enviarlas.

5.1 Elementos de Docker

Para poder contenerizar nuestras aplicaciones desarrolladas en java, primero se debe generar el JAR que contenga el código de la aplicación y todas sus dependencias. En el anexo C se incluyen los pasos necesarios para la creación de jar.

Ya se tiene empaquetada la aplicación con todo lo necesario para ejecutarla; ahora se debe crear un dockerfile para generar la imagen de la aplicación. El primer paso para la implementación de cualquier aplicación en un entorno de Kubernetes es la creación de las imágenes de contenedor que se quiere 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. Utilizaremos las aplicaciones anteriormente empaquetadas y lo único que se tiene que hacer es realizar los cambios requeridos para que nuestra aplicación funcione, como por ejemplo, copiar los ficheros de configuración específicos que se han desarrollado, ejecutar comandos o exponer puertos. En el anexo D se recogen los dockerfile utilizados.

Para construir la imagen de Docker a partir del Dockerfile:

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

Se puede hacer una prueba de funcionamiento de la imagen con el motor de Docker utilizando:

- **docker run -p <puerto>:<puerto> nombre_de_imagen**

Ahora se tiene disponible un contenedor Docker donde se corre la aplicación y que tiene asociado un puerto real de la máquina con el puerto del contenedor docker.

Para poder utilizar las imágenes que hemos creado de nuestras aplicaciones y desplegarlas en Kubernetes en el cluster de SONOC, primero se necesita subir las imágenes a su Docker registry privado para que queden accesibles.
Se etiquetan las imágenes:

- **docker image tag proxy dockers.sonoc.io/proxy**
- **docker image tag envio_mensaje dockers.sonoc.io/envio_mensaje**

Se realiza el push de las imágenes con el tag al repositorio:

- **docker push dockers.sonoc.io/proxy**
- **docker push dockers.sonoc.io/envio_mensaje**

5.2 Despliegue con Kubernetes

Una vez creadas las imágenes de contenedor que se van a utilizar, se puede pasar a su implementación en un entorno de Kubernetes. Como ya se mencionó anteriormente, 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. En nuestro caso vamos a utilizar un cluster que tiene configurado la empresa para simular un entorno de producción.

Dadas las distintas necesidades que tendrán los dos servicios, debido a su naturaleza, tendremos que incluir componentes de Kubernetes que actúen sobre los deployment de las aplicaciones que tenemos. Para el primer servicio de proxy, puesto que el número de clientes que abren conexiones es fijo, se puede dimensionar el sistemas y establecer un número fijo de réplicas. Sin embargo, para el segundo caso se necesita poder autoescalar

el número de réplicas para el envío de mensajes conforme los clientes necesiten enviarlos. También se debe de considerar que en el caso del primer servicio se debe mantener conexiones con los clientes y no se pueden eliminar máquinas ya que se podrían perder máquinas con conexiones abiertas con los clientes.

5.2.1 HPA (Horizontal Pod Autoscaler)

Para la orquestación de los contenedores que ejecutan la aplicacion de enviar mensajes es necesario un componente de Kubernetes 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 se puede ver en las figuras 11 y 12:

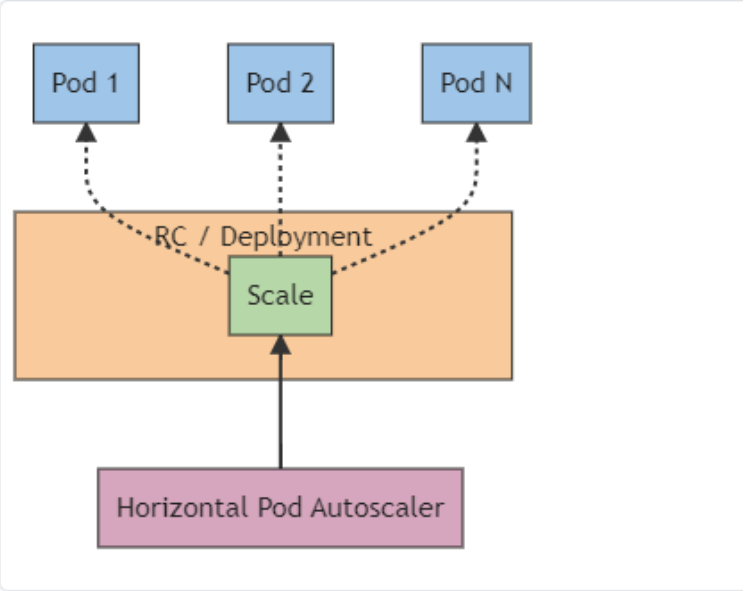

Figura 11. Kubernetes Horizontal Pod Autoscaler

Figura tomada de https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

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

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*.

Figura 12. Fórmula Autoescalado

Por ejemplo, si el valor actual de la métrica es 200 m y el valor deseado es 100 m, el número de réplicas se duplicará, ya que 200,0 / 100,0 == 2,0 Si el valor actual es 50 m,

reducirá a la mitad el número de réplicas ya que 50.0 / 100.0 == 0.5. La fórmula de autoescalado ha sido tomada de la documentación oficial de kubernetes https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Kubernetes implementa el escalado automático horizontal de pods como un bucle de control que se ejecuta de forma intermitente. El intervalo lo establece el parámetro --horizontal-pod-autoscaler-sync-period para el kube-controller-manager y de forma predeterminada es 15 segundos, para las ha sido necesario ajustar este parámetro a un intervalo menor.

En Kubernetes se pueden configurar diferentes métricas para el escalado horizontal utilizando el Horizontal Pod Autoscaler (HPA). Estas métricas se dividen en dos categorías principales:
Métricas estándar:

- Utilización en porcentaje de la CPU de los pods.
- Utilización en porcentaje de la memoria de los pods.

