Universidad Zaragoza

Proyecto Fin de Carrera

Evaluación y aplicación de mejoras en red virtual de sistema distribuido en producción

Autor

Jaime Soriano Pastor

Directores

Javier Camuñas Velasco
Mario Rodríguez Molins

Ponente

José Luis Briz Velasco

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
2017

Evaluación y aplicación de mejoras en red virtual de sistema distribuido en producción

RESUMEN

La arquitectura de microservicios ofrece una serie de mejoras en los procesos de desarrollo de aplicaciones complejas, permite aislar mucho más los diferentes componentes y limitar sus diferentes responsabilidades. Pero su gestión plantea nuevos desafíos. La popularización de esta arquitectura ha ido de la mano de nuevos desarrollos de código abierto que facilitan su implantación, como Kubernetes, un sistema distribuido para la automatización, escalado y gestión de aplicaciones. Los despliegues de estas soluciones, por su naturaleza distribuida y dinámica añaden nuevos niveles de complejidad.

En este proyecto se abordan una serie de problemas encontrados en la red de un cluster de Kubernetes con carga real de producción y se desarrollan y evalúan soluciones y mejoras para mitigarlos. Para ello se estudia a fondo el sistema, poniendo especial hincapié en los componentes más cercanos a la red, buscando cuellos de botella que puedan afectar a la escalabilidad y estabilidad del sistema.

La búsqueda y aplicación de mejoras se centra principalmente en dos partes. En los balanceadores los mayores desafíos residen en que soportan todo el tráfico externo del sistema y en que necesitan reconfigurarse frecuentemente respondiendo a los cambios en el clúster. Se introducen mejoras para minimizar los problemas causados por las recargas y optimizaciones para poder manejar una mayor cantidad de tráfico. En la red que interconecta los servicios se analiza el impacto del uso de encapsulación VXLAN y se desarrolla una solución para dejar de usarla sin necesidad de parar la plataforma.

En varios puntos del proyecto se realizan pruebas de carga de modo que se pueda evaluar de forma objetiva la aportación de cada una de las mejoras propuestas.

Índice

1 Introducción....5
1.1 Motivación y objetivo general....5
1.2 Contexto del trabajo....5
1.3 Objetivos específicos....7
1.4 Principales resultados....7
1.5 Planificación del PFC....8
2 Fundamentos....9
2.1 Arquitectura de *Kubernetes*....9
2.1.1 Modelo de datos....9
2.1.2 Agentes....10
2.1.3 Red....11
2.1.4 Arquitecturas....13
2.2 flannel....15
2.2.1 VXLAN....16
2.2.2 Host gateway....17
3 Desarrollo del proyecto....19
3.1 Análisis del despliegue inicial....19
3.1.1 Identificación de problemas....21
3.1.2 Evaluación del rendimiento y situación inicial....26
3.2 Mejoras en los *kubelbs*....28
3.2.1 Reducción de reglas de *iptables*....28
3.2.2 Nuevos límites de conexiones y sesiones activas....29
3.2.3 Mejoras en las recargas de configuración....31
3.2.4 Evaluación del rendimiento tras mejoras en *kubelbs*....36
3.3 Reemplazo del sistema de enrutado....37
3.3.1 Comparativa de VXLAN frente a *Host gateway*....37
3.3.2 Desarrollo de herramienta de migración....37
3.3.3 Migración de VXLAN a *Host gateway*....43

3.3.4 Evaluación del rendimiento tras cambios en el sistema de enrutado..................44
4 Metodología..45
5 Conclusiones...46
Anexo I: Ejecuciones de pruebas de carga...49

1 Introducción

1.1 Motivación y objetivo general

El desarrollo de este proyecto pretende abordar la mitigación o resolución de una serie de problemas que se han ido identificando en un sistema en explotación en la compañía Tuenti[1] y que han afectado en repetidas ocasiones a la disponibilidad de servicios ofrecidos a clientes y revendedores.

Estos problemas han sido principalmente causados por limitaciones de capacidad originadas por el diseño de la red, así como en problemas de configuración de ésta. Por tanto, el desarrollo del proyecto consiste en la evaluación y aplicación de optimizaciones y mejoras en esta red.

Es difícil evaluar la repercusión económica total causada por estos problemas, pero si nos remitimos tan sólo a los informes de incidencias en servicios en producción desde diciembre de 2016 a marzo de 2017, se puede ver que al menos cinco de ellos hacen referencia directa a problemas en esta red, suponiendo un total de casi 11 horas de afectación, varias de ellas con caídas totales del sistema. Estas cifras evidencian la severidad de los problemas y la necesidad de optimizaciones en la configuración y cambios en el diseño.

1.2 Contexto del trabajo

Este proyecto se realiza como parte de un plan de mejoras y estabilización de los sistemas en producción de Tuenti, para aproximarnos a este contexto hemos de hacer un breve recorrido histórico por esta compañía, centrándonos en los cambios tecnológicos realizados en los últimos años.

Tuenti nace en 2006 como una red social orientada al público joven, estaba implementada como un único proyecto monolítico en PHP. Entonces, y durante varios años, todas las herramientas, sistemas y flujos de trabajo se correspondían con los adecuados para este tipo de aplicación. Conforme el desarrollo se va consolidando, se observan deficiencias en el modelo de proyecto único y se comienza a migrar hacia una arquitectura orientada a servicios, de modo que se puedan aplicar cambios a partes del sistema sin requerir para ello actualizaciones completas. Durante estos años también van apareciendo otros desarrollos que introducen diferentes tecnologías: aplicaciones móviles para diferentes sistemas operativos, una plataforma de chat implementada en *erlang*[2] y algunos servicios empiezan a implementarse en Java. También durante estos años se comienza a crear una nueva área de negocio en la compañía, la de operador móvil virtual, con unos requerimientos diferentes a los de un sitio web con alto tráfico y a la que actualmente se dedican la práctica totalidad de recursos, con operaciones en España y en varios países de América Latina.

1 Tuenti Technologies S.L. es una compañía tecnológica española que actualmente forma parte del grupo Telefónica.
2 *erlang* es un lenguaje de programación orientado a concurrencia creado por Ericsson para la implementación de aplicaciones distribuidas con alta tolerancia a fallos en el entorno de las telecomunicaciones.

En resumen, quedándonos más en el punto de vista técnico, la compañía pasa de centrarse en un único proyecto, y un único *stack* tecnológico, a trabajar con todo un abanico de tecnologías. Esto complica las operaciones, los equipos encargados de los sistemas necesitan conocer y satisfacer los requerimientos de cada plataforma y su mantenimiento resulta cada vez más complejo, incluso proyectos basados en las mismas tecnologías pueden necesitar en un momento dado versiones diferentes del mismo software. Aparecen barreras de conocimiento; por un lado los equipos al cargo de la infraestructura no pueden tener un conocimiento tan profundo de todas las tecnologías y necesitan cada vez herramientas más complejas para la automatización y orquestación de los sistemas, por otro lado, los equipos de desarrollo conocen las plataformas, pero se encuentran con que para poder llevar ese conocimiento a la infraestructura necesitan conocer a fondo las herramientas de orquestación. La compañía no podía escalar operativamente.

A mediados de 2015 se crea un *squad*[3] para intentar solucionar estos nuevos problemas, se lleva a cabo un proceso de análisis de los métodos de trabajo de otras compañías con problemáticas similares y de tecnologías que puedan servir para agilizar los procesos. Como conclusión de este *squad*, se toman principalmente las siguientes decisiones:

- Migración de la arquitectura orientada a servicios hacia una arquitectura de “microservicios”[4].
- Elección de *Kubernetes* (Sec. 2.1) como plataforma de referencia para el despliegue de estos microservicios.

A principios de 2016 se despliega el primer clúster de *Kubernetes* en la compañía, y a lo largo del año se van migrando servicios. A principios de 2017 la arquitectura se compone de unos 90 servicios, casi todos ellos desplegados en *Kubernetes*. La utilización de este sistema puede considerarse un éxito a efectos de reducción de la carga operacional de los equipos, pero está planteando nuevos desafíos en cuanto a escalabilidad y fiabilidad de la plataforma, especialmente a nivel de red.

Mi relación con este proyecto comienza en 2011. En junio comienzo a trabajar en Tuenti, inicialmente en los entornos de desarrollo y pruebas, y a partir de 2014 tomo responsabilidades también en los sistemas de producción. En 2015 formo parte del *squad* que decide la migración a microservicios y desde entonces dedico la mayor parte de mi tiempo al desarrollo y mantenimiento de los despliegues de *Kubernetes*.

3 En Tuenti, un *squad* es un equipo temporal creado para abordar un problema concreto.
4 No hay una definición clara de lo que supone una arquitectura de microservicios, como características generalmente aceptadas [1] sería una arquitectura en la que cada componente es un servicio independiente con unas responsabilidades limitadas, sin estado, cada uno con su base de datos y cada uno ofreciendo una interfaz a otros servicios utilizando protocolos estándar como REST sobre HTTP o gRPC.

1.3 Objetivos específicos

El objetivo de este proyecto es la optimización del rendimiento de la red utilizada en *Kubernetes*, con el objetivo final de trasladar el cuello de botella de la escalabilidad del sistema de este punto a los recursos (especialmente CPU y memoria) de los nodos, permitiendo escalar horizontalmente y evitando las limitaciones de capacidad causadas por la red.

Las áreas de actuación con este fin se pueden agrupar en estos tres puntos:

- Optimización de los ajustes de los elementos encargados del balanceo de tráfico entre nodos de ejecución.
- Optimización de los ajustes de red del sistema operativo Linux de los nodos de ejecución.
- Reemplazo del sistema de enrutado entre servicios del clúster.

Estos cambios se van a realizar en un sistema en producción, por lo que su aplicación ha de poder realizarse de forma progresiva, implicando esto que los cambios deben ser compatibles con los ajustes previos y deben de poder ser reversibles. En el caso del sistema de enrutado implica también el desarrollo de un software que mantenga automáticamente la conectividad entre servicios con ambos sistemas.

1.4 Principales resultados

Como resultado de las diferentes tareas realizadas como parte de este proyecto hemos conseguido proporcionar diversos beneficios a los clústers de *Kubernetes* de producción de Tuenti. En general hemos conseguido identificar y eliminar todos los cuellos de botella causados por la red permitiendo escalar horizontalmente y hacer un mejor uso de los recursos. Esto ha permitido también tener una mejor visibilidad de la capacidad total de los clústers en general y de los nodos en particular, al no estar limitada más que por los propios recursos de los nodos.

Más concretamente, se han realizado mejoras en los balanceadores de tráfico: se han optimizando algunos parámetros para mejorar la resiliencia de la plataforma ante algunos problemas, se han eliminado funciones de red innecesarias y se ha modificado el sistema de recargas de configuración para evitar pérdidas de paquetes durante los reinicios. También se han diseñado y desplegado como parte de este proyecto escenarios de pruebas de carga más ajustados a la realidad, lo que permite evaluar de una forma más objetiva los cambios futuros que puedan presentarse.

Finalmente se han identificado problemas de rendimiento en la la red virtual de *Kubernetes*, se han evaluado alternativas y se ha desarrollado un plan de migración del sistema que puede realizarse sin afectar a los servicios en producción.

1.5 Planificación del PFC

El proyecto consta de las siguientes fases en orden cronológico. Las fechas concretas dependen en gran medida de las prioridades de la empresa, por lo que no se detallan (más sobre la metodología de trabajo en Sec. 4).

1. Medición de capacidad actual
2. Desarrollo del proyecto
    1. Balanceadores de tráfico
        1. Implementación de mejoras en los balanceadores de tráfico
        2. Evaluación de las mejoras aplicadas
    2. Sistema de enrutado
        1. Comparativa de rendimiento de VXLAN frente a otras tecnologías
        2. Selección de tecnología de enrutado
        3. Implementación de herramienta de migración
        4. Migración de tecnología de enrutado
3. Medición de capacidad final y obtención de conclusiones

2 Fundamentos

2.1 Arquitectura de *Kubernetes*

Kubernetes es un sistema distribuido de código abierto para la automatización del despliegue, escalado y gestión de aplicaciones en contenedores. Se fundamenta en un modelo de datos que define el estado deseado de las aplicaciones, y en un conjunto de agentes que llevan a cabo las tareas necesarias para que el estado real se corresponda con el estado definido [2].

2.1.1 Modelo de datos

El modelo de datos traslada a estructuras de datos bien definidas la mayoría de conceptos fundamentales del sistema. Todos los datos representados en este modelo de datos tienen algunas características comunes:

- Siempre pueden representarse en formato JSON, facilitando su almacenamiento y transmisión. En versiones más recientes se utiliza también *Protocol Buffers*[5] para almacenar y transmitir estos datos, pero una estructura de datos que puede ser codificada como *Protocol Buffers*, puede también serlo como JSON.
- Como parte de su estructura siempre tienen unos metadatos comunes, entre ellos el nombre, el *namespace* al que pertenecen, etiquetas y anotaciones. En un mismo clúster de *Kubernetes* no puede haber dos objetos del mismo tipo con el mismo nombre en el mismo *namespace*.

A modo de ejemplo y como referencia, veamos algunos de los recursos que pueden representarse en este modelo de datos y que usaremos también a lo largo de esta memoria:

- *Pod*: Unidad mínima de ejecución de aplicaciones en *Kubernetes*, sería equivalente al concepto de proceso en un sistema operativo. La definición de un *pod* incluye una lista de contenedores Linux a ejecutar, así como los recursos solicitados al sistema, las cuotas máximas y los puertos expuestos.
- Servicio: Recurso utilizado para representar una aplicación que se ha de exponer externamente o hacia otros *pods*. Básicamente contiene una serie de puertos expuestos, un mapeo entre estos puertos y los puertos de los *pods* que implementan el servicio y un selector de *pods*. El selector consiste en una lista de etiquetas que se utilizan para elegir los *pods* que implementan el servicio. Existen varios tipos de servicios dependiendo de cómo se expone.
- Nodo: Cuando una máquina se une al clúster para ejecutar *pods*, se le llama nodo, y también tiene su representación en este modelo de datos. Un nodo tendrá la información de red de la

5 Se llama *Protocol Buffers* a un mecanismo agnóstico del lenguaje para serializar datos estructurados definido y desarrollado por Google [3].

máquina y datos sobre su estado, los recursos que tiene disponibles, los *pods* que tiene asignados, etc.

- Controladores: Los controladores son un conjunto de tipos de recursos que se encargan de la gestión automatizada de otros recursos, probablemente el más utilizado es el *Deployment*[6], que declara la cantidad de réplicas de un *pod* que deben existir en todo momento.

2.1.2 Agentes

En *Kubernetes* cada agente tiene unas responsabilidades limitadas y bien conocidas [4]. Cada funcionalidad proporcionada por un clúster viene dada por estos agentes, lo que da una gran flexibilidad en cuanto a poder modificar o ampliar las funcionalidades. Una característica importante de estos agentes es que están pensados como operadores, es decir, automatizan operaciones en las máquinas pero no son necesarios para el correcto funcionamiento de las aplicaciones desplegadas, de modo que por ejemplo podrían pararse sin afectar directamente a las aplicaciones hasta que una operación fuera necesaria.

Los agentes que forman parte del sistema distribuido son los siguientes:

- *etcd* es la única base de datos soportada actualmente por *Kubernetes*, es una base de datos clave/valor distribuida y se utiliza para mantener el estado del clúster utilizando el modelo de datos descrito anteriormente.

- El *API server* es un servidor HTTP que se encarga de exponer las APIs que soportan las operaciones sobre los recursos del clúster. Es el único agente del sistema que puede acceder a la base de datos, y por tanto, siempre que se quiera modificar el estado del clúster se hará a través de este agente. Tanto los clientes utilizados por operadores humanos, como el resto de agentes, interactúan con el clúster a través del *API server*. Puede haber varios por razones de capacidad y alta disponibilidad y disponen de mecanismos para coordinarse y evitar conflictos.