Métricas personalizadas:

- Métricas personalizadas relacionadas con objetos de Kubernetes como número de pods iniciándose.
- Métricas personalizadas no relacionadas con objetos de Kubernetes como número de peticiones HTTP o mensajes recibidos por la aplicación. En este caso son las propias aplicaciones las que deben exponer estas métricas para que puedan ser utilizadas por Kubernetes.

Inicialmente se utilizó el uso de memoria como métrica predeterminada para realizar el autoescalado ya que no requería de ninguna configuración adicional. Tras realizar pruebas se observó que utilizando una una métrica personalizada basada en el número de mensajes recibidos por segundo se lograba un autoescalado más preciso lo que se traduce en una mejora en el rendimiento y en el consumo de recursos.

Nuestra métrica personalizada se expondrá a través de Custom Metrics API para ser utilizada por el HPA. Para utilizar métricas personalizadas, es necesario que el clúster Kubernetes tenga una solución de monitoreo, como Prometheus, configurada y en funcionamiento. En el anexo G se encuentra explicada toda la configuración que ha sido necesaria para utilizar métricas personalizadas.

En el anexo F se tiene un archivo que describe un HorizontalPodAutoscaler que escala automáticamente el número de réplicas del deployment de envio_mensaje en función de la utilización de la memoria y de la métrica personalizada mensajes recibidos por segundo. La configuración establece un mínimo de 1 réplica y un máximo de 10 réplicas que se ajustan automáticamente en función del número de peticiones.

5.2.2 Networking

Los servicios de Kubernetes son una abstracción que permite el descubrimiento, equilibrio de carga y comunicación entre los pods de una aplicación. Como puede verse en la figura

13, para la primera aplicación proxy se ha utilizado un servicio NodePort, que expone la aplicación en un puerto en cada nodo del clúster y asigna un puerto externo para acceder al servicio a través de una IP pública única. En nuestra configuración se utiliza el puerto 8057 para acceder al servicio y se especifica que la conexión utiliza el protocolo TCP. Esto proporciona la IP pública y el puerto para acceder al servicio a través de la red.

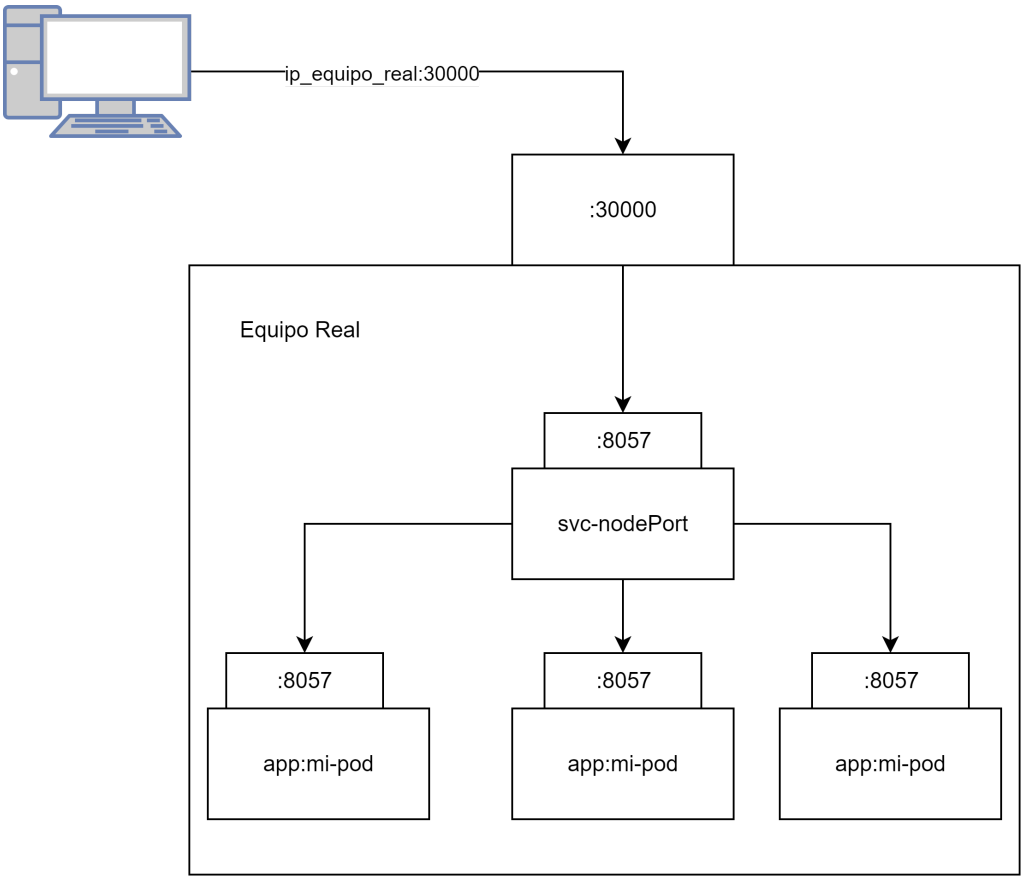

Figura 13. Esquema funcionamiento nodePort

Un servicio NodePort es la forma más sencilla de permitir tráfico externo directamente a un servicio. NodePort, como su nombre lo indica, abre un puerto específico en todos los nodos (las máquinas virtuales), y cualquier tráfico que se envíe a este puerto se reenvía al servicio. Cuando la aplicación cliente se conecta al servicio NodePort, Kubernetes utiliza su balanceador de carga interno para distribuir automáticamente la conexión entre las réplicas del servicio. Cada réplica del servicio NodePort tendrá su propia dirección IP, que se usará para enrutar la conexión entrante a la réplica correspondiente. La distribución del tráfico se realiza de manera transparente para el cliente.

Respecto a la comunicación entre los dos deployments, cuando un pod que ejecuta proxy quiere comunicarse con los pod de envio_mensaje, puesto que están en el mismo clúster de Kubernetes, utilizará el nombre el nombre del servicio mensaje-service como un nombre de host DNS. El nombre del servicio se resuelve automáticamente con la dirección IP del servicio mediante el DNS interno de Kubernetes. Como se ha mencionado

anteriormente, el mensaje-service será el encargado de repartir los mensajes entre las réplicas de envio_mensaje.