- El *Controller Manager* es el encargado de ordenar las acciones necesarias para que el estado real se corresponda con el estado deseado. Por ejemplo si se aumenta la cantidad de réplicas de un *deployment*, este agente será el encargado de solicitar la creación de nuevos *pods*. También es el encargado vigilar el estado de los nodos y de desalojarlos si tienen algún problema. Sólo puede haber uno de estos agentes en cada clúster, en el caso de haber varios, elegirán a un líder que será el único que realizará cambios.

- El *Scheduler* asigna pods a nodos. Cuando un operador o un agente ordenan la creación de un *pod*, lo que hacen es simplemente crear un recurso de tipo *pod*, pero eso no lanza los contenedores en ningún nodo. Este agente es el encargado de buscar un nodo que se ajuste a

6 La funcionalidad de los *Deployments* la ofrecían en las primeras versiones de *Kubernetes* los *Replication Controllers*.

los requisitos de ejecución del *pod* y asignárselo para que ejecute los contenedores. Al igual que con el *Controller Manager*, solo puede haber un *Scheduler* en cada clúster.

- *Kubelet* es el agente encargado de la gestión de cada uno de los nodos de ejecución. En cada nodo se ejecuta uno de estos agentes. Tiene varias responsabilidades:
  - Gestiona todo el ciclo de vida de los *pods* que le ha asignado el *Scheduler*: Descarga las imágenes de los contenedores a ejecutar, los ejecuta, configura sus recursos (límites, permisos, volúmenes, red...) y los finaliza cuando son borrados. También ejecuta los chequeos de salud de los contenedores si tienen alguno definido.
  - Registra el nodo en el clúster y mantiene actualizado su estado de salud.
  - Realiza tareas de manutención en el nodo, como la eliminación de imágenes y archivos locales que ya no son necesarios para los contenedores que haya ejecutado.
- *Kube-proxy* es otro proceso que ha de ejecutarse en cada nodo. Se encarga de configurar la conectividad con cada servicio como veremos más adelante (Sec. 2.1.3).
- *Kubedns* es un servidor DNS que se configura automáticamente de acuerdo a los servicios desplegados en el clúster y que puede ser utilizado para el autodescubrimiento de servicios.
- *Kube2lb*[7] es un agente desarrollado en Tuenti que automatiza la configuración de balanceadores de carga HTTP y TCP en función del estado del clúster.

2.1.3 Red

El trabajo realizado en este proyecto se centra especialmente en elementos de la red del clúster, veamos entonces cuáles son estos elementos y las redes involucradas.

En *Kubernetes* cada *pod* se ejecuta en un *namespace* de red diferente, esto quiere decir que cada uno de ellos va a tener su propia configuración de red. Uno de los requisitos fundamentales de *Kubernetes* es que todos los *pods* que se ejecuten en un clúster tengan una IP única y accesible para el resto de *pods* independientemente del nodo en el que se ejecuten [6]. Este requisito hace que por un lado se tenga que gestionar la lista de IPs utilizadas y disponibles, y por otro que se tenga que enrutar de algún modo el tráfico entre estas IPs entre distintos nodos. Existe software especializado en este tipo de operaciones, en Tuenti usamos *flannel* (Sec. 2.2).

Otro rango de red importante en *Kubernetes* es el de los servicios. Cuando se define un servicio en *Kubernetes*, automáticamente se le asigna una IP única, esta IP realmente no se configura en ninguna interfaz de red, pero es utilizada por elementos de enrutado como *kube-proxy* para dirigir el tráfico hacia los *pods* que implementan el servicio, *kube-proxy* lo hace configurando reglas NAT[8] con *iptables*[9]. Dependiendo del tipo de servicio, además de una IP, se le asigna un puerto aleatorio,

7 El código y la documentación de uso de *kube2lb* están disponibles públicamente en [5].

8 NAT – Network address translation (Traducción de direcciones de red) es un método para reemplazar direcciones y/o puertos en un paquete TCP, UDP o IP [7].

9 *iptables* es una herramienta de espacio de usuario para gestionar las tablas del firewall del kernel Linux

este puerto es conocido como *node port* y es un puerto que se expone en todos los nodos para poder acceder a los *pods* desde fuera del clúster.

En un clúster de *Kubernetes* tenemos entonces tres redes independientes a tener en cuenta:

- La red física en la que están las máquinas utilizadas como nodos de ejecución.
- La red de servicios.
- La red de *pods*, o red interna.

En la Figura 1 podemos ver el recorrido de una conexión desde un *pod* a otro *pod* que implementa un servicio:

1. Un *pod* realiza una conexión a una IP de servicio.
2. Todas las conexiones de un *pod* son enmascaradas por el nodo en el que se ejecuta, una regla NAT configurada por *kube-proxy* en el nodo reenvía la conexión a uno de los *pods* que implementan el servicio elegido aleatoriamente.
3. Si el *pod* está en otro nodo, la red virtual configurada por *flannel* encapsula la conexión hasta ese nodo.

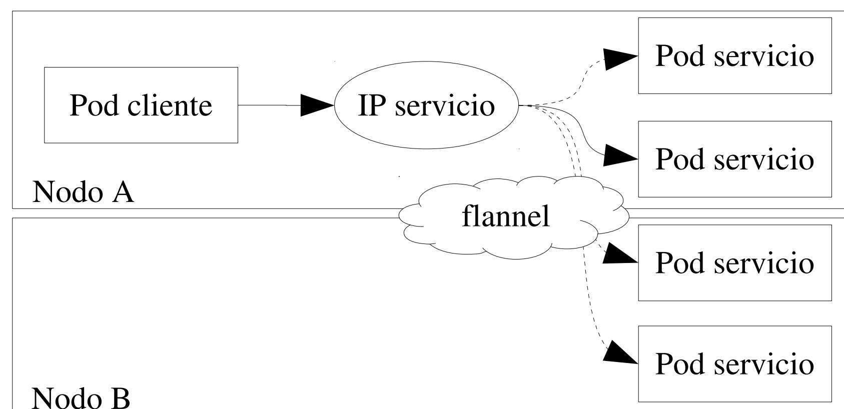

Figura 1: Conexión desde un pod a otro a través de una IP de servicio

Y como podemos ver en la Figura 2, para que una conexión llegue desde fuera del clúster a un pod utilizando un *node port*, ha de pasar por los siguientes puntos:

1. Un cliente realiza una conexión a uno de los nodos al *node port* de uno de los servicios.
2. Una regla NAT configurada por *kube-proxy* reenvía la conexión a uno de los *pods* que implementan el servicio elegido aleatoriamente.
3. Si el *pod* está en otro nodo, la red configurada por *flannel* encapsula la conexión hasta ese nodo.

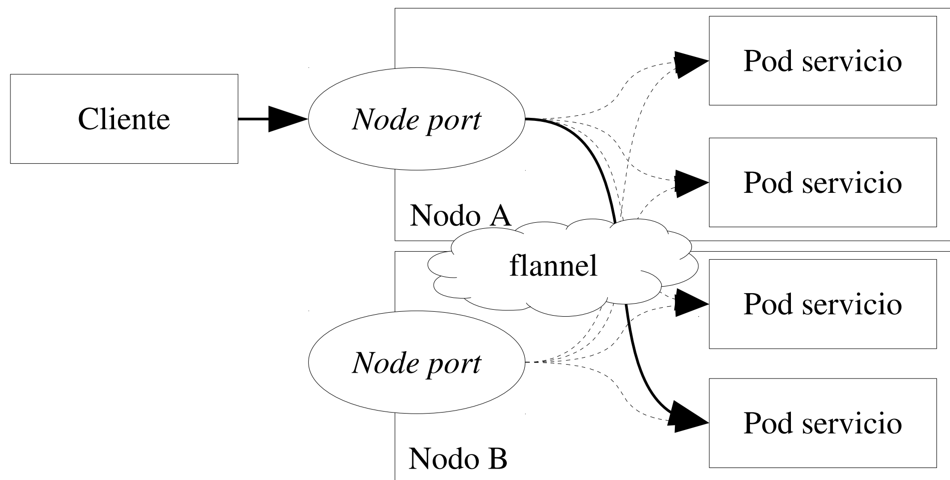

Figura 2: Conexión desde un cliente externo a un pod a través de un node port

2.1.4 Arquitecturas

La naturaleza distribuida de *Kubernetes* permite múltiples arquitecturas para su despliegue. Como hemos visto en los puntos anteriores su funcionalidad viene dada por una colección de agentes, pero en casi ningún caso existen requisitos en cuanto a donde o como debe desplegarse cada uno de ellos.

Podemos agrupar las arquitecturas posibles en tres tipos:

- Monolítica: en la que desplegaríamos una instancia de cada agente en una única máquina. Esta arquitectura no da ninguna garantía de disponibilidad y por tanto no es conveniente para sistemas en producción, pero sin embargo sí que es común en entornos de desarrollo, como es el caso del proyecto *minikube*[10], o del entorno de desarrollo del mismo proyecto *Kubernetes*.
- Con nodos especializados: en la que tendríamos nodos con diferentes roles, como podría ser uno para las bases de datos, otro para los servicios de control (*Controller Manager*, *Scheduler* y *API server*) y otro para los nodos de ejecución. Este tipo de arquitecturas es el más común en sistemas en producción.
- Distribuida: en la que todos los nodos serían iguales a nivel de configuración y en la que ejecutaríamos todos los servicios en el propio clúster. Esta arquitectura es la más acorde al diseño de *Kubernetes*, y con la que se tendría un mayor aprovechamiento de recursos, pero es también la más compleja. Tiene una problemática de huevo-gallina, ya que ¿quién se encarga de planificar la ejecución del primer *Scheduler* en un nuevo despliegue? Necesita partir de una arquitectura monolítica para tener las funcionalidades mínimas en el despliegue inicial. Pese a sus desafíos técnicos, la tendencia es hacia este tipo de arquitecturas difuminando cada vez más los roles de cada nodo.

10 *minikube* es una herramienta que permite la ejecución de un “clúster” de *Kubernetes* local para su uso como entorno de desarrollo. Lo consigue ejecutando los agentes en una máquina virtual.

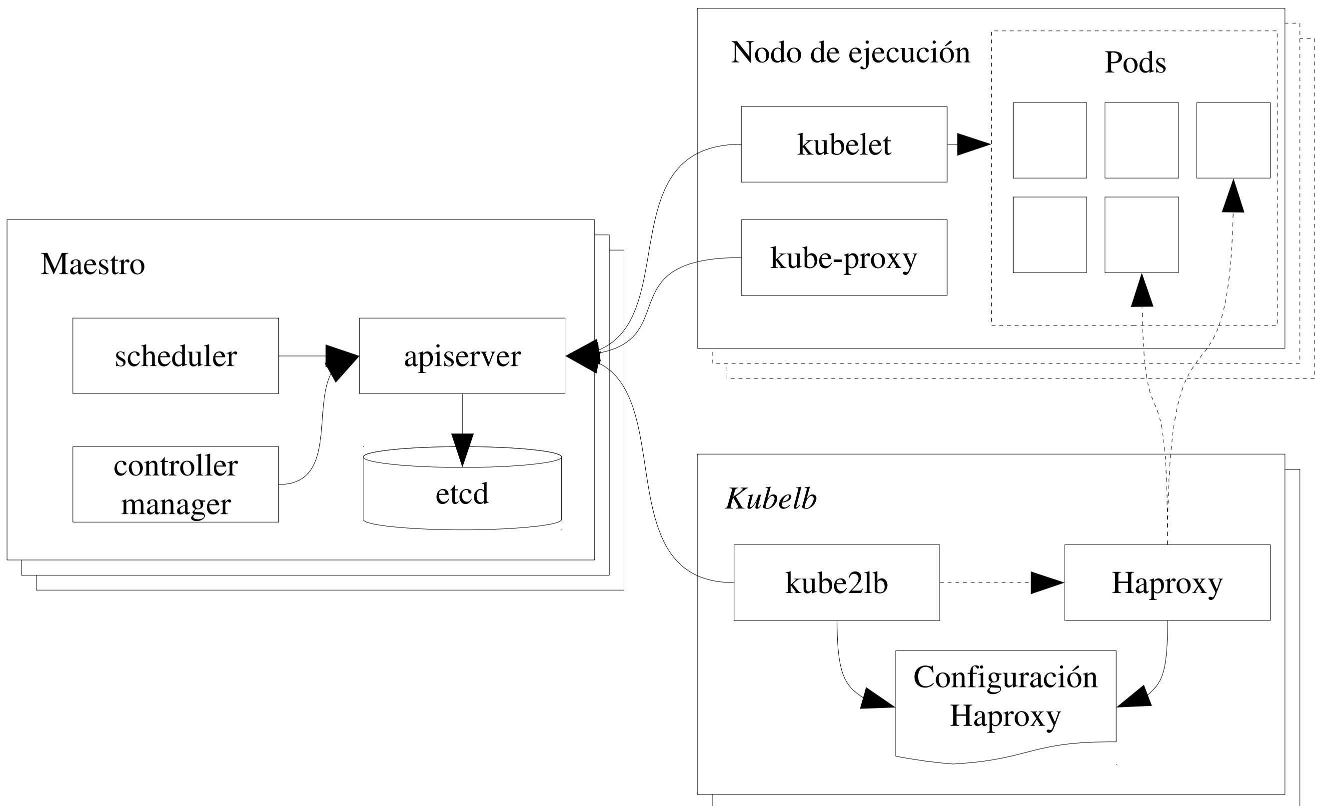

Figura 3: Arquitectura general del clúster de Kubernetes

En Tuenti tenemos nodos especializados con estos tres roles:

- Maestro: en el que desplegamos *etcd*, *Controller manager*, *Scheduler* y *API server*.
- Nodo de ejecución, con *kubelet* y *kube-proxy*.
- "*kubelb*", para las conexiones externas al clúster, con un balanceador de carga *Haproxy*[11] configurado con *kube2lb*.

Para garantizar la disponibilidad de los servicios, tenemos tres nodos maestros y dos nodos *kubelb*.

La arquitectura del despliegue sobre el que trabajaremos quedaría como en la Figura 3[12].

11 Haproxy es un proxy con caracterísiticas de balanceo de tráfico TCP y HTTP de código abierto.
12 En la figura no se representa el detalle de los elementos de red involucrados, sólo la relación entre los componentes del clúster de *Kubernetes*.

2.2 *flannel*

Flannel es un sistema de enrutado de capa 3 (IP) que permite la interconexión de aplicaciones que utilizan interfaces de red virtuales dentro de nodos diferentes como si estuvieran en el mismo nodo [8]. Uno de los objetivos de este sistema es implementar el requisito de *Kubernetes* que vimos en el capítulo 2.1.3 de que todos los pods deben tener una IP única en el clúster y deben ser capaces de conectar con otros pods utilizando estas IPs.

Flannel funciona como un sistema distribuido coordinado a través de un elemento central, en general se utiliza para esta coordinación *etcd*, pero en recientes versiones también pueden utilizarse los *apiservers* de *Kubernetes*. Se despliega con un agente en cada uno de los nodos que forman parte del clúster, este agente tiene una doble responsabilidad, por un lado anunciar al resto de agentes de la existencia del nodo y de la IP que pueden utilizar para conectar con él, y por otro lado la configuración del enrutamiento en el nodo para que las aplicaciones que se ejecuten en él puedan conectar con las aplicaciones en otros nodos.

Dispone de varias implementaciones, o *backends*, para la configuración del enrutado, pero siempre con la idea de que sea completamente transparente para las aplicaciones. Estas implementaciones hacen uso de mecanismos de red que que están generalmente disponibles en sistemas Linux, como veremos más adelante, pero también pueden encargarse de configurar soluciones de nubes privadas; actualmente dispone de implementaciones para las redes privadas virtuales de *Amazon Web Services*, *Google Cloud* y *AliCloud*.

Entrando un poco más en detalles técnicos, analizaremos el funcionamiento de un agente de *flannel*. Cuando comienza su ejecución lee la configuración de *etcd*, esta configuración consiste en un documento JSON con información sobre la red y el *backend* a utilizar. Por ejemplo con la configuración de la Tabla 1 estaríamos indicando a los agentes que utilicen el *backend* VXLAN, que disponen de la red `10.1.0.0/16` y que pueden utilizar subredes de esta red con máscaras de 24 bits, es decir, con esta configuración dispondríamos de 256 subredes con 256 IPs cada una.

```
{
    "Network": "10.1.0.0/16",
    "SubnetLen": 24,
    "Backend": {
        "Type": "vxlan"
    }
}
```

Tabla 1: Ejemplo de configuración de flannel

Con esta información el agente de cada nodo puede reservar una subred disponible y registrar al nodo como propietario de esa subred junto con la IP que otros nodos pueden utilizar para alcanzarle. A continuación comienza a observar las reservas de otros nodos para, utilizando el *backend* elegido, configurar la red.

Finalmente, genera un fichero `subnet.env` como el de la Tabla 2 con variables de entorno con información sobre la subred reservada para que las aplicaciones desplegadas en el nodo puedan saber el rango de IPs que pueden utilizar.

```
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.24.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=true
```

Tabla 2: Ejemplo de fichero de variables de entorno generado por flannel

Utilizando un cliente de *etcd* de consola como `etcdctl`, podemos ver la información con la que trabaja *flannel*. En la Tabla 3 se muestran a modo de ejemplo las entradas generadas en un clúster con 3 nodos, la configuración, y la reserva para la subred de uno de los nodos.

```
core@core-01 ~ $ etcdctl ls --recursive /coreos.com/network/
/coreos.com/network/config
/coreos.com/network/subnets
/coreos.com/network/subnets/10.1.24.0-24
/coreos.com/network/subnets/10.1.100.0-24
/coreos.com/network/subnets/10.1.71.0-24

core@core-01 ~ $ etcdctl get /coreos.com/network/config | jq .
{
  "Network": "10.1.0.0/16",
  "Backend": {
    "Type": "vxlan"
  }
}

core@core-01 ~ $ etcdctl get /coreos.com/network/subnets/10.1.100.0-24 | jq .
{
  "PublicIP": "172.17.8.103",
  "BackendType": "vxlan",
  "BackendData": {
    "VtepMAC": "ba:54:16:31:fe:6c"
  }
}
```

Tabla 3: Datos en etcd con los que trabaja flannel

A continuación se describen los *backends* con los que trabajaremos en este proyecto: VXLAN y *Host Gateway*.

2.2.1 VXLAN

VXLAN[13] (*Virtual eXtensible Local Area Network*) es un protocolo de red ideado para aumentar la flexibilidad de la configuración de redes en grandes despliegues. Esta tecnología es comúnmente utilizada junto con tecnologías de virtualización y busca proveer a estos entornos de las mismas funcionalidades ofrecidas por VLAN[14], pero sin sus limitaciones.

13 Tecnología descrita en RFC 7348 [9]

14 VLAN es una tecnología de virtualización de redes que permite crear múltiples redes lógicas independientes dentro de una misma red física.

Cabecera MAC	Cabecera IP	Cabecera UDP	Cabecera VXLAN	Trama ethernet original

Figura 4: Encapsulación VXLAN

Funciona mediante la encapsulación de tramas Ethernet en paquetes UDP como podemos ver en la Figura 4.

Esta encapsulación tiene la ventaja de que permite interconectar de forma transparente *pods* que se ejecutan en nodos distintos incluso si estos se encuentran en distintas redes físicas, pero lleva consigo la penalización de tener que procesar una mayor cantidad de cabeceras para los mismos datos transmitidos. Esto puede mitigarse en algunos despliegues con *Jumbo-frames*[15], pero el uso de esta técnica obliga a cambios en todas las redes intermedias involucradas, lo cual no siempre es posible.

El enrutamiento en VXLAN funciona mediante la creación de una interfaz de red virtual que se encarga del encapsulamiento. A esta interfaz se le pueden añadir “vecinos” que harían las veces de máquinas conectadas a un mismo segmento de red ethernet, pero que pueden estar en otras redes físicas. Finalmente, para enrutar tráfico a IPs concretas, se han de añadir reglas de capa 3 que indiquen cual de los vecinos anteriormente añadidos tiene esta IP. El propio protocolo tiene un mecanismo con un fin similar al del protocolo ARP[16] que permite a la red generar eventos “*miss*”cuando no tienen información suficiente para enrutar tráfico a una IP concreta. Como respuesta a estos eventos se pueden añadir reglas dinámicamente.

El soporte para VXLAN fue incluido en el kernel de Linux en su versión 3.7.0 y puede gestionarse con herramientas habituales como `iproute2`, disponible en cualquier distribución moderna.

2.2.2 *Host gateway*

Flannel en modo *host gateway* utiliza la tabla de enrutamiento IP de cada nodo como un enrutador de capa 3 configurándole rutas estáticas para las subredes asignadas a cada uno de los otros nodos. Se le llama *Host gateway* porque el *host* actúa como *gateway* de red para las aplicaciones que se ejecutan sobre él.

Este modo tan solo necesita de la pila TCP/IP que puede encontrarse en cualquier sistema operativo, pero por otro lado tiene la limitación de que todos los nodos deben disponer de conectividad en capa 2, es decir, deben estar en la misma red física. Para clusters desplegados completamente en centros de datos o nubes privadas esto es lo más habitual, pero no lo es tanto en despliegues híbridos o en nubes públicas.

---

15 Se conoce como *Jumbo-frames* a tramas ethernet con una carga de datos superior al máximo de 1500 bytes establecido en el estándar. Este tamaño puede conseguirse mediante la configuración del MTU (*Maximum Transmission Unit* – Unidad máxima de transmisión) de las conexiones de red y permite reducir el coste de procesado de cabeceras de red, al tener menos cabeceras para una misma cantidad de datos.

16 ARP – *Address Resolution Protocol*, es un protocolo utilizado en redes TCP/IP para traducir direcciones de capa 3 a direcciones de capa 2, es decir direcciones de red como una dirección IP a direcciones físicas, como la MAC de una interfaz de red ethernet.

En el caso de *flannel*, para cada nodo que se une al clúster añade una nueva regla de enrutamiento mediante la cual redirige todo el tráfico dirigido a la subred reservada para este nodo a la IP del nodo a través de la interfaz de red física. Al igual que comentábamos con VXLAN, en Linux podemos ver y gestionar la tabla de enrutamiento también con `iproute2`. Por ejemplo podemos ver en la Tabla 4 la tabla de enrutamiento de un nodo en un clúster de tres nodos, en ella podemos observar como se han añadido dos reglas a subredes de la red de *flannel* `10.1.0.0/16`. Se puede ver también en dicha tabla una tercera regla a una de estas subredes, esta sería la que utilizaría el nodo para conectar con sus propios contenedores Linux.

```
default via 10.0.2.2 dev eth0  proto dhcp  src 10.0.2.15  metric 1024
10.0.2.0/24 dev eth0  proto kernel  scope link  src 10.0.2.15
10.0.2.2 dev eth0  proto dhcp  scope link  src 10.0.2.15  metric 1024
10.1.1.0/24 via 172.17.8.103 dev eth1
10.1.25.0/24 via 172.17.8.102 dev eth1
10.1.33.0/24 dev docker0  proto kernel  scope link  src 10.1.33.1
172.17.8.0/24 dev eth1  proto kernel  scope link  src 172.17.8.101
```

Tabla 4: Tabla de enrutamiento de un nodo con flannel en modo Host gateway

3 Desarrollo del proyecto

3.1 Análisis del despliegue inicial

Para entender el despliegue que sirve como punto de partida del proyecto, se debe exponer la evolución que este ha tenido desde su forma inicial hasta su situación actual.

El primer despliegue de *Kubernetes* se realizó en servidores físicos en el centro de datos de Tuenti a imagen de los despliegues documentados para entornos de nube pública como pueden ser *Amazon Web Services* o *Google Cloud*. En estos entornos los servicios desplegados en el clúster se exponen utilizando los *node ports* de cada uno de los nodos de ejecución y balanceadores de carga HTTP o TCP ofrecidos por estas plataformas y que se configuran automáticamente. Para exponer un servicio a Internet se siguen entonces estos pasos:

1. Se define el servicio en *Kubernetes* de tipo *LoadBalancer*, esto le indica al software que realiza la integración con la plataforma de nube pública que ha de exponer el servicio utilizando un balanceador de carga y de forma privada también en un puerto aleatorio en cada máquina (lo que se conoce como *node port*).
2. Al definir el servicio, los *kube-proxies* de todos los nodos comienzan a exponerlo en el *node port*. Es especialmente importante entender que los servicios se exponen desde todos los nodos, independientemente de si están ejecutando algún *pod* de ese servicio o no.
3. Se crea automáticamente un balanceador de carga HTTP o TCP que redirige tráfico a todos los nodos del clúster, al *node port* del servicio.

De este modo, como se puede ver en la Figura 5, una conexión externa al clúster pasará primero por el balanceador de carga, este elegirá un nodo con el que conectar y el nodo finalmente enviará el tráfico a un *pod* que puede o no estar ejecutándose en el mismo nodo.

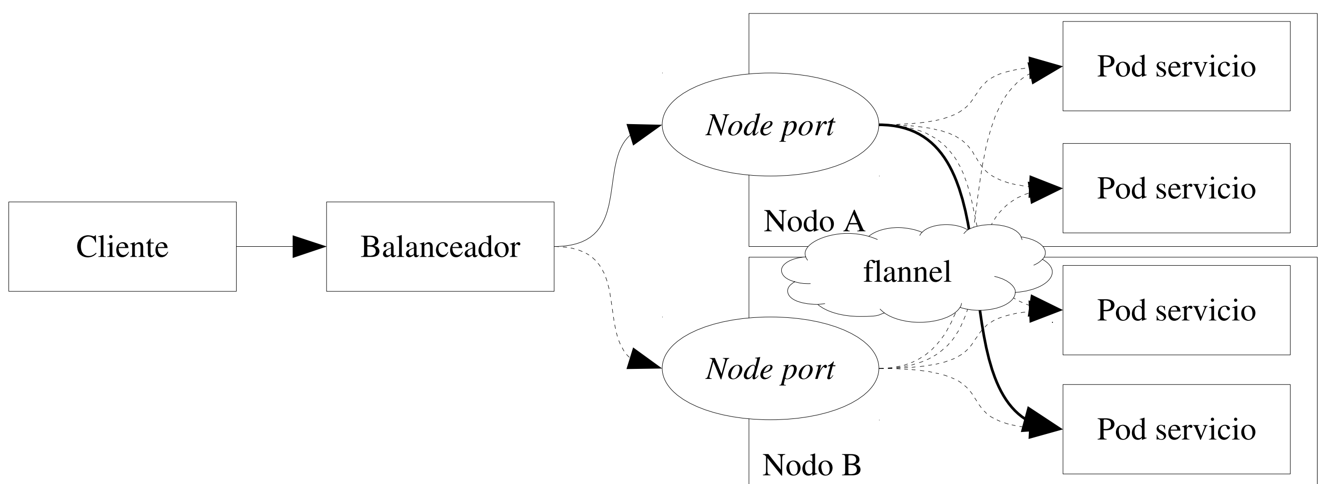

Figura 5: Conexión desde un cliente externo a través de un balanceador de tráfico

Ésta es la solución propuesta por *Kubernetes* y permite una gran escalabilidad. Por un lado cada servicio dispone de su propio balanceador de carga, que en el caso de servicios en nube pública podemos considerar que escalan *ad-infinitum* (en el sentido de que escalan de forma automática y transparente para el usuario/administrador) y por otro lado la carga en los servicios se puede escalar horizontalmente tan solo añadiendo nodos al clúster. Pero cuando el despliegue se realiza en máquinas físicas no se dispone de estos balanceadores de carga de gran capacidad y que pueden crearse de forma dinámica.

Para suplir la carencia de estos balanceadores de carga, diseñamos la figura del *kubelb*, una máquina con un balanceador de tráfico HTTP/TCP que se configura automáticamente de acuerdo a los servicios desplegados en el clúster. En el caso de los balanceadores de carga de los servicios de nube pública en los que cada servicio dispone de un balanceador con su propia IP, en un *kubelb* sólo vamos a tener una IP, y la selección de servicio se realiza por la cabecera *Host* para servicios HTTP o por puerto para servicios TCP. Como se puede observar este balanceador soporta tráfico de red de todos los servicios, y su correcta configuración ha supuesto uno de los mayores desafíos para la migración de servicios a esta plataforma.

Hay algo que diferencia a nuestros *kubelbs* de los balanceadores de carga de los proveedores de nube pública, y es que en el caso de estos últimos estamos hablando de simples elementos de red con una función muy limitada de redirigir conexiones recibidas hacia otras máquinas, pero en el caso de los *kubelbs* son máquinas con sistemas operativos completos y funcionalidades de mucho más alto nivel. Esto permite configurar en los *kubelbs* la red interna de *Kubernetes* dando acceso directo a los *pods* sin necesidad de pasar por los *node ports*, lo cual tiene las siguientes ventajas:

- Evitamos una redirección NAT, la de los *node ports* hacia los pods.
- Un nodo de ejecución solo recibe tráfico de un servicio si tiene *pods* de ese servicio. Esto permite que sea más fácil aislar nodos en caso de operaciones o mal funcionamiento simplemente drenándolos[17].

Pero también añade cierta complejidad:

- Los nodos configurados como *kubelbs* pasan a tener mucha más lógica de red.
- Los balanceadores de carga en los *kubelbs* tienen que actualizar su configuración con mucha más frecuencia, antes solo debían hacerlo cuando un servicio o un nodo cambiaba, pero con esta aproximación también cuando los *pods* cambian, lo que ocurre mucho más a menudo.

Es en estos dos últimos puntos en los que han aparecido más problemas y en los que se centran algunas de las mejoras elaboradas durante este proyecto.

Mientras diseñábamos este modo de operación de los *kubelbs* vimos que realmente se podrían desplegar como nodos de *Kubernetes* normales, con conectividad con el resto de *pods*, pero en los

17 En el contexto de *Kubernetes*, drenar un nodo se refiere a marcarlo como no disponible para asignarle nuevos *pods* y retirarle todos los *pods* que tiene asignados.

que sus *pods* tuvieran además privilegios[18] para poder configurar sus propias IPs. La implementación final se basó en esto, los *kubelbs* pasaron a ser nodos normales, con unas etiquetas específicas para que solo ejecutaran el software de balanceo, y este software pasó a ejecutarse como *pods*. Esto tiene además la ventaja de que la gestión y las operaciones sobre el software de balanceo son muy similares a las del resto del aplicaciones desplegadas en el clúster. Esta operación se hizo en dos fases: primero se añadió la configuración necesaria para que los balanceadores tuvieran visibilidad con la red de los *pods* y después se reemplazó esta configuración por la de un nodo de *Kubernetes* y se lanzaron los balanceadores como *pods*.