Para permitir mandar las peticiones HTTP desde los pods del cluster hacia el IRIS GW se debe configurar el networking de Kubernetes, para ello se debe crear una NetworkPolicy que permita el tráfico saliente hacia cualquier dirección IP y puerto TCP fijo: 8088. En general, cuando los pods realizan peticiones salientes, utilizan la dirección IP del nodo en el que están programados como la dirección de origen en las conexiones salientes. Esto se debe a que los pods se ejecutan en el contexto del nodo y se comunican a través de la interfaz de red del nodo.

En el anexo E se encuentran todos los archivos de despliegue, servicio y configuración necesarios para el despliegue completo del sistema. Aquí se tiene un esquema del cluster final con todos los deployment y servicios.

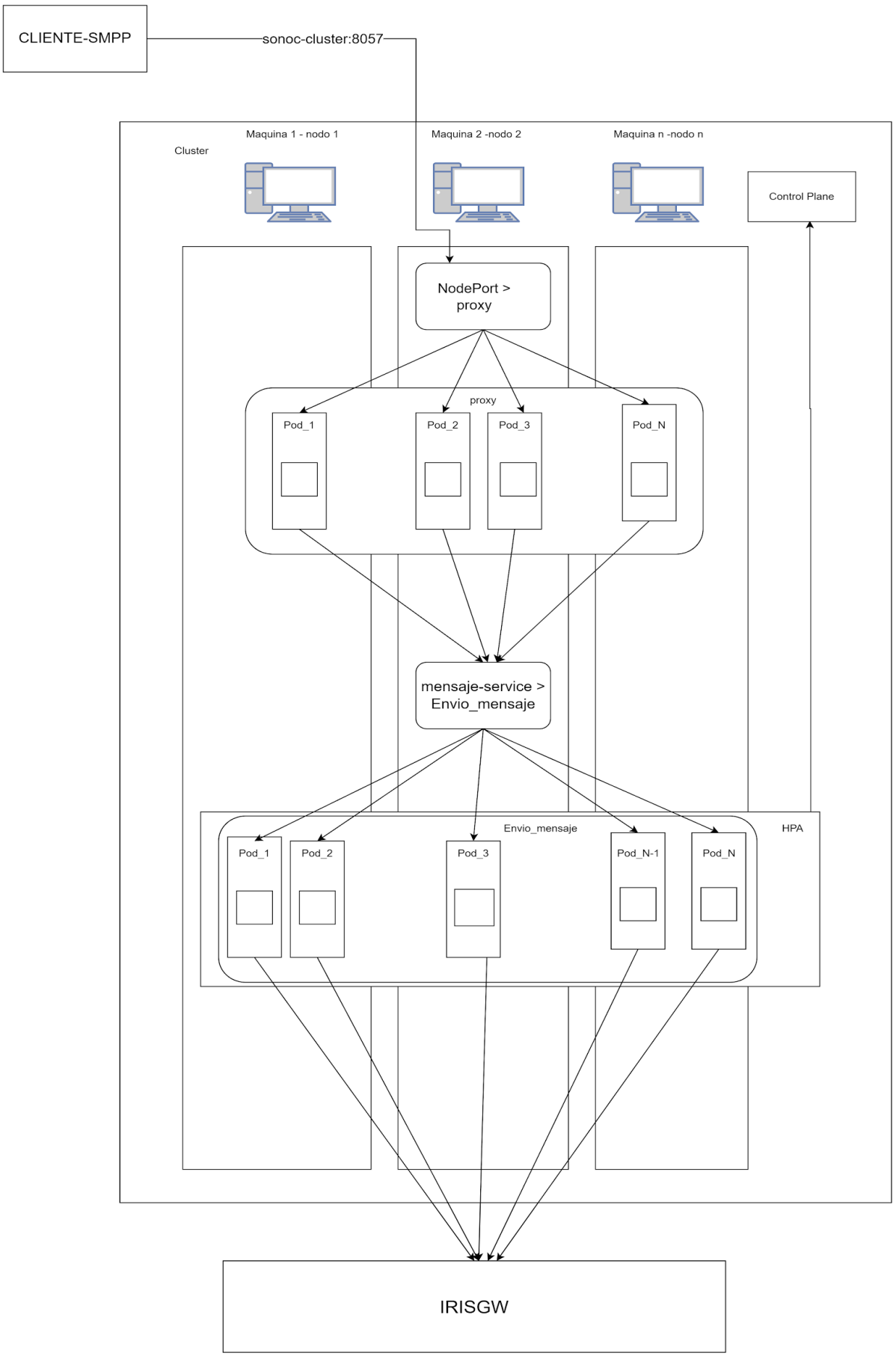
CLIENTE-SMPP
sonoc-cluster:8057
Cluster
Maquina 1 - nodo 1
Maquina 2 -nodo 2
Maquina n -nodo n
Control Plane
NodePort >
proxy
proxy
Pod_1
Pod_2
Pod_3
Pod_N
mensaje-service >
Envio_mensaje
Envio_mensaje
HPA
Pod_1
Pod_2
Pod_3
Pod_N-1
Pod_N
IRISGW

Figura 14. Despliegue de la solución

En la figura 14, se observa como el servicio nodePort recibe la petición de conexión externa del cliente, luego envía la petición a una de las instancias de la aplicación proxy que precederá a su autenticación. Si se ha completado el proceso correctamente se establecerá una conexión SMPP entre el cliente y el proxy de forma que se podrá empezar a enviar mensajes. Cuando el proxy recibe un mensaje del cliente, lo envía hacia el servicio envio_mensaje que se encarga de repartirlo entre alguna de las réplicas de envio_mensaje que se reducirán o crecerán en número según la carga de trabajo, esta aplicación se encarga de enviar una petición HTTP a partir del mensaje recibido hacía IRISGW para que sea enviada a los clientes finales. Si el mensaje llega al destinatario final se genera un DLR que se envía al cliente.

6. Pruebas realizadas

En el siguiente capítulo se han realizado las pruebas de comunicación y de carga de tráfico pertinentes sobre el escenario que se ha desarrollado.

6.1 Prueba de concepto

Se hizo una primera prueba que comprueba el correcto funcionamiento del sistema. La prueba consiste en un usuario capaz de iniciar una sesión SMPP y enviar mensajes SMS a través de ella, además se comprueba que ambos servicio realizan un balanceo de carga entre las réplicas disponibles y que la aplicación de envio_mensaje aumenta o disminuye el número de réplicas disponibles según la carga.

Se ha desarrollado un programa en Java que simula el comportamiento de un cliente SMPP con unas credenciales registradas en nuestra plataforma que desea enviar seis mensajes SMS a otro usuario final. En esta primera prueba no se activó el HPA por tanto se tienen 3 réplicas fijas para el servicio de envio_mensaje sin importar la carga de mensajes. En el anexo H se encuentran las pruebas realizadas a fondo.

Inicialmente, como se puede ver en la figura 15, 1.- el cliente se conecta a la IP del cluster 192.168.3.124 e intenta iniciar una sesión SMPP. 2.- Se envía la petición y el servicio NodePort se encarga de balancear la petición externa entre las réplicas de proxy. 3.-La aplicación proxy se encarga de comprobar que el usuario está registrado y tiene binds disponibles y permite establecer una conexión SMPP al cliente.

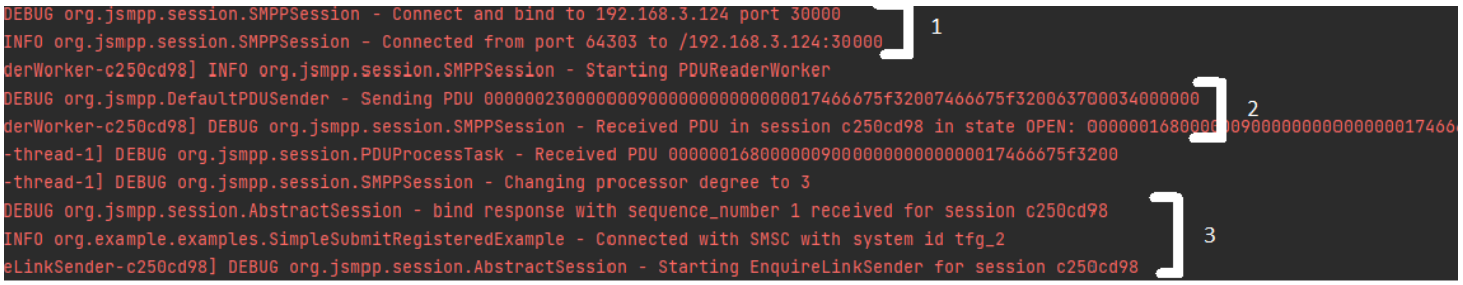

Figura 15. Prueba Concepto. Logs Cliente. Establecimiento de conexión por el cliente

En la figura 16 se puede observar cómo 1.- se recibe la conexión del cliente y se comprueban parámetros como nombre de usuario y contraseña. Posteriormente, 2.- se revisa el rateLimiter asociado a ese cliente y se añade ese bind entre las sesiones abiertas por el cliente. Finalmente 3.- se le permite iniciar una sesión como TRX.

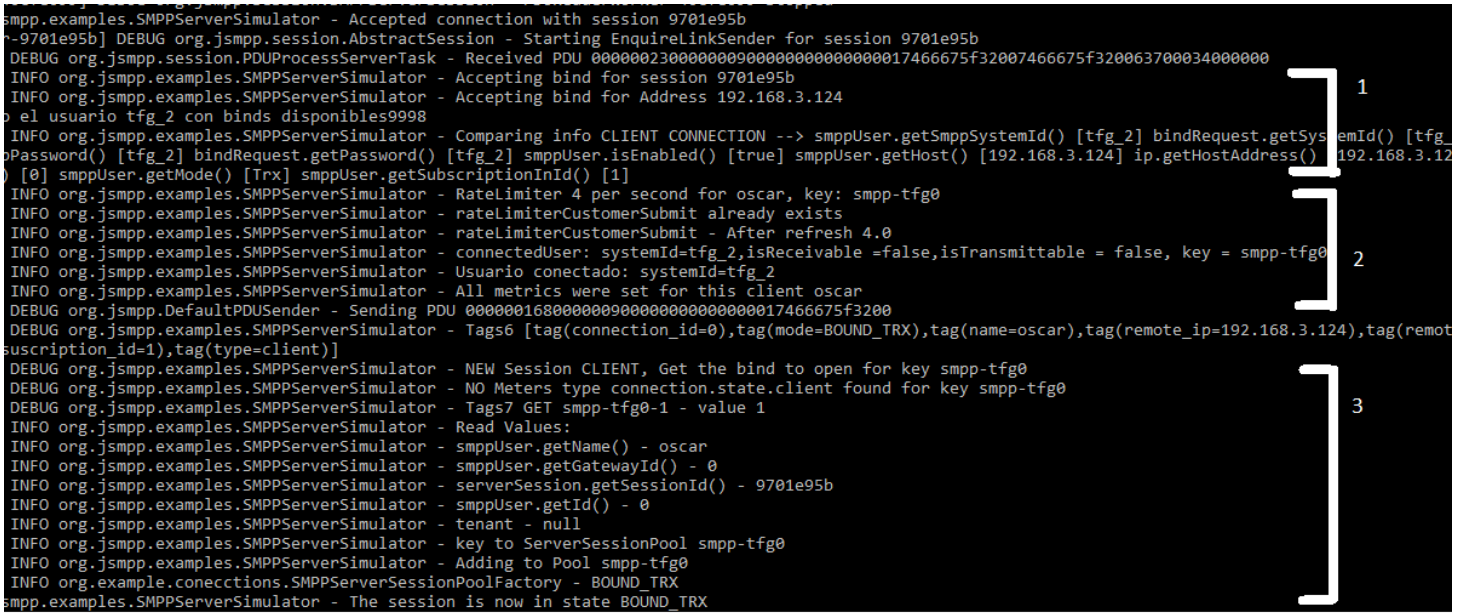

Figura 16. Prueba Concepto. Logs Proxy. Establecimiento de conexión

Posteriormente se ve en la figura 17 cómo 1.- el cliente manda el primer mensaje con id=de20f9c y luego 2.- recibe el DLR (delivery receipt) para el mensaje.