3.1.1 Identificación de problemas

Con este nuevo despliegue de los *kubelbs* empezamos a tener problemas puntuales, que parecían vinculados a capacidad, pero que no pudimos localizar en un principio. La afectación era pequeña, y no siempre la misma, en general veíamos que de vez en cuando había servicios que perdían algunas conexiones por límites de tiempo. En nuestros sistemas de monitorización no veíamos nada que pudiera indicar que llegáramos a límites de recursos, aunque los problemas aparecían más frecuentemente entre semana a horas centrales del día en España, lo que coincide tanto con las horas de trabajo de nuestros equipos de desarrollo como con la de mayor actividad de nuestra zona geográfica con más clientes, así que todo apuntaba a algún problema de capacidad.

Fuimos identificando varios problemas:

- Aunque habíamos ajustado desde el principio los límites globales de conexiones, estos límites, junto con los de sesiones globales en los balanceadores de tráfico *Haproxy* no eran suficientes para evitar que el bloqueo[19] de una aplicación acabara saturando todas las conexiones afectando a todos los servicios (ver Sec. 3.2.2).
- Durante las recargas de configuración de *Haproxy* se perdían conexiones incluso usando las funcionalidades que ofrece para recargas de configuración en caliente. Ya conocíamos con anterioridad que esto podía pasar, y en principio decidimos ignorarlo asumiendo que eran solo unas pocas conexiones. Pero la tendencia de esta cifra era incremental en dos dimensiones: por un lado cada vez teníamos más actividad de más clientes y por tanto más conexiones que podrían verse afectadas, y por otro teníamos cada vez más actividad en el clúster al tener cada vez más servicios y más desarrolladores trabajando en ellos. Con más conexiones, y más cambios de configuración cada vez iba a ser mayor el número de conexiones perdidas por algo que no era un defecto si no parte de la funcionalidad básica de nuestro clúster. Decidimos dejar de ignorar el problema (ver Sec. 3.2.3)

18 *Kubelet* se puede lanzar con un modo que permite a los *pods* que ejecuta solicitar permisos adicionales utilizando *Linux capabilities*. Por ejemplo para configurar su red, un *pod* necesita la *capability NET_ADMIN*. Con la configuración por defecto, un *pod* no puede solicitar permisos adicionales.

19 Entendemos aquí bloqueo como una situación en la que una aplicación recibe conexiones y las mantiene sin cerrarlas nunca, produciendo pérdida de recursos.

Pero para identificar el problema más serio tuvimos que pasar varias horas en caída completa. Llegó un día en que ese pequeño número de conexiones que fallaban provocó una especie de efecto de bola de nieve que saturó completamente los balanceadores de tráfico. Fue durante esa caída cuando pudimos ver la raíz del problema. Todas nuestras métricas seguían mostrando valores dentro de los límites, tanto de uso de procesador y memoria como de conexiones y ancho de banda, pero entrando en la máquina vimos que un proceso llamado "`ksoftirqd/0`" estaba utilizando el 100% de una de las CPUs.

`ksoftirqd` es un hilo del kernel Linux encargado del manejo de las *softirqs*, un tipo de funciones diferidas que se ejecutan en contexto de interrupción y no pueden contener ningún mecanismo que provoque cambio de contexto. Normalmente activadas desde rutinas de servicio a interrupción hardware de entrada/salida, se utilizan para descargar de tareas no críticas el código de la rutina de servicio, que interesa que devuelva el control al kernel lo antes posible. En particular, los controladores de tarjetas de red hacen un uso muy intensivo de las *softirqs*, lo que proporcionaba una primera hipótesis de posible causa del problema. De este hilo se lanza uno por CPU, indicando el número el identificador de esta. Como indica la página *man*[20] de este proceso [10]: "si *ksoftirqd* está consumiendo más de un pequeño porcentaje del tiempo de CPU, esto indica que la máquina está bajo una gran carga de interrupciones software", por tanto comenzamos a investigar qué estaba originando semejante carga.

Para ello el primer punto es analizar el contenido del fichero `/proc/interrupts`, este fichero contiene una tabla con las interrupciones por CPU y por dispositivo que las genera, como podemos ver en el ejemplo del fichero en una máquina virtual con dos CPUs en la Tabla 5.

20 *man* es el comando que sirve de interfaz con los manuales de referencia en un sistema Linux

```
core@localhost ~ $ cat /proc/interrupts
           CPU0       CPU1
  0:        142          0   IO-APIC   2-edge      timer
  1:         10          0   IO-APIC   1-edge      i8042
  8:          0          0   IO-APIC   8-edge      rtc0
  9:          0          0   IO-APIC   9-fasteoi   acpi
 12:         72          0   IO-APIC  12-edge      i8042
 14:       4392          0   IO-APIC  14-edge      ata_piix
 15:          0          0   IO-APIC  15-edge      ata_piix
 19:       1535          0   IO-APIC  19-fasteoi   virtio0
NMI:          0          0   Non-maskable interrupts
LOC:       6652       5866   Local timer interrupts
SPU:          0          0   Spurious interrupts
PMI:          0          0   Performance monitoring interrupts
IWI:          0          0   IRQ work interrupts
RTR:          0          0   APIC ICR read retries
RES:       4433       5256   Rescheduling interrupts
CAL:        742       2665   Function call interrupts
TLB:         88         95   TLB shootdowns
TRM:          0          0   Thermal event interrupts
THR:          0          0   Threshold APIC interrupts
DFR:          0          0   Deferred Error APIC interrupts
MCE:          0          0   Machine check exceptions
MCP:          1          1   Machine check polls
ERR:          0
MIS:        203
PIN:          0          0   Posted-interrupt notification event
PIW:          0          0   Posted-interrupt wakeup event
```

Tabla 5: Ejemplo de `/proc/interrupts`

En el caso de los *kubelbs* se vió que el mayor número de interrupciones en la CPU 0 era originado por el controlador de las tarjetas de red[21], como podemos ver en el extracto de su fichero `/proc/interrupts` en la Tabla 6, lo que resultaba además coherente con la hipótesis.

```
          CPU0       CPU1       CPU2  ...
. . .
50:          1          0          0       PCI-MSI 1048576-edge      eno1
51: 3508416808          0          0       PCI-MSI 1048577-edge      eno1-TxRx-0
52:  273092045          0          0       PCI-MSI 1048578-edge      eno1-TxRx-1
53: 3525918583          0          0       PCI-MSI 1048579-edge      eno1-TxRx-2
54: 4022960697          0          0       PCI-MSI 1048580-edge      eno1-TxRx-3
55: 2409992185          0          0       PCI-MSI 1048581-edge      eno1-TxRx-4
56: 3156886296          0          0       PCI-MSI 1048582-edge      eno1-TxRx-5
57: 2953355510          0          0       PCI-MSI 1048583-edge      eno1-TxRx-6
58: 2942209742          0          0       PCI-MSI 1048584-edge      eno1-TxRx-7
. . .
61:          1          0          0       PCI-MSI 1054720-edge      eno2
62: 1958202329          0          0       PCI-MSI 1054721-edge      eno2-TxRx-0
63: 3354806884          0          0       PCI-MSI 1054722-edge      eno2-TxRx-1
64: 4205525126          0          0       PCI-MSI 1054723-edge      eno2-TxRx-2
65:  441829734          0          0       PCI-MSI 1054724-edge      eno2-TxRx-3
66: 3013508527          0          0       PCI-MSI 1054725-edge      eno2-TxRx-4
67: 3810554405          0          0       PCI-MSI 1054726-edge      eno2-TxRx-5
68: 3319154190          0          0       PCI-MSI 1054727-edge      eno2-TxRx-6
69: 3306670483          0          0       PCI-MSI 1054728-edge      eno2-TxRx-7
```

Tabla 6: Extracto del fichero `/proc/interrupts` *en un kubelb*

21 Cada *kubelb* tiene dos tarjetas de red por razones de alta disponibilidad

En el extracto de la Tabla 6 podemos ver también algunas características de las controladoras de red que utilizamos, unas *Intel I350.* Estas controladoras disponen de ocho colas de envío y recepción independientes por puerto [11], esta característica se publicita para su uso junto a soluciones de virtualización para poder aislar el tráfico de distintas máquinas virtuales, como si cada una dispusiera de su propia interfaz de red física. Pero puede ser también utilizada para balancear la carga de interrupciones entre distintos procesadores logrando manejar una mucho mayor cantidad de tráfico de red. Como solución temporal utilizamos esta característica.

Para ello se ha de configurar la afinidad por CPU de las *IRQ*[22]. En sistemas operativos Linux se puede interactuar con el subsistema que maneja las *IRQs* a través del sistema de ficheros virtual `procfs`, montado normalmente en `/proc`. Dentro de `/proc` podemos encontrar un directorio `irq`, que a su vez contiene un directorio por cada una de las interrupciones que antes vimos en el fichero `/proc/interrupts`. En el directorio de cada una de las interrupciones podemos encontrar algunos ficheros relacionados con la afinidad en sistemas multiprocesador (*smp*, *symmetric multiprocessing*). De entre estos ficheros, nos centraremos en `smp_affinity` y `smp_affinity_list`, ambos son interfaces diferentes para la misma funcionalidad; indican mediante máscaras de bits el primero y mediante rangos de números el segundo qué CPUs pueden destinarse al manejo de esta interrupción. Las interrupciones de la *Intel I350* solo utilizan la primera de las CPUs disponibles. Como ejemplo, en la Tabla 7 podemos ver el contenido por defecto de estos ficheros para la interrupción 51, asociada con la primera cola de la primera interfaz de red. En estas máquinas disponemos de dos procesadores con ocho núcleos con *hyperthreading*[23] cada uno y como podemos ver, por defecto, estas interrupciones tienen afinidad con los dieciséis núcleos, evitando los otros dieciséis que ve el sistema operativo por el *hyperthreading*.

```
$ ls /proc/irq/51/
affinity_hint  eno1-TxRx-0  node  smp_affinity  smp_affinity_list  spurious
$ cat /proc/irq/51/smp_affinity
00000000,00000000,00ff00ff
$ cat /proc/irq/51/smp_affinity_list
0-7,16-23
```

Tabla 7: Afinidad por procesador de una irq

Para cambiar la afinidad, se ha de escribir en ese fichero para cada interrupción. En nuestro caso asignamos a cada uno de los ocho núcleos principales del segundo procesador una cola de cada interfaz de red. Evitamos los ocho primeros núcleos, el primero lo van a usar algunas interrupciones por defecto y los seis siguientes se los hemos asignado a los seis procesos de *Haproxy* que utilizamos actualmente para el balanceo de tráfico. Evitamos también los hilos de *hyperthreading*

22 *IRQ – interrupt request* – denominación de las peticiones de interrupción, cada una identificada en el sistema operativo por un número único

23 *Hyperthreading* es la implementación de Intel de la tecnología s*imultaneos multithreading*, la cual permite a múltiples hilos ejecutarse simultáneamente en un mismo núcleo de procesador. Esta tecnología aumenta el rendimiento general del procesador realizando computación paralela, pero puede aumentar la latencia de ciertas operaciones.

para evitar aumentos en la latencia. Para escribir las afinidades utilizamos un código en *bash*[24] similar al de la Tabla 8.

```
#!/bin/bash

first_cpu=16
first_irq_eno1=51
first_irq_eno2=62

function set_affinity() {
    local first_irq=$1
    for i in $(seq 0 7); do
        local cpu=$(( $first_cpu + $i ))
        local irq=$(( $first_irq + $i ))

        echo ${cpu} > /proc/irq/${irq}/smp_affinity_list
    done
}

set_affinity $first_irq_eno1
set_affinity $first_irq_eno2
```

Tabla 8: Código de bash para la asignación de afinidades por cpu de las interrupciones

Con esto logramos balancear la carga y paliar los problemas que estábamos sufriendo inicialmente, pero la solución no parece del todo adecuada, ya que no resuelve las causas de la gran carga que estábamos teniendo. Para enfrentarnos a la raíz del problema habríamos de realizar más cambios que veremos en las secciones 3.2.1 Reducción de reglas de iptables y 3.3 Reemplazo del sistema de enrutado.

24 *Bash (Bourne-again shell)* es un intérprete de comandos y lenguaje de programación de código abierto ampliamente utilizado en sistemas Linux.

3.1.2 Evaluación del rendimiento y situación inicial

La evaluación del rendimiento del sistema en este proyecto se va a centrar en localizar posibles cuellos de botella que perjudiquen el aprovechamiento eficiente de los recursos de los nodos de ejecución (en especial procesador y memoria) y dificulten por tanto el escalado horizontal del clúster. Especialmente vamos a buscar los cuellos de botella causados por la red para el manejo de peticiones HTTP, que suponen la práctica totalidad del tráfico en el clúster. Para ello haremos observaciones sobre un escenario que se aproxima lo máximo posible a la infraestructura en producción:

- Todos los nodos utilizados son máquinas que normalmente forman parte del clúster principal. Para estas pruebas se eliminan las instancias de servicios que estén ejecutando para que no se vean afectados, ni afecten, al tráfico de producción.
- Uno de los nodos es configurado como los *kubelbs* de producción, y sobre este se prueban los distintos cambios de configuración de estas máquinas antes de aplicarlos.
- Dos nodos de ejecución se dedican a ejecutar cada uno una instancia de un servidor web *nginx*[25] que tan solo va a servir una página estática por HTTP. Para *nginx* se utiliza su configuración por defecto excepto por el número de procesos, establecido a 8, y por la configuración de los *logs* de acceso, que se desactiva para evitar escrituras a disco durante las pruebas. Se ejecutan pruebas de carga contra una y contra dos instancias de modo que pueda observarse la capacidad de escalado del clúster. Este servicio se despliega utilizando *Kubernetes*, de modo que las conexiones hacia él pasen a través de los elementos de red por los que pasaría el tráfico hacia un servicio real.
- Las pruebas de carga se lanzan desde una máquina física con 32 hilos de CPU (2 procesadores con 8 núcleos con *hyperthreading*) y 32GB de RAM. Se realizan con *ab* (*Apache Benchmark*), un software de código abierto ampliamente utilizado para este fin, y que si bien no es muy sofisticado, nos sirve para apreciar el impacto de los cambios propuestos en este proyecto.

En este escenario no se busca evaluar el rendimiento de servicios reales, si no buscar los límites de la propia infraestructura de red, por ello se realizan pruebas contra un servicio desplegado únicamente con este fin y no contra un servicio real.

Los resultados de la ejecución de las pruebas de carga pueden verse en el Anexo I: Ejecuciones de pruebas de carga.

Como primera prueba de carga, y para que sirva de referencia, ejecutamos *ab* contra un servidor *nginx* desplegado directamente en una de las máquinas sin utilizar *Kubernetes*, de modo que la conexión entre cliente y servidor es directa, sin pasar por ninguno de los elementos de red del

---

25 *nginx* es un servidor HTTP ampliamente utilizado y de alto rendimiento, especialmente sirviendo contenido estático. Es el servidor utilizado en prácticamente la totalidad de los servicios de producción en Tuenti.

clúster. En esta prueba observamos (Tabla 19) que *nginx* con esta configuración es capaz de atender más de 20 mil peticiones por segundo, atendiendo al 99% de estas peticiones en 3 ms como máximo.

Si miramos a las cifras de las pruebas de carga contra el mismo servicio desplegado en *Kubernetes* con la configuración inicial (Tabla 20), vemos que la tasa de conexiones se reduce casi a una cuarta parte, y el tiempo de respuesta en el percentil 99 se duplica. Esta comparación nos hace ver que existe bastante margen de mejora.

Al aumentar a 2 el número de réplicas desplegadas en *Kubernetes*, vemos que pese al balanceo de peticiones entre ambas instancias, no se aprecia ninguna diferencia significativa ni en la tasa de peticiones ni en los tiempos de respuesta (Tabla 21 y Figura 6).

Con estas pruebas tenemos datos para confirmar que con esta configuración no es posible escalar horizontalmente añadiendo instancias adicionales de un servicio.