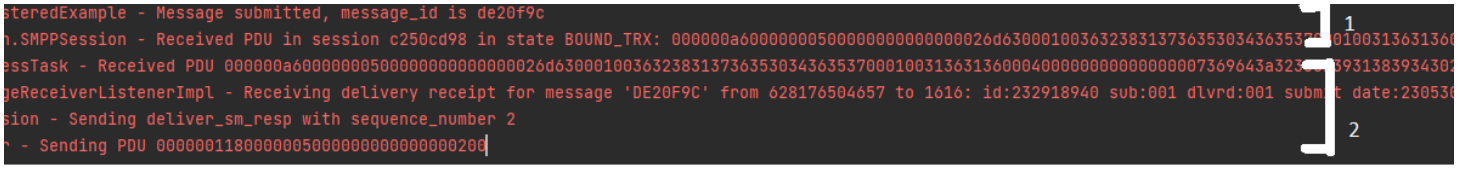

Figura 17. Prueba Concepto. Logs Cliente. Envío de mensaje y recepción DLR

En la figura 18 se puede ver cómo 1.- el proxy recibe el mensaje del cliente y le envía un submit_sm_resp para verificar que ha recibido el mensaje. Posteriormente, se envía a la aplicación envio_mensaje que se encargará de su envío a IRIS. 2.- Cuando se confirma la entrega el proxy envía el DLR correspondiente al primer mensaje con id=de20f9c.

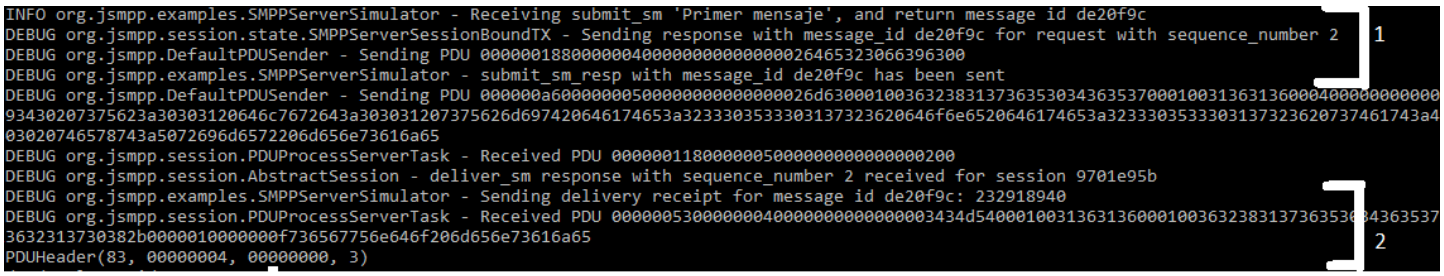

Figura 18. Prueba Concepto. Logs Proxy. Recepción de mensaje y envío de su correspondiente DLR

Para finalizar como se ven en las figuras 19 y 20 el cliente manda un paquete unbind para cerrar el bind con el proxy y terminar la conexión.

```
main] DEBUG org.jsmpp.session.AbstractSession - Unbind and close session c250cd98
main] DEBUG org.jsmpp.DefaultPDUSender - Sending PDU 00000010000000060000000000000007
PDUReaderWorker-c250cd98] DEBUG org.jsmpp.session.SMPPSession - Received PDU in session c250cd98 in state BOUND_TRX: 00000010800000060000000000000007
pool-1-thread-1] DEBUG org.jsmpp.session.PDUProcessTask - Received PDU 00000010800000060000000000000007
main] DEBUG org.jsmpp.session.AbstractSession - unbind response with sequence_number 7 received for session c250cd98
main] DEBUG org.jsmpp.session.AbstractSession - Close session c250cd98 in state BOUND_TRX
main] DEBUG org.jsmpp.session.AbstractSession - Stop enquireLinkSender for session c250cd98
EnquireLinkSender-c250cd98] DEBUG org.jsmpp.session.AbstractSession - EnquireLinkSender stopped for session c250cd98
main] DEBUG org.jsmpp.session.AbstractSession - Close session context c250cd98 in state BOUND_TRX
PDUReaderWorker-c250cd98] INFO org.jsmpp.session.SMPPSession - Reading PDU session c250cd98 in state CLOSED: Socket closed
PDUReaderWorker-c250cd98] DEBUG org.jsmpp.session.SMPPSession - PDUReaderWorker-c250cd98 stopped
```

Figura 19. Prueba Concepto. Logs Cliente. Finalización de conexión

```
INFO org.jsmpp.session.state.AbstractGenericSMPPSessionBound - Receiving unbind request
DEBUG org.jsmpp.DefaultPDUSender - Sending PDU 00000010800000060000000000000007
701e95b] INFO org.jsmpp.session.SMPPServerSession - Unbound session 9701e95b socket closed
701e95b] DEBUG org.jsmpp.session.AbstractSession - Close session 9701e95b in state UNBOUND
701e95b] DEBUG org.jsmpp.session.AbstractSession - Stop enquireLinkSender for session 9701e95b
-9701e95b] DEBUG org.jsmpp.session.AbstractSession - EnquireLinkSender stopped for session 9701e95b
701e95b] DEBUG org.jsmpp.session.AbstractSession - Close session context 9701e95b in state UNBOUND
701e95b] DEBUG org.jsmpp.session.SMPPServerSession - PDUReaderWorker-9701e95b stopped
```