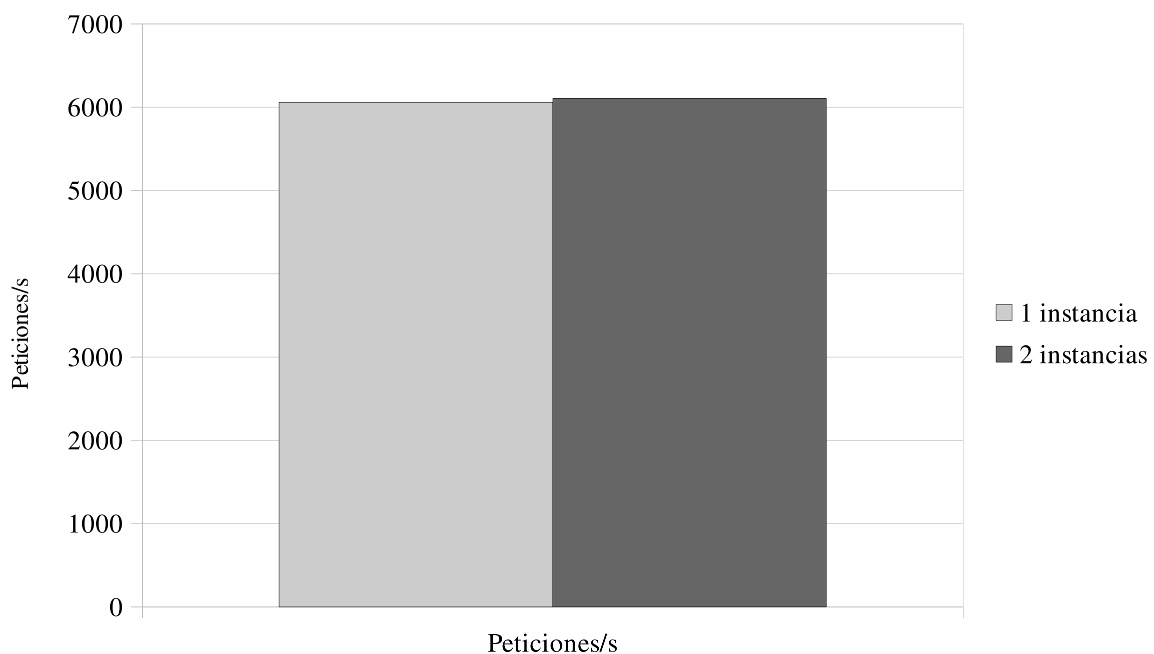

Figura 6: Peticiones/s con la configuración inicial

3.2 Mejoras en los *kubelbs*

Como hemos visto en el capítulo 3.1, nuestro despliegue de *Kubernetes* tenía algunos problemas en su diseño que mermaban el rendimiento y fiabilidad de la red, nos hacían infrautilizar parte de nuestros recursos y no nos permitían escalar tanto como sería deseable. Uno de los puntos clave del despliegue son los *kubelbs*, ya que su función les hace soportar gran parte del tráfico de red del clúster. En este capítulo se describen los cambios introducidos en estas máquinas para mejorar la situación.

3.2.1 Reducción de reglas de *iptables*

La causa del incremento de carga por *software interrupts* debía estar originada por los cambios realizados en los *kubelbs* para introducirlos como nodos de ejecución en el clúster de *Kubernetes*. Analizando estos cambios llegamos a la conclusión de que *kube-proxy* y *flannel* podrían estar introduciendo una sobrecarga por funcionalidades que realmente no necesitábamos en estos nodos.

Kube-proxy introduce una gran cantidad de reglas de *iptables* para configurar las IPs de servicio y *node ports* y en estos nodos, al no estar ejecutando *pods* de servicio no lo necesitábamos realmente. Los únicos servicios en ejecución son *kube2lb* y *haproxy* y están configurados para utilizar y gestionar directamente las interfaces físicas del nodo.

Flannel configura el nodo para que pueda ser utilizado como puerta de enlace por los *pods* que se ejecutan en él, esto conlleva activar el enmascarado IP[26], lo que a su vez conlleva activar *conntrack*. *conntrack* mantiene un registro de todas las conexiones de red gestionadas por el nodo para recordar a qué IPs internas enmascaradas pertenecen. Esto supone una sobrecarga en las conexiones, y es algo que realmente tampoco necesitábamos en estos nodos al no estar actuando como puerta de enlace de ningún *pod* o nodo.

A continuación se detalla como se eliminaron los puntos en los que se activaban funcionalidades relacionadas con *iptables* que podían estar incrementando la carga y que no necesitábamos:

- *Kube-proxy* añadía las reglas para las IPs de servicio y *node ports*, lo desinstalamos completamente.
- *Flannel* por defecto activa en enmascaramiento IP, lo desactivamos.
- *Docker*, el sistema que utilizamos para gestionar contenedores Linux, también activa el enmascaramiento IP y también utiliza *iptables* para diferentes configuraciones de red que no utilizamos. Desactivamos tanto el enmascaramiento IP como el uso de *iptables*.
- *Flannel* añade configuraciones para que *Docker* pueda utilizar sus subredes, esta configuración es incompatible con las opciones desactivadas en el punto anterior. Como solución nos aseguramos de eliminar estas configuraciones antes de arrancar *Docker*.

26 El enmascarado IP es una función de red en servidores Linux que permite a múltiples nodos situados en una red interna comunicarse con nodos situados en otras redes utilizando la IP del *gateway* de su red.

- *Kubelet*, al arrancar, comprueba con *iptables* que ciertos parámetros de red están correctamente configurados, entre ellos el enmascaramiento IP. *iptables* carga automáticamente los módulos del kernel que necesita para sus funciones; en este caso las comprobaciones necesitaban listar la tabla NAT, lo cual a su vez carga el módulo `iptables_nat`, que depende de `nf_conntrack`. Es decir, el mero listado de reglas NAT puede activar el registro de conexiones de *conntrack*. Por suerte *Kubelet* dispone de una opción para evitar toda interacción con *iptables*, que la activamos para evitar este tipo de efectos colaterales.

Tras estos cambios, comprobamos que tras un reinicio de la máquina no se utiliza *iptables*, *conntrack* está desactivado y no se cargan los módulos del kernel innecesarios.

3.2.2 Nuevos límites de conexiones y sesiones activas

Para mitigar los problemas que puedan venir derivados de un *pod* o un servicio bloqueado extendimos el uso de las diferentes opciones de las que *Haproxy* dispone para configurar los límites de conexiones activas simultáneas. Inicialmente sólo configuramos el límite global, pero como hemos visto en Sec. 3.1.1 esto hacía que un único *pod* bloqueado pudiera llegar a saturar todo el balanceador.

En *Haproxy* podemos distinguir tres elementos principales, cada uno de los cuales tiene su propia configuración de límites de conexiones [12]:

- Los *frontends* son los puntos desde los cuales *Haproxy* sirve conexiones a clientes, por cada uno de ellos abre un puerto en una IP concreta de la máquina.
- Los *backends* abstraen la implementación de un servicio concreto, en ellos se configuran listas de servidores y comprobaciones de salud sobe ellos.
- Los *servers* son cada una de las direcciones (IP y puerto) que pueden atender las peticiones de un determinado *backend*, cada uno de ellos puede tener una configuración diferente, que irá especialmente orientada a determinar la proporción de conexiones que debe atender.

En general un mismo *backend* puede atender conexiones de diferentes *frontends*, pero en nuestro caso cada *backend* está en un solo *frontend*.

Como ejemplo de configuración de *Haproxy* podemos tener tres servicios, el primero con tres servidores, el segundo solo con uno y el tercero con dos. Podríamos tener un *frontend* para los dos primeros servicios escuchando en todas las interfaces en el puerto 80, y otro para el tercer servicio escuchando en el 8080 (ver Figura 7).

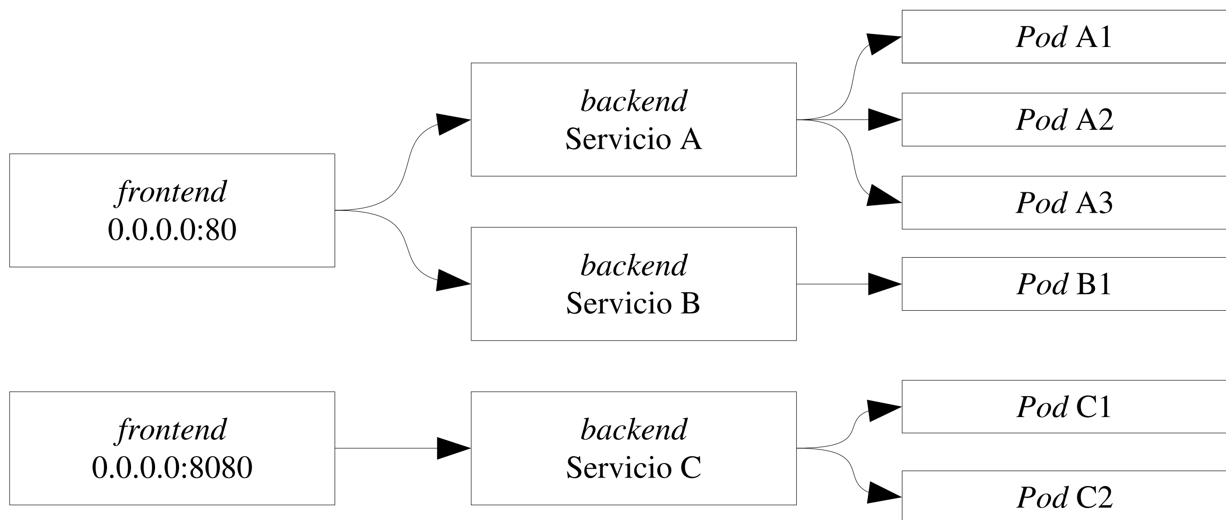

Figura 7: Ejemplo esquematizado de configuración de Haproxy

En este ejemplo, y con la configuración de partida, si por ejemplo el único *pod* del segundo servicio se satura hasta el punto de tener tantas conexiones abiertas como el límite global, todo el resto de servicios tendrían problemas para responder, incluso el tercer servicio situado en otro *frontend*. Esta configuración global puede ser útil para limitar la capacidad total del balanceador, por ejemplo si disponemos de una cantidad limitada de memoria, pero no nos ayuda cuando un solo *server* o un *backend* tienen un comportamiento errático.

Para completar esta configuración definimos entonces dos nuevos límites de conexiones, uno por *frontend* y otro por *server*, de modo que el primero es menor que el global y el segundo es menor que el primero. Podríamos definir también uno por *backend*, pero este viene dado realmente por la suma de los límites de sus servidores. De este modo si por ejemplo se bloquea el *pod* A3 del ejemplo anterior (ver Figura 7) hasta el punto de que llegue a su límite de conexiones, el servicio A puede seguir sirviendo utilizando los *pods* A1 y A2 y el resto de servicios no se ven afectados.

Con esta nueva configuración comprobamos que el bloqueo en un *pod* o un conjunto de *pods* no afecta a las conexiones del resto de *frontends* o *backends*.

Adicionalmente, también permitimos configurar límites de tiempo de conexión diferentes para cada servicio [13], esto nos permite tener unos límites relativamente bajos por defecto, para que conexiones bloqueadas se descarten rápidamente, pero a la vez poder atender peticiones de servicios que por su naturaleza necesitan de más tiempo.

3.2.3 Mejoras en las recargas de configuración

Como se describía en 3.1.1, al modificar *kube2lb* para que configurara el balanceador de tráfico utilizando los *pods* directamente en lugar de los *node ports,* la cantidad de recargas de configuración necesarias se incrementó considerablemente llevando a un incremento de fallos de conexión. Cuando se empezó a abordar este problema, se consideró revertir los cambios en *kube2lb*, pero el hecho de utilizar *pods* en vez de *node ports* nos traía ventajas que queríamos mantener, en especial la de enviar tráfico solo a nodos que tuvieran *pods* en ejecución.

Para mitigar este problema se definieron dos líneas de actuación:

- Investigar si realmente todas las recargas de configuración que estábamos haciendo eran necesarias, e intentar evitar las que no lo fueran. Es decir, reducir en lo posible el número de recargas de configuración.
- En las recargas que sí sean necesarias, intentar eliminar la pérdida de conexiones.

En cuanto a la cantidad de recargas de configuración, se realizaron observaciones para entender cuales eran los mayores causantes de "estrés" de recargas de configuración. Para estas observaciones simplemente se analizaron los eventos que causaban estas recargas utilizando *kubectl*, el cliente de línea de comandos de *Kubernetes*. Este cliente puede suscribirse a los eventos del *apiserver* igual que hacemos con *kube2lb*, pero en el caso de *kubectl* este imprime los eventos por consola. Con estas observaciones identificamos algunos puntos de mejora:

- Versiones recientes del *apiserver* ofrecen mecanismos para la elección de líder en agentes que lo necesiten, como pueden ser el *Scheduler* o el *Controller Manager*. Estos mecanismos generan eventos constantemente que estaban disparando recargas de configuración en *kube2lb*. Estos eventos se caracterizan porque son similares a los que se pueden observar cuando la lista de *pods* que implementan un servicio cambia, pero en este caso las listas de *pods* están siempre vacías.
- Cambios en servicios de tipo *ClusterIP* estaban disparando cambios de configuración, a pesar de que para estos servicios no se genera configuración, ya que no están pensados para ser expuestos fuera del clúster.
- Cada vez que *kube2lb* reconectaba con el *apiserver*, este volvía a solicitar toda la información y a regenerar toda la configuración, a pesar de que en la mayoría de ocasiones nada había cambiado durante el reducido tiempo de desconexión.
- Por la operativa normal del clúster, cuando se realiza la actualización de una aplicación, todos sus *pods* se van parando uno a uno y se van reemplazando por *pods* con la nueva versión. Cada vez que se elimina un *pod* viejo, o se crea uno nuevo se produce una recarga de configuración. Estas recargas son legítimas, así que no podemos hacer nada por reducirlas, pero debemos tener en cuenta que estas recargas han de realizarse lo más rápidamente posible.