Figura 20. Prueba Concepto. Logs Proxy. Finalización de conexión

Como se puede ver en la figura 21 las conexiones SMPP de los clientes se reparten entre las instancias por lo que se verifica que se están balanceando las peticiones entrantes entre las distintas réplicas del servicio.

```
oscar@LAPTOP-S43KSC93:~/despliegues$ kubectl logs proxy-deployment-6658d96589-whp94
[main] INFO org.jsmpp.examples.SMPPServerSimulator - Listening on port 8057
[main] INFO org.jsmpp.examples.SMPPServerSimulator - Accepted connection with session daaf1b83
[EnquireLinkSender-daaf1b83] DEBUG org.jsmpp.session.AbstractSession - Starting EnquireLinkSender for session daaf1b83
[pool-5-thread-1] DEBUG org.jsmpp.session.PDUProcessServerTask - Received PDU 0000002300000009000000000000000017466675f32007466675f320063700034000000
[pool-2-thread-1] INFO org.jsmpp.examples.SMPPServerSimulator - Accepting bind for session daaf1b83
[pool-2-thread-1] INFO org.jsmpp.examples.SMPPServerSimulator - Accepting bind for Address 192.168.3.124
```

```
oscar@LAPTOP-S43KSC93:~/despliegues$ kubectl logs proxy-deployment-6658d96589-zwzvd
[main] INFO org.jsmpp.examples.SMPPServerSimulator - Listening on port 8057
[main] INFO org.jsmpp.examples.SMPPServerSimulator - Accepted connection with session 3ef4d74a
[EnquireLinkSender-3ef4d74a] DEBUG org.jsmpp.session.AbstractSession - Starting EnquireLinkSender for session 3ef4d74a
[pool-5-thread-1] DEBUG org.jsmpp.session.PDUProcessServerTask - Received PDU 0000002300000009000000000000000017466675f32007466675f320063700034000000
[pool-2-thread-1] INFO org.jsmpp.examples.SMPPServerSimulator - Accepting bind for session 3ef4d74a
[pool-2-thread-1] INFO org.jsmpp.examples.SMPPServerSimulator - Accepting bind for Address 192.168.3.124
```

Figura 21. Prueba Concepto. Demostración Reparto de los mensajes entre réplicas de proxy

En la figura 22 se recogen los logs de una réplica de la aplicación envio_mensaje previos al envío de mensajes SMS por un cliente, en la parte inferior aparecen los logs posteriores

donde se observa como solo ha enviado dos mensajes de los seis que envía cada cliente por tanto se concluye que el resto de esos mensajes han sido enviados por otra replica de envio_mensaje. Esto significa que se hace un reparto equitativo de los mensajes enviados entre las réplicas del servicio.

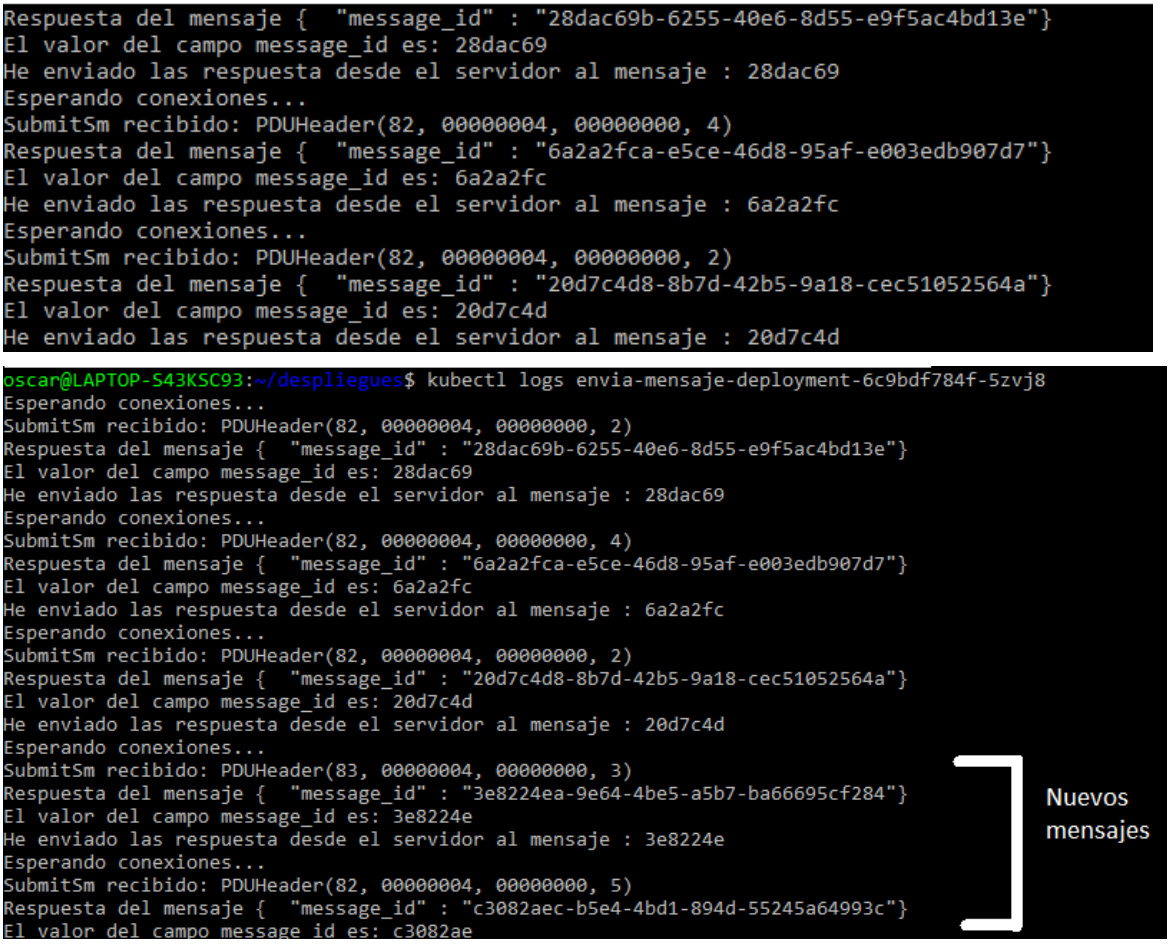

Figura 22. Prueba Concepto. Envío de mensajes al IRISGW y posterior envío del DLR al proxy

A continuación se va analizar el comportamiento del componente HPA en el sistema. Como se puede ver cuando se activa el HPA según la carga de trabajo aumentan o disminuyen el número de réplicas de los pods envio_mensaje. Inicialmente se tienen 3 réplicas que había previamente. Como se ha visto antes en la configuración el mínimo de réplicas es 1 y el maximo 10

Como se ha mencionado anteriormente en esta prueba los clientes únicamente envían seis mensajes por lo que la carga de trabajo es reducida y HPA reduce el número el número de réplicas hasta quedarse con el mínimo como se observa en la figura 23.

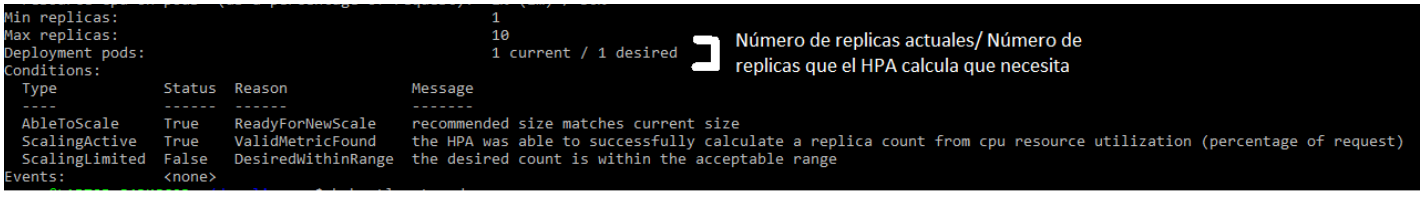

Figura 23. Prueba Concepto. Reducción del número de réplicas en HPA

Para poder comprobar completamente el correcto funcionamiento del HPA ha sido necesario aumentar el número de mensajes que se envían desde los clientes. El HPA detecta que se necesitan más réplicas y procede a su creación, incluso se puede ver que supera el límite configurado y pasa a estado TooManyReplicas que indica que ha necesitado superar el límite de réplicas establecidas.

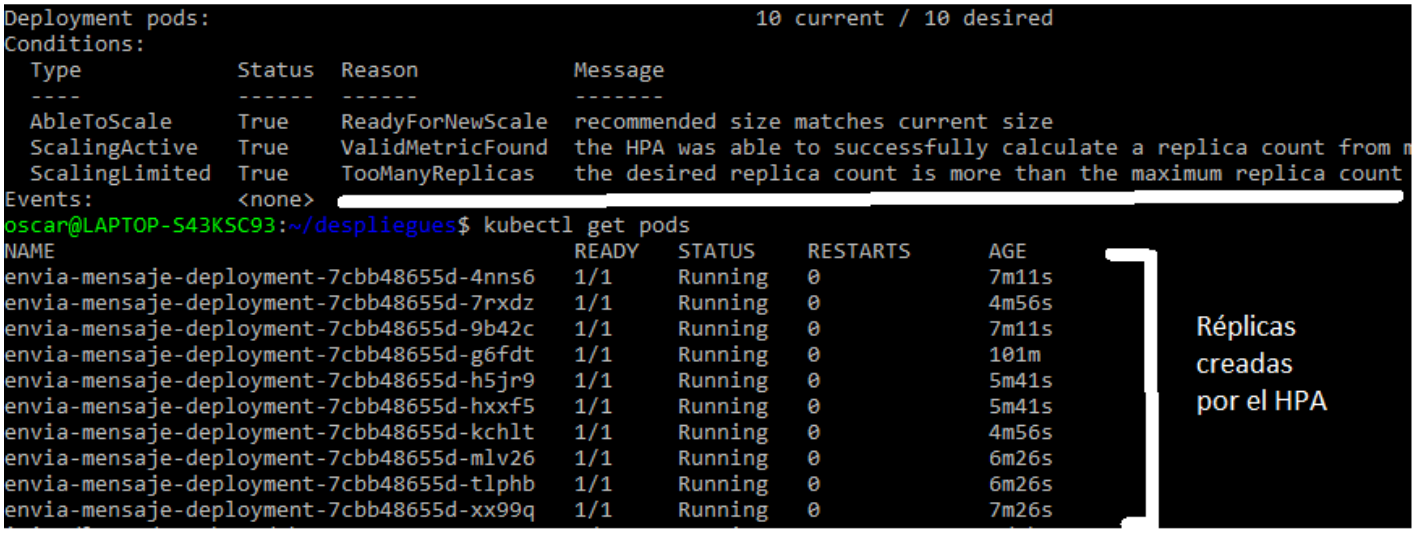

Figura 24. Prueba Concepto. Escalado de réplicas en HPA

6.2 Prueba de carga

Se han realizado pruebas de carga del sistema donde se han enviado simultáneamente 10, 100 y 1000 mensajes SMS por parte de un cliente. En estas pruebas hemos evaluado el tiempo total necesario para el envío de los mensajes que nos indica la capacidad de dar servicio a los clientes y el tiempo de delay de cada mensaje individualmente que nos permite analizar si hay mensajes que han tenido un tiempo de envío elevado. El tiempo de delay es acumulativo por tanto en cada prueba realizada el último mensaje en ser procesado será el que tenga un mayor delay puesto que transcurre más tiempo en ser enviado.

Para esta prueba se ha desarrollado un programa que simula el envío masivo de mensajes SMS de forma simultánea. Se utilizó como base el código disponible en https://github.com/opentelecoms-org/jsmpp/blob/master/jsmpp-examples/src/main/java/org/jsmpp/examples/StressClient.java.