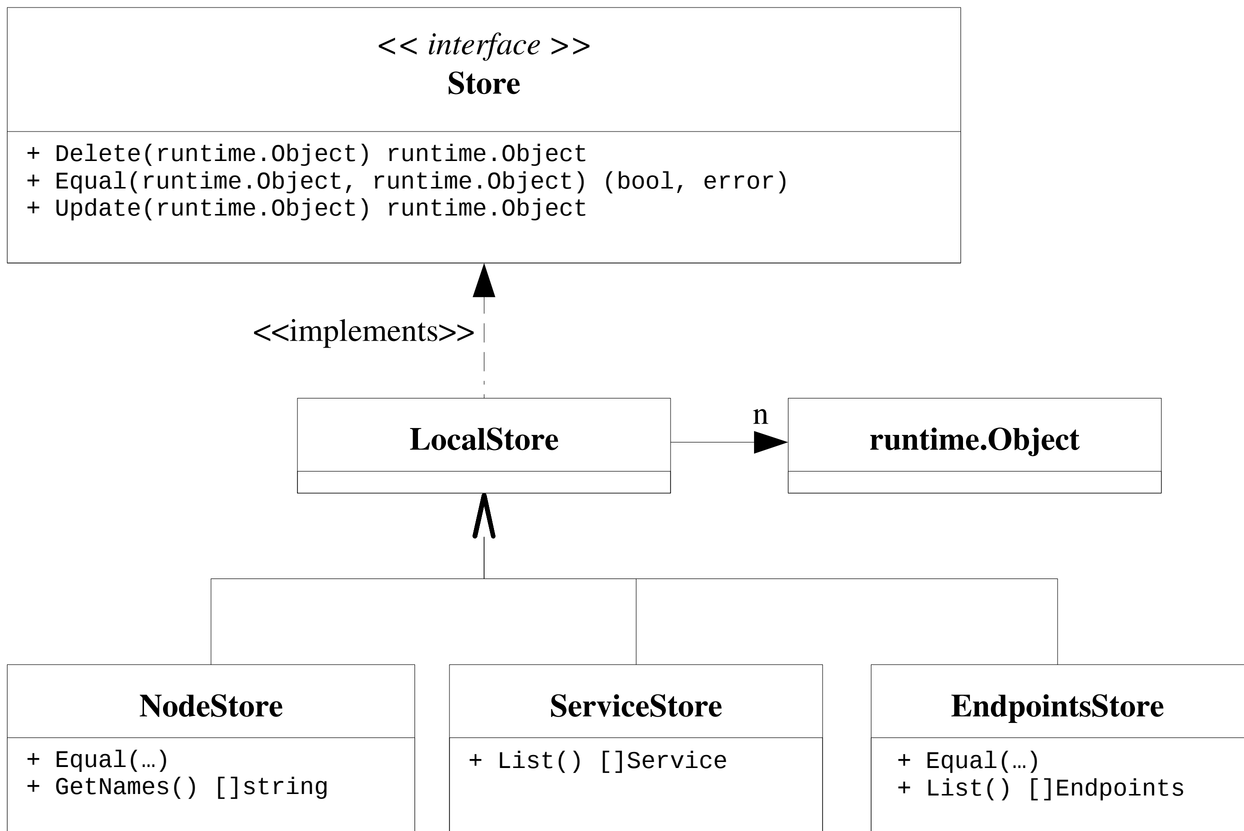

Figura 8: Diagrama de clases de la caché local

La solución implementada se basa en dotar a *kube2lb* de una caché local en la que mantiene la información que necesita para generar la configuración. En versiones anteriores se decidió no guardar ningún estado local por simplicidad, cada vez que se recibía un evento se solicitaba toda la información necesaria para generar la configuración al *apiserver*.

Esta caché local se basa en una tabla *hash* por cada tipo de datos que utilizamos para generar la configuración (nodos, servicios y *endpoints* o listas de *pods*), cada una de estas estructuras implementa una interfaz con métodos para actualizar su contenido (`Update` y `Delete`) y un método `Equal` que permite comparar los objetos recibidos en los eventos con los objetos que tenemos almacenados en las cachés. Si el cambio ha de provocar una recarga de configuración la provoca y si no simplemente se mantiene actualizada la caché. Además de la interfaz común, cada estructura de datos implementa también métodos específicos para las colecciones de objetos que almacenan que facilitan posteriormente la generación de configuración a partir de ellos.

El diseño de estas estructuras de datos queda como en la Figura 8.

El algoritmo para un evento podría resumirse con el pseudocódigo de la Tabla 9.

```
almacen := nuevo AlmacenLocal()

para cada evento := recibeEvento():
    seleccionar evento.Tipo:
    caso AÑADIDO:
        almacen.Actualiza(evento.Objeto)
    caso MODIFICADO:
        viejo := almacen.Actualiza(evento.Objeto)
        si almacen.Igual(viejo, evento.Objeto) entonces
           continua
        fin
    caso BORRADO:
        almacen.Borra(evento.Objeto)
    fin
    actualizaConfiguracion()
fin
```

Tabla 9: Algoritmo de manejo de eventos

La mayor diferencia con la implementación anterior es que ésta permite ver si un evento está realmente cambiando el estado, evitando las recargas causadas por las elecciones de líderes y en parte las causadas por reconexiones.

Para impedir totalmente estas recargas causadas por reconexiones, aprovechamos que *Kubernetes* versiona todos los objetos y eventos con un número monotónicamente creciente. Esto permite que cuando un agente se suscribe a los eventos del clúster, pueda indicar la versión a partir de la cual está interesado. En la nueva versión de *kube2lb* se guarda la versión del último evento recibido y cuando se reconecta solicita eventos a partir de este punto, de modo que se puede tener la certeza de que ni se pierde información ni se reciben eventos duplicados.

Haciendo uso de esta caché local no solo se consiguió eliminar la práctica totalidad de recargas de configuración innecesarias si no que también se disimuyó tremendamente el volumen de las interacciones entre *kube2lb* y el *apiserver*, reducido en la actualidad a los eventos recibidos, sin peticiones adicionales para obtener la información necesaria para generar la configuración.

Para evitar el problema de las conexiones perdidas en cada recarga legítima se tuvo que investigar la situación y explorar varias alternativas.

Para aproximarnos al problema de estas conexiones perdidas, tenemos que entender como recarga configuraciones *haproxy*. Se puede describir de manera simplificada en estos pasos:

1. Se lanza un nuevo proceso *haproxy* con la nueva configuración. Este inicia a su vez los procesos hijos que manejarán las conexiones.
2. Los nuevos procesos de *haproxy* notifican a los procesos antiguos que han de finalizar.
3. Los procesos antiguos dejan de aceptar nuevas conexiones, pero siguen sirviendo las conexiones existentes.
4. Los procesos nuevos comienzan a aceptar conexiones.
5. Los procesos antiguos van finalizando conforme sus conexiones activas finalizan.

Este método de recarga de configuración permite que las conexiones existentes no queden afectadas, pero como se puede observar, entre los pasos 3 y 4 hay un tiempo en el que ninguno de los servidores, ni los nuevos ni los viejos, están aceptando conexiones. En realidad, *haproxy* tiene una solución parcial para este hueco de tiempo, utiliza una opción implementada por los *sockets* en sistemas BSD y Linux llamada `SO_REUSEPORT`, que permite a un nuevo socket escuchar en un puerto en uso. Esta solución es parcial, porque se ha observado que en escenarios con mucho tráfico y muchos cambios de configuración, como es nuestro caso, durante las recargas, algunos clientes reciben paquetes `RST` al conectar, lo que hace fallar la conexión.

En un detallado artículo [14], Willy Tarreau, desarrollador principal de *Haproxy* y contribuidor del kernel Linux, analiza este problema e indica que la causa es una condición de carrera entre los procesos de *haproxy* y los mecanismos de sincronización entre procesadores en sistemas multiprocesador. En este mismo artículo describe la solución definitiva que están implementando para futuras versiones, y analiza algunas otras soluciones alternativas que pueden aplicarse mientras tanto.

La solución implementada en nuestro caso se basa en algunas de estas ideas, particularmente en la de "retener" los paquetes `SYN` de las conexiones nuevas mientras se recarga la configuración. Valoramos también otras opciones:

- Descartar todos los paquetes `SYN` durante la recarga, forzando un reintento `TCP`. Esta opción, la más sencilla de implementar, se aprovecha del funcionamiento del protocolo `TCP`, el cual retransmite un paquete del que no recibe confirmación. El problema de esta solución es que el estándar define el tiempo mínimo de espera antes de realizar una retransmisión en un segundo, un tiempo demasiado alto.
- Escuchar en interfaces de red virtuales y redirigir el tráfico desde la interfaz real utilizando *iptables*. Esta opción, que debería funcionar, la consideramos demasiado compleja para una operación tan delicada y la descartamos casi desde el principio. Al poder haber varias "generaciones" de procesos *haproxy* corriendo cuando suceden varias recargas consecutivas, requeriría de un sistema de orquestación que mantuviera una cantidad indeterminada de interfaces de red, IPs, reglas de *iptables* y procesos *haproxy*. Además, como vimos también en 3.2.1, queríamos evitar en lo posible añadir reglas de *iptables*.

Para retener los paquetes encontramos dos opciones, la primera, comentada en el artículo anteriormente citado de Willy Tarreau, es una implementación de Joseph Lynch para Yelp en la que utiliza los mismos mecanismos de control de tráfico que se utilizan para calidad de servicio [15], con ellos se pueden configurar diferentes colas para diferentes paquetes y aplicarles diferentes políticas denominadas *qdiscs* (*queue disciplines*) [16], una de estas políticas permite que los paquetes se encolen, pero no se desencolen hasta que se les retire esta política. El problema con esta opción es que solo funciona con tráfico saliente, en Yelp pueden utilizarla ya que tienen proxys locales en cada máquina, pero en nuestro caso el proxy y los clientes están en distintas máquinas y no podríamos sincronizar las recargas de configuración de los proxies con todos ellos.

Finalmente se implementó una aproximación parecida, pero utilizando colas de *netfilter* en vez de *qdiscs*. *netfilter* es un framework del kernel Linux utilizado para filtrar y manipular paquetes de red, es por ejemplo utilizado por *iptables*. Una de las funcionalidades de este framework es la de enviar paquetes a un proceso en espacio de usuario con el fin de que se puedan analizar o filtrar con lógicas de más alto nivel [18]. En nuestra implementación[27] los retenemos en el mismo proceso que se encarga de orquestar las recargas de *haproxy* hasta que la recarga ha finalizado.

Con esta implementación conseguimos evitar todas las pérdidas de conexiones y las conexiones retenidas lo son tan solo durante unas decenas de milisegundos.

27 La implementación del sistema de retención de paquetes implementado durante la realización de este proyecto está disponible públicamente [17]

3.2.4 Evaluación del rendimiento tras mejoras en *kubelbs*

Tras aplicar los cambios descritos en los puntos anteriores en los *kubelbs* realizamos nuevas pruebas y observamos varias mejoras:

- La cantidad de peticiones por segundo que el clúster es capaz de atender aumenta visiblemente (Tabla 22 y Tabla 23).
- Mejora la capacidad de escalar horizontalmente al observar un incremento en la cantidad de peticiones por segundo al añadir nuevos nodos (ver Figura 9). Esta mejora es inferior a la deseable, ya que el aumento no es lineal. Además todavía es inferior a la medida de referencia (Tabla 19), se sigue observando una penalización achacable al uso de VXLAN.
- La pérdida de conexiones debidas a recargas de configuración prácticamente desaparece. No tenemos métricas directas para avalar esta afirmación, pero observamos en los sistemas en producción una importante reducción en la cantidad de errores de aplicación registrados que puedan estar relacionados con esta causa, y dejamos de poder reproducir estos errores en los escenarios de pruebas con *haproxy*.

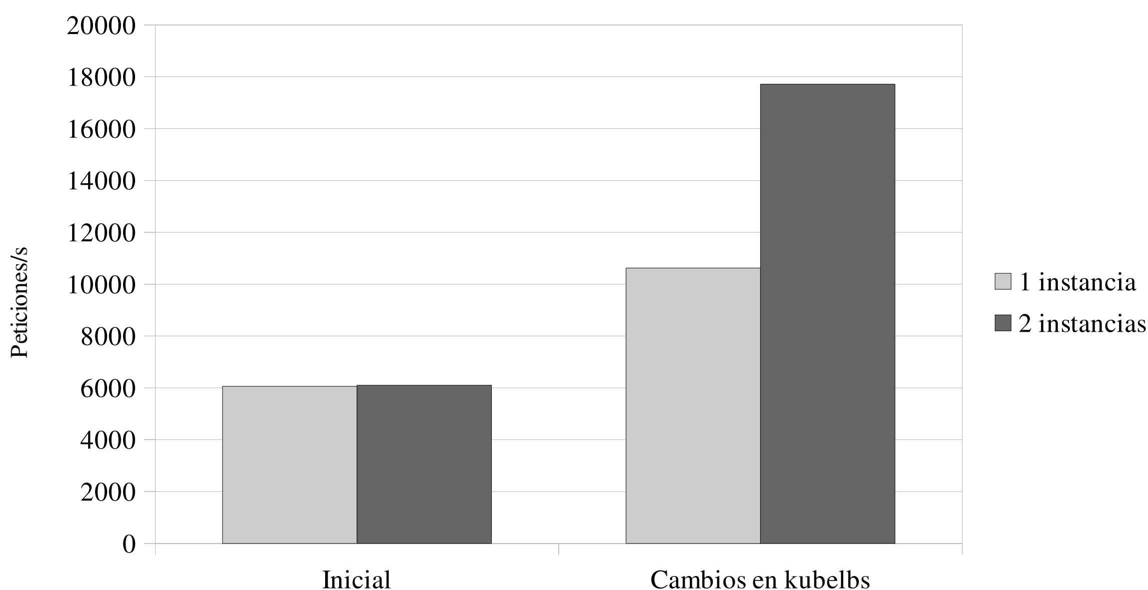

Figura 9: Peticiones/s tras cambios en kubelbs

3.3 Reemplazo del sistema de enrutado

Tras realizar los cambios en los elementos de balanceo de tráfico seguimos viendo que el rendimiento de la red del clúster de *Kubernetes*, aunque suficiente para nuestra carga actual, resulta aún insuficiente para aprovechar totalmente los recursos de los nodos, y carece de la capacidad de escalado que necesitamos para el crecimiento que esperamos. En este capítulo se estudia el proceso de migración de enrutamiento VXLAN a *Host Gateway* de un clúster de *Kubernetes* en producción sin detener el sistema en ningún momento.

3.3.1 Comparativa de VXLAN frente a *Host gateway*

Como prueba de concepto configuramos manualmente un *kubelb* y un nodo en modo *Host gateway* y realizamos pruebas de carga, vemos que los resultados (Tabla 24) se aproximan a la medida de referencia (Tabla 19). Si comparamos ambas medidas con el resto de pruebas realizadas con una única instancia podemos ver claramente la mejora (Figura 10).

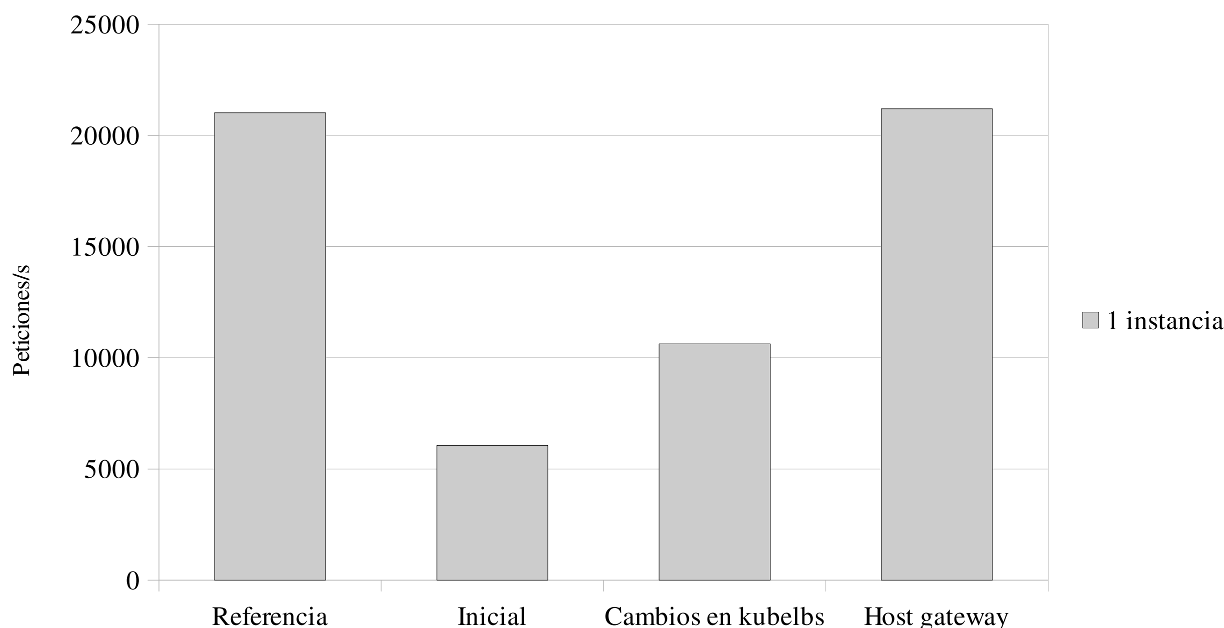

Figura 10: Comparativa de diferentes configuraciones con una instancia

Host gateway tiene la limitación de necesitar que todos los nodos tengan visibilidad en capa 2 (Sec. 2.2.2), pero es algo que cumplen nuestros clústers. En consecuencia, consideramos que merece la pena la migración del sistema de enrutado.