Como es lógico en cada ejecución de las pruebas se pueden obtener resultados. Por ello, cada prueba de envío de 10, 100 y 1000 se realizó 50 veces, calculándose una media de los resultados obtenidos. A continuación se muestra una tabla comparativa:

Mensajes enviados	Tiempo total(ms)	Delay(ms)	Número de réplicas
10	2020	1135	10

100	8070	7223	10
100	4063	2154	20
1000	6065	5266	20

En el anexo I se recoge a fondo las pruebas realizadas.

Durante las pruebas de envío simultáneo de 100 mensajes SMS, el sistema ha necesitado las 10 réplicas disponibles del servicio envio_mensaje. El tiempo necesario para que se detecte el aumento del tráfico y se levanten las réplicas necesarias provoca una pequeña acumulación en los mensajes pero se resuelve adecuadamente. Posteriormente, ya que se utilizaron el total de las réplicas, se decidió aumentar el número de réplicas disponibles de 10 a 20 para una nueva prueba. Se analizó si un aumento del número de réplicas se traduce en una reducción del tiempo necesario para el envío de los mensajes o en una reducción en el delay de los mensajes de forma individual. Como se aprecia en la tabla se consigue una reducción considerable tanto en el tiempo total requerido, como en el delay de los mensajes. La reducción significativa del máximo delay se debe a que, como se ha mencionado, el tiempo de delay es acumulativo y con 20 réplicas ninguno de los mensajes tiene que esperar a que el módulo envíe previamente otros mensajes, como sí ocurre cuando se tienen menos réplicas disponibles.

En el envío de 1000 mensajes no se produce un aumento en el tiempo total proporcional al número de mensajes enviados respecto al envío de 100 mensajes, esto se debe a que el sistema está orientado al envío masivo de mensajes y hay más que ganar en el procesado de mensajes que en la gestión de los binds.

Durante el proceso de pruebas también se probaron tres clientes de forma simultánea de forma que cada cliente era atendido por una instancia del proxy. Se demostró que era mejor que un cliente inicie de forma simultánea 3 sesiones y enviar 100 mensajes a través de cada sesión que únicamente abrir una sesión y envíar 300 mensajes a través de ella. Esto es debido a que a pesar de que la carga de trabajo es la misma, el proceso de recibir los mensajes, enviarlos al servicio de envio_mensaje y el envío de los DLR se realiza concurrentemente entre las 3 réplicas y no de forma secuencial. Esto era de esperar ya que como explicamos en el capítulo 4 los cliente contratan con SONOC un número de sesiones disponibles y, por tanto, contratar un número mayor se traduce en una mayor capacidad de envío simultáneo de mensajes.

7. Conclusiones y líneas futuras

7.1 Conclusiones

Vamos a resumir lo aprendido en el presente trabajo:

En primer lugar, se ha desarrollado un módulo balanceador de carga para el protocolo de SMPP que se divide en dos aplicaciones encargadas de distintas funcionalidades; una de ellas para procesar los mensajes SMS para optimizar su envío y la otra para gestionar las conexiones SMPP de los clientes. Esto ha requerido estudiar y comprender el protocolo SMPP y que funcionalidades de la biblioteca JSMPP eran necesarias para el trabajo. También ha sido clave entender el escenario previo al trabajo para ser capaces de integrar las nuevas funcionalidades de forma que se obtuviesen los mejores resultados para los clientes.

A continuación, se ha pasado a la implementación del escenario desarrollado en un entorno de Kubernetes. Esto ha afianzado los conocimientos obtenidos de las aplicaciones anteriormente mencionadas, pues se ha sido capaz de configurarlas en distintos entornos para ofrecer la misma funcionalidad con una mejora en la fiabilidad y disponibilidad. Por supuesto, el mayor conocimiento que se ha obtenido ha sido en el despliegue de aplicaciones en Kubernetes, para lo cual se ha realizado un estudio de los diferentes elementos de un clúster, se han valorado y se ha 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, es importante destacar que este trabajo ha sido el primer paso para evolucionar de la arquitectura monolítica que existe actualmente en IRIS hacia una arquitectura distribuida que permita una mejora en el rendimiento y en la disponibilidad del sistema. Este trabajo ha abordado la primera fase que ha comprendido la gestión de las conexiones de los clientes y el procesamiento de los mensajes SMS a enviar.

7.2 Líneas futuras

La mejora más clara para nuestro sistema es eliminar el componente IRISGW, de forma que sean los módulos de envio_mensaje los que en vez de mandar una petición HTTP contra IRISGW manden el mensaje a Kafka, de esta forma mejoramos el tiempo de envío y procesamiento del mensaje además de hacer el sistema mucho más escalable al eliminar un cuello de botella en la capacidad de procesamiento de IRISGW.

Actualmente los datos que usa la aplicación para la comprobación y gestión de los cliente se insertan estáticamente en el código, una implementación futura y para la que el código ya está preparado sería que la aplicación consumiera la información de una cache Redis que se actualizase de forma dinámica la información y pudiese ser modificada también por el resto de módulos de IRIS.

Otra línea de desarrollo futuro es la relacionada con el despliegue del sistema. Actualmente se está desplegando en único cluster para realizar las pruebas, sin embargo puesto que SONOC tiene múltiples clientes alrededor del mundo es necesario que se despliegue el sistemas en múltiples cluster por todo el mundo de forma que todos los usuarios tengan buena calidad de servicio.

Referencias

GSMA [BAN] https://bandaancha.eu/articulos/vodafone-finiquita-mensajes-chat-rcs-10475 consultado en 29/05/2023

HTTP [HTT] https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto consultado en 29/05/23

SMPP [SMM] https://smpp.org/ consultado en 29/05/2023

Docker [DOC] https://docs.docker.com/ consultado en 29/05/2023

Kubernetes [KUB] https://kubernetes.io/docs/home/ consultado en 29/05/2023