3.3.2 Desarrollo de herramienta de migración

La migración de VXLAN a *Host gateway* conlleva un cambio en la configuración de *flannel*. Como vimos en Sec. 2.2, todos los agentes se configuran utilizando unos mismos ajustes almacenados en *etcd*, una base de datos distribuida. Cada agente, al arrancar, se configura utilizando estos valores. Otro punto importante es que cuando un agente es configurado para utilizar un determinado

backend, únicamente utiliza este *backend*, evitando configurar el nodo para conectar con *pods* en nodos que utilicen otros *backends*. Estos dos puntos nos hacen ver la problemática de migrar un sistema en producción. Para realizar el cambio deberíamos primero cambiar la configuración en *etcd* y después reiniciar nodo a nodo para que utilicen esta nueva configuración, pero al hacer esto tendríamos dos redes separadas, los nodos configurados con VXLAN sólo podrían conectar con nodos con la misma configuración, y a los nodos configurados únicamente con la nueva configuración de *Host Gateway* les resultaría imposible alcanzar los *pods* en VXLAN. Esto no es admisible, ya que produciría problemas de conectividad que afectarían al servicio.

Para solucionar este problema se tendrían que ir añadiendo reglas de enrutado en todos los nodos conforme se fueran reiniciando con la nueva configuración. Para cada nodo se tendría que añadir:

- En el nuevo nodo, una regla por cada uno de los nodos con VXLAN.
- En cada uno de los nodos configurados con VXLAN, una regla para el nuevo nodo reconfigurado. Esta regla tendría que añadirse en la tabla de enrutamiento (como haría el backend *Host Gateway*) ya que no se podría utilizar el encapsulamiento VXLAN al no estar configurada en el nuevo nodo.

Como prueba de concepto se realizó este cambio manualmente para un solo nodo, y se vio que hacerlo con una docena de ellos resultaría realmente engorroso y propenso a errores. Además resultaría peligroso, ya que si un nodo se reinicia inesperadamente durante la operación carecería de estas reglas configuradas manualmente, y dispararía cambios de configuración inesperados. Esta prueba de concepto sirvió de todos modos para validar la mejora de rendimiento que cabría esperar al eliminar la encapsulación VXLAN (Sec. 3.3.1).

Para solucionar este problema se plantea el desarrollo de una herramienta que pueda utilizarse durante la migración de sistemas de enrutado que automatice el mantenimiento de estas reglas que de otro modo se tendrían que añadir manualmente.

Esta herramienta sería como una versión simplificada y complementaria de *flannel* con las siguientes características:

- Debería lanzarse en todos los nodos, tanto aquellos configurados con VXLAN, como en aquellos configurados con *Host Gateway*.
- Como *flannel*, leería su configuración, y las subredes del resto de nodos del almacenamiento distribuido en *etcd*.
- También como *flannel*, configuraría las reglas de enrutado de acuerdo con estas subredes, pero a diferencia de *flannel*, lo haría solo para las subredes de nodos configurados con *backends* diferentes.
- Sólo tendría que implementar la gestión de reglas de enrutado del nodo, ya que es el sistema de enrutado que ambas configuraciones pueden usar, tengan VXLAN o no.

Todas estas características ya están presentes en *flannel*, por lo que se decidió estudiar el código de este software, que está disponible públicamente[28], con el fin de reutilizar código si fuera posible.

Si analizamos el código comenzando por el método principal, se pueden ver muy bien varias de las cosas de las que hemos ido hablando.

Para obtener la configuración de las subredes utiliza un objeto que obtiene mediante un método factoría que le permite soportar múltiples sistemas de coordinación. Cada sistema que se quiera soportar, ha de implementar la interfaz `Manager` definida en el módulo `subnet`. Podemos ver (Tabla 10) que implementar esta interfaz requiere métodos para obtener la configuración de la red (`GetNetworkConfig()`), así como para reservar subredes (métodos `Reservation`), o interactuar con "cesiones" de subredes (métodos `Lease`).

```
type Manager interface {
    GetNetworkConfig(ctx context.Context) (*Config, error)
    AcquireLease(ctx context.Context, attrs *LeaseAttrs) (*Lease, error)
    RenewLease(ctx context.Context, lease *Lease) error
    RevokeLease(ctx context.Context, sn ip.IP4Net) error
    WatchLease(ctx context.Context, sn ip.IP4Net, cursor interface{}) (
        LeaseWatchResult, error)
    WatchLeases(ctx context.Context, cursor interface{}) (LeaseWatchResult, error)

    AddReservation(ctx context.Context, r *Reservation) error
    RemoveReservation(ctx context.Context, subnet ip.IP4Net) error
    ListReservations(ctx context.Context) ([]Reservation, error)
    Name() string
}
```

Tabla 10: Definición de la interfaz `subnet.Manager`[29]

Para la migración estaríamos interesados en el método para obtener la configuración de la red, para saber qué *backend* tenemos activo, y en el método `WatchLeases()` para observar las subredes "cedidas" al resto de nodos.

Vemos que el módulo `subnet/etcdv2` incluye una implementación de esta interfaz para *etcd* versión 2[30].

Si continuamos analizando el método principal, vemos que una vez obtenida la configuración de la red se utiliza de nuevo un método factoría para obtener la implementación del *backend* específico que ha de utilizarse para gestionar la red indicada en la configuración. La implementación de este método factoría y de los diversos *backends* podemos encontrarla como submódulos del módulo `backend`.

---

28 En esta memoria se hace referencia al código de la última versión liberada a julio de 2017, *flannel* 0.8.0, disponible públicamente [19].

29 Vemos que todos los métodos han de aceptar un contexto (`context.Context`), este es un patrón de concurrencia muy habitual en el lenguaje de programación Go. Estos contextos facilitan el paso de información dependiente de contexto, la finalización explícita de llamadas encadenadas o el establecimiento de plazos de ejecución [20].

30 Actualmente `etcd` dispone de dos versiones principales, la versión 2, más sencilla, está ampliamente soportada. La versión 3, orientada a alto rendimiento, cambia la interfaz y el formato de almacenamiento, lo que la hace incompatible con la versión 2. Normalmente los binarios de la versión 3 soportan también la versión 2 teniendo dos almacenamientos separados.

Cada *backend* debe implementar la interfaz `backend.Backend`, con un único método que a su vez debe instanciar un objeto que implemente la interfaz `backend.Network`, y que será el que se encargue finalmente de toda la gestión de la red. Puede verse la definición de estas interfaces en la Tabla 11.

```
type Backend interface {
    // Called when the backend should create or begin managing a new network
    RegisterNetwork(ctx context.Context, config *subnet.Config) (Network, error)
}

type Network interface {
    Lease() *subnet.Lease
    MTU() int
    Run(ctx context.Context)
}
```

Tabla 11: Definición de las interfaces del módulo `backend`

Acabando ya con el método principal, este ejecuta el método `Run()` del objeto `backend.Network`, que estará ejecutándose durante toda la vida del proceso.

Si observamos ahora las implementaciones del método `Run()` de los *backends* con los que estamos trabajando, vemos que tienen una estructura bastante similar. Ambos métodos tienen un bucle de eventos alimentado por el método `WatchLeases()` cuya interfaz vimos en la Tabla 10. El bucle del *backend* de VXLAN es alimentado también por eventos "*miss*" del propio subsistema VXLAN (Sec. 2.2.1), pero no nos centraremos en estos al no ser tan relevantes para migración hacia *Host gateway*. En cuanto a los primeros eventos, en ambos *backends* se invoca una función manejadora que de nuevo tiene una estructura bastante parecida en ambos casos, para cada evento añade o elimina la configuración necesaria para un nodo dependiendo de si se ha unido al clúster o lo ha abandonado.

Algo relevante que podemos ver en las funciones manejadoras de los eventos generados por `WatchLeases()` es el código que se encarga en cada *backend* de asegurarse de que no intente configurar subredes de nodos configurados para utilizar otros *backends*. Por ejemplo en la Tabla 12 vemos el código que ejecuta el *backend* de VXLAN al añadir un nodo, lo primero que hace es comprobar si coincide el tipo de *backend*, y si no, ignora el evento.

```
if event.Lease.Attrs.BackendType != "vxlan" {
    log.Warningf("Ignoring non-vxlan subnet: type=%v", event.Lease.Attrs.BackendType)
    continue
}
var attrs vxlanLeaseAttrs
if err := json.Unmarshal(event.Lease.Attrs.BackendData, &attrs); err != nil {
    log.Error("Error decoding subnet lease JSON: ", err)
    continue
}
nw.routes.set(event.Lease.Subnet, net.HardwareAddr(attrs.VtepMAC))
nw.dev.AddL2(neighbor{
    IP: event.Lease.Attrs.PublicIP,
    MAC: net.HardwareAddr(attrs.VtepMAC)})
```

Tabla 12: Código ejecutado por el backend VXLAN al añadir un nodo

Estas comprobaciones defienden al sistema de situaciones inesperadas, por ejemplo un *backend Host Gateway* no puede enrutar tráfico a un nodo VXLAN si ambos nodos no están en la misma red física, o un nodo con VXLAN no puede utilizar su interfaz virtual para comunicarse con otro nodo que no tenga una de estas interfaces que le permitan desencapsular los paquetes de red. Sin embargo en nuestro escenario queremos precisamente lo contrario, que los *backends* puedan manejar configuraciones mixtas.

Llegados a este punto del análisis podemos valorar una alternativa. En vez de implementar una herramienta de migración que tenga un comportamiento complementario a *flannel* basado en el mismo código, y que tenga que estar ejecutándose simultáneamente, podemos preparar también una versión modificada del mismo *flannel* con estas comprobaciones desactivadas. En el caso de nodos ya configurados con *Host gateway*, al estar en la misma red física, podemos confiar en que podrán enrutar tráfico a nodos configurados con VXLAN para una determinada IP de su subred, es algo que vimos en las pruebas de concepto.

Podemos, por tanto, añadir un parámetro a *flannel* que permita desactivar esta comprobación para *Host gateway*, simplemente modificando su comprobación como se ve en el fragmento de código de la Tabla 13. Tras este cambio los nodos con la nueva configuración podrán enrutar tráfico a *pods* que estén en nodos con la configuración antigua.

```
if !n.mixed && evt.Lease.Attrs.BackendType != "host-gw" {
    log.Warningf("Ignoring non-host-gw subnet: type=%v", evt.Lease.Attrs.BackendType)
    continue
}
```

Tabla 13: Comprobación de tipo de backend modificada para Host gateway

Para nodos configurados con el *backend* de VXLAN no es suficiente con ignorar la comprobación. En estos nodos tenemos que hacer que para nodos con la configuración antigua se siga utilizando esta implementación, pero para nodos con la nueva configuración se utilice la implementación *Host gateway*. Esto es algo que se puede hacer literalmente. Si un agente configurado para utilizar el *backend* de VXLAN recibe un evento para añadir una subred de tipo *Host gateway*, utilizamos esa implementación encapsulándola en un nuevo método de los objetos que implementan la interfaz `backend.Network` para VXLAN. La comprobación quedaría entonces como se puede ver en el código de la Tabla 14. El método `handleAddHostgwSubnetEvent()` haría exactamente lo mismo que hace el backend *Host gateway* cuando tiene que añadir una subred.

Para soportar la posibilidad de revertir la operación, tenemos que asegurarnos de que cuando un nodo vuelva a ser configurado con VXLAN, las reglas de enrutamiento creadas para *Host gateway* son eliminadas, para ello utilizamos el método `hostgwSubnetCleanup()`. De no hacerlo puede haber conexiones que envíen paquetes por la interfaz VXLAN, pero reciban la respuesta por la interfaz física, esto haría que los paquetes de respuesta fueran descartados, ya que estarían utilizando direcciones IP que no se corresponden con la conexión.

```
if nw.mixed {
    if event.Lease.Attrs.BackendType == "host-gw" {
        nw.handleAddHostgwSubnetEvent(event)
        continue
    }

    // It could have been a host-gw subnet changed to vxlan
    nw.hostgwSubnetCleanup(event)
}

if event.Lease.Attrs.BackendType != "vxlan" {
    log.Warningf("Ignoring non-vxlan subnet: type=%v", event.Lease.Attrs.BackendType)
    continue
}
```

Tabla 14: Comprobación de tipo de backend modificada para VXLAN

También los nodos que abandonan el clúster generan eventos de forma implícita cuando las concesiones de sus subredes caducan. En los manejadores de estos eventos realizamos modificaciones similares, en el backend de *Host gateway* no se realiza la comprobación, y en el caso de VXLAN utiliza la implementación de *Host gateway* cuando ha de borrar la configuración de un nodo configurado con este modo.

Finalmente podemos exponer este parámetro como configuración de la red añadiendo un nuevo campo como se puede ver en la Tabla 15.

```
type Config struct {
    Network     ip.IP4Net
    SubnetMin   ip.IP4
    SubnetMax   ip.IP4
    SubnetLen   uint
    BackendType string          `json:"-"`
    Backend     json.RawMessage `json:",omitempty"`
    Mixed       bool
}
```

Tabla 15: Estructura de datos de la configuración de red de flannel tras añadir el campo "Mixed"[31]

Con esta implementación[32] podemos evitarnos tener que desarrollar y desplegar un nuevo agente que complemente a *flannel*, y disponemos de un parámetro de configuración para controlar el comportamiento deseado de nuestra versión modificada de *flannel*. Realizamos pruebas en entornos virtualizados con esta nueva versión y vemos que funciona perfectamente, por lo que lo consideramos suficiente para realizar la migración de los clústers de producción.

Podría integrarse en el código oficial de *flannel* una versión de estos cambios más general e independiente de nuestro escenario, pero queda fuera del alcance del proyecto y no se ha estimado por ahora.

31 En el lenguaje de programación Go la librería de codificación JSON puede convertir directamente objetos representados en JSON a objetos Go cuando los nombres y tipos de sus campos coinciden. Tipos complejos pueden definir sus propios métodos de decodificación [21].

32 La implementación completa está disponible públicamente [22].

3.3.3 Migración de VXLAN a *Host gateway*

A partir de la versión de *flannel* desarrollada como se describe en la sección anterior se prepara un plan de migración de los clústers de producción. Este plan está diseñado para poder aplicarse de forma progresiva, sin parada total del sistema, y con posibilidad de ser revertido en cualquier punto.

Consiste en las siguientes fases:

1. Cambio de configuración en *etcd* para añadir el parámetro `Mixed` con valor a `true` como se puede ver en la Tabla 16, de modo que la nueva implementación quede activa cuando la nueva versión sea desplegada.
2. Despliegue de la nueva versión de *flannel* nodo a nodo.
3. Cambio de configuración para utilizar el *backend* de tipo *Host gateway* como en la Tabla 17 (recordemos que los agentes de *flannel* seguirán utilizando la configuración que leyeron cuando fueron lanzados hasta que sean reiniciados).
4. Reinicio controlado de nodos, uno a uno, comprobando en cada uno que todo funciona correctamente, hasta que un 25% de los nodos esté utilizando el modo *Host gateway*.
5. Valoración tras un tiempo de entre una y tres semanas del estado de la operación, en especial comprobar que la nueva implementación no está produciendo nuevas incidencias en los sistemas en producción.
6. Reinicio controlado, uno a uno, del resto de nodos del clúster.
7. Cuando se considere, despliegue de nuevo de una versión oficial de *flannel*, y eliminación del parámetro *Mixed* de la configuración como en el ejemplo de la Tabla 18.

Se soporta también la posibilidad de revertir el proceso de migración en cualquiera de sus puntos si se encontrara algún problema grave, para ello se tendrían que seguir los siguientes pasos:

1. Cambio de configuración para utilizar el *backend* de VXLAN.
2. Reinicio controlado, uno a uno, de los nodos que ya hubieran sido migrados.
3. Eliminación del parámetro *Mixed* de la configuración.
4. Despliegue de la antigua versión de *flannel*, nodo a nodo.

```
{
    "Network": "172.20.0.0/16",
    "Backend": {
        "Type": "vxlan"
    },
    "Mixed": true
}
```

Tabla 16: Ejemplo de configuración de flannel, para VXLAN, con el parámetro Mixed

```
{
    "Network": "172.20.0.0/16",
    "Backend": {
        "Type": "host-gw"
    },
    "Mixed": true
}
```

Tabla 17: Ejemplo de configuuración de flannel, para Host Gateway, con el parámetro Mixed

```
{
    "Network": "172.20.0.0/16",
    "Backend": {
        "Type": "host-gw"
    }
}
```

Tabla 18: Ejemplo de configuración de flannel, para Host Gateway, sin el parámetro Mixed

3.3.4 Evaluación del rendimiento tras cambios en el sistema de enrutado

En el momento de escribir esta memoria todavía no se ha realizado la migración en los clusters con la carga principal de producción debido a cambios en las prioridades de la empresa, por lo que no se puede realizar una evaluación del impacto de este cambio más allá de las pruebas de concepto realizadas y la comparativa previa descrita en la sección 3.3.1.

En todo caso, la operación estaría preparada gracias al desarrollo de este proyecto y las pruebas realizadas han sido satisfactorias. Se espera migrar el clúster principal de Madrid a lo largo del último trimestre de 2017.

4 Metodología

Este proyecto se ha realizado como parte del trabajo habitual del equipo SRE[33] de Tuenti, y por tanto se ha seguido su metodología de trabajo. En Tuenti hay una gran flexibilidad en cuanto a la metodología que cada equipo puede seguir. En general se mezclan conceptos del método *kanban*[34] y del modelo *Scrum*[35] con el fin de que el desarrollo de los proyectos sea ágil e incremental. En algunos equipos se utiliza también el llamado modelo "*Scrumban*", que es una mezcla más formal de los métodos enunciados anteriormente y que es más adecuada para equipos que se encargan tanto del desarrollo como del mantenimiento de sus productos, como es el caso de la mayoría de equipos en esta compañía.

Centrándonos más en SRE, el equipo en el que se desarrolla este proyecto, nos encontramos con un equipo que realiza tanto operaciones como desarrollos de software orientados principalmente a la automatización y gestión de estas operaciones. El hecho de que sea un equipo de operaciones hace que muchas veces la carga de trabajo sea impredecible, ya que puede verse afectada por necesidades urgentes e imprevistas de otros equipos o por incidentes que afecten a los sistemas en producción.

La metodología del equipo, al igual que la de otros equipos de la compañía, toma conceptos de otros métodos, y se fundamenta en estas herramientas:

- Panel de tareas, organizado por columnas al estilo de *kanban*: un *backlog* ordenado por prioridades, estado de las tareas en desarrollo y listado de tareas completadas.
- *Sprints* semanales. No es habitual tener *sprints* tan cortos en equipos de desarrollo, pero como se ha comentado, en un equipo con carga de trabajo operativa es habitual que surjan imprevistos, por lo que cada semana se revisan las prioridades.
- Reunión semanal de planificación, en la que se revisan las prioridades del equipo y se estiman o reestiman las tareas para la semana.
- Reunión semanal de retrospectiva, en la que se analiza lo que ha ido mal y lo que ha ido bien, tanto en las tareas mismas como en los procesos. Esta reunión es útil para buscar mejoras en la forma de abordar las distintas tareas e intentar evitar que se repitan incidencias.

Las tareas de este proyecto se han ido introduciendo en la planificación habitual del equipo.

33 SRE – *Site Reliability Engineering*, término acuñado por Google y adoptado por muchas otras empresas para denominar a equipos de operaciones integrados por ingenieros de *software*. En estos equipos se busca abordar problemas o tareas habituales de operaciones como proyectos *software*, con el objetivo de aumentar la escalabilidad operativa mediante la automatización.

34 *Kanban* es un método para coordinar y visualizar el estado de un conjunto de tareas. Se basa en un panel separado por columnas, cada una de ellas representando un estado, y en las que se van colocando tarjetas que representan tareas concretas [23].

35 *Scrum* es un modelo de trabajo en equipo utilizado especialmente en desarrollo de software. Está diseñado para equipos de menos de diez personas y plantea un desarrollo de producto incremental e iterativo basado en ciclos de unas dos semanas llamados "*sprints*" tras los cuales se hace una "entrega". Son también características de este modelo las "s*tand-ups*", reuniones diarias rápidas para hacer seguimiento del progreso de las tareas [24].

5 Conclusiones

La Ingeniería Informática dota a sus titulados de un abanico de conocimientos que sirve como herramienta clave a la hora de enfrentarse a nuevos problemas y desafíos en proyectos en los que se involucren sistemas de computación. En un mundo de "nubes", cada vez más lleno de frameworks y abstracciones de alto nivel, la aplicación práctica de algunos de estos conocimientos parece quedar muy lejana, y es a veces complicado entender la importancia de comprender protocolos de comunicaciones, patrones de desarrollo de sistemas concurrentes o los parámetros del núcleo de un sistema operativo.

En este proyecto, en el que trabajamos con un sistema que pretende exponer a desarrolladores todo un centro de datos como una única reserva de recursos de cómputo, podemos ver que estas abstracciones de tan alto nivel están construidas en realidad sobre estos mismos conocimientos fundamentales, y que un análisis basado en ellos puede llevarnos a hacer un mejor uso de los recursos de los que disponemos.

Las mejoras que se han evaluado y aplicado en este proyecto nos han llevado por un recorrido por muchas de las áreas que forman la Ingeniería Informática. Se ha estudiado el funcionamiento de un sistema distribuido complejo. Se ha medido la capacidad de la red del sistema. Se han visto las implicaciones de utilizar unos protocolos de comunicaciones u otros o los efectos del manejo de un número excesivo de interrupciones en un sistema real. Se han configurado y administrado sistemas en explotación, teniendo siempre presente que lo más importante de estos sistemas es su fiabilidad y estabilidad. Se ha analizado y modificado código tanto propio como ajeno, recordándonos que gran parte del tiempo trabajamos con sistemas legados y es importante entenderlos.

Todo ello en el contexto de una empresa real, trabajando sobre una plataforma que da actualmente servicio a cientos de miles de clientes, dejando ver el impacto del esfuerzo realizado.

Los resultados han sido bastante satisfactorios, quedando solucionados los problemas más graves, las incidencias relacionadas con los aspectos tratados en este proyecto prácticamente han desaparecido. Queda pendiente la aplicación del cambio del sistema de enrutado debido a cambios de prioridades en la empresa, pero esta operación queda lista para realizarse en cualquier momento, y servirá para mejorar la capacidad de escalar el sistema.

Esta memoria queda también como documentación de referencia para futuros cambios que puedan necesitarse en este o en otros despliegues similares.

Bibliografía

[1]: Editores de la Wikipedia (Consultado en 2017), *Arquitectura de microservicios*, https://es.wikipedia.org/wiki/Arquitectura_de_microservicios

[2]: The Kubernetes Authors (Consultado en 2017), *What is Kubernetes?*, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[3]: Editores de la Wikipedia (Consultado en 2017), *Protocol Buffers*, https://en.wikipedia.org/wiki/Protocol_Buffers

[4]: The Kubernetes Authors (Consultado en 2017), *Kubernetes Components*, https://kubernetes.io/docs/concepts/overview/components/

[5]: Soriano Pastor, Jaime y otros (2017), *Kube2lb*, https://github.com/tuenti/kube2lb

[6]: The Kubernetes Authors (Consultado en 2017), *Kubernetes Cluster Networking*, https://kubernetes.io/docs/concepts/cluster-administration/networking/

[7]: Editores de la Wikipedia (Consultado en 2017), *Traducción de direcciones de red*, https://es.wikipedia.org/wiki/Traducci%C3%B3n_de_direcciones_de_red

[8]: Varios (Consultado en 2017), *Flannel project in github*, https://github.com/coreos/flannel

[9]: Varios miembros del IETF (2014), *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*, https://tools.ietf.org/html/rfc7348

[10]: Varios (Consultado en 2017), *ksoftirqd(9)*, https://man.cx/ksoftirqd(9)

[11]: Intel Corporation (2014), *Product Brief Intel® Ethernet Server Adapter I350*, https://www.intel.com/content/www/us/en/ethernet-products/gigabit-server-adapters/ethernet-i350-server-adapter-brief.html

[12]: Tarreau, Willy (2017), *Manual de configuración de haproxy 1.7*, http://cbonte.github.io/haproxy-dconv/1.7/configuration.html

[13]: Soriano Pastor, Jaime (2017), *Timeouts configuration for haproxy backends in kube2lb project*, https://github.com/tuenti/kube2lb/pull/12

[14]: Tarreau, Willy (2017), *Trully seamless reloads with Haproxy*, https://www.haproxy.com/blog/truly-seamless-reloads-with-haproxy-no-more-hacks/

[15]: Lynch, Joseph (2015), *True Zero Downtime HAProxy Reloads*, https://engineeringblog.yelp.com/2015/04/true-zero-downtime-haproxy-reloads.html

[16]: Hubert, Bert (1999), *Linux Traffic Control manual*, https://linux.die.net/man/8/tc

[17]: Tuenti Technologies S.L. (2017), *Implementación de retención de paquetes basada en netqueue*, https://github.com/tuenti/haproxy-docker-wrapper/blob/master/netqueue.go

[18]: Welte, Harald y otros (2005), *The libnetfilter_queue project*, http://netfilter.org/projects/libnetfilter_queue/index.html

[19]: Varios (2017), *flannel 0.8.0*, https://github.com/coreos/flannel/tree/v0.8.0

[20]: Ajmani, Sameer (2014), *Go concurrency patterns: Context*, https://blog.golang.org/context

[21]: Gerrand, Andrew (2011), *JSON and Go*, https://blog.golang.org/json-and-go

[22]: Soriano Pastor, Jaime (2017), *Modificaciones en flannel para soportar despliegues mixtos de VXLAN y Host Gateway*, https://github.com/jsoriano/flannel/tree/vxlan-to-hostgw-migration

[23]: Editores de la Wikipedia (Consultado en 2017), *Kanban (desarrollo)*, https://es.wikipedia.org/wiki/Kanban_(desarrollo)

[24]: Editores de la Wikipedia (Consultado en 2017), *Scrum (desarrollo de software)*, https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software)

Anexo I: Ejecuciones de pruebas de carga

```
$ ab -c 32 -n 60000 http://10.95.5.108/

Server Software:        nginx/1.13.1
Server Hostname:        10.95.5.108
Server Port:            80

Document Path:          /
Document Length:        612 bytes

Concurrency Level:      32
Time taken for tests:   2.855 seconds
Complete requests:      60000
Failed requests:        0
Total transferred:      50700000 bytes
HTML transferred:       36720000 bytes
Requests per second:    21017.96 [#/sec] (mean)
Time per request:       1.523 [ms] (mean)
Time per request:       0.048 [ms] (mean, across all concurrent requests)
Transfer rate:          17343.92 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.2      1       3
Processing:     0    1   0.3      1       4
Waiting:        0    1   0.3      1       4
Total:          0    2   0.5      1       5
ERROR: The median and mean for the total time are more than twice the standard
       deviation apart. These results are NOT reliable.

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      2
  75%      2
  80%      2
  90%      2
  95%      3
  98%      3
  99%      3
 100%      5 (longest request)
```

Tabla 19: Pruebas de carga contra una instancia desplegada fuera de Kubernetes

```
$ ab -c 32 -n 60000 -H "Host: benchmark.playground.svc.mad.tuenti.int" http://10.95.5.108/

Server Software:        nginx/1.13.1
Server Hostname:        10.95.5.108
Server Port:            80

Document Path:          /
Document Length:        612 bytes

Concurrency Level:      32
Time taken for tests:   9.902 seconds
Complete requests:      60000
Failed requests:        0
Total transferred:      50700000 bytes
HTML transferred:       36720000 bytes
Requests per second:    6059.08 [#/sec] (mean)
Time per request:       5.281 [ms] (mean)
Time per request:       0.165 [ms] (mean, across all concurrent requests)
Transfer rate:          4999.93 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.3      1      12
Processing:     2    5   9.7      4     209
Waiting:        2    5   9.7      4     209
Total:          2    5   9.7      5     210

Percentage of the requests served within a certain time (ms)
  50%      5
  66%      5
  75%      5
  80%      5
  90%      6
  95%      6
  98%      7
  99%      7
 100%    210 (longest request)
```

Tabla 20: Pruebas de carga con configuración inicial, 1 instancia

```
$ ab -c 32 -n 120000 -H "Host: benchmark.playground.svc.mad.tuenti.int" http://10.95.5.108/


Server Software:        nginx/1.13.1
Server Hostname:        10.95.5.108
Server Port:            80

Document Path:          /
Document Length:        612 bytes

Concurrency Level:      32
Time taken for tests:   19.658 seconds
Complete requests:      120000
Failed requests:        0
Total transferred:      101400000 bytes
HTML transferred:       73440000 bytes
Requests per second:    6104.51 [#/sec] (mean)
Time per request:       5.242 [ms] (mean)
Time per request:       0.164 [ms] (mean, across all concurrent requests)
Transfer rate:          5037.41 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.4      1      45
Processing:     1    5  10.8      4    1031
Waiting:        1    5  10.8      4    1031
Total:          1    5  10.8      5    1031

Percentage of the requests served within a certain time (ms)
  50%      5
  66%      5
  75%      5
  80%      6
  90%      6
  95%      7
  98%      7
  99%      7
 100%   1031 (longest request)
```

Tabla 21: Pruebas de carga con configuración inicial, 2 instancias

```
ab -c 32 -n 60000 -H "Host: benchmark.playground.svc.mad.tuenti.int" http://10.95.5.108/

Server Software:        nginx/1.13.1
Server Hostname:        10.95.5.108
Server Port:            80

Document Path:          /
Document Length:        612 bytes

Concurrency Level:      32
Time taken for tests:   5.647 seconds
Complete requests:      60000
Failed requests:        0
Total transferred:      50700000 bytes
HTML transferred:       36720000 bytes
Requests per second:    10624.96 [#/sec] (mean)
Time per request:       3.012 [ms] (mean)
Time per request:       0.094 [ms] (mean, across all concurrent requests)
Transfer rate:          8767.67 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0      10
Processing:     1    3   1.7      3      77
Waiting:        1    3   1.7      3      77
Total:          2    3   1.7      3      77

Percentage of the requests served within a certain time (ms)
  50%      3
  66%      3
  75%      3
  80%      3
  90%      3
  95%      4
  98%      4
  99%      4
 100%     77 (longest request)
```

Tabla 22: Pruebas de carga después de cambios en los kubelbs, 1 instancia

```
$ ab -c 32 -n 120000 -H "Host: benchmark.playground.svc.mad.tuenti.int" http://10.95.5.108/

Server Software:        nginx/1.13.1
Server Hostname:        10.95.5.108
Server Port:            80

Document Path:          /
Document Length:        612 bytes

Concurrency Level:      32
Time taken for tests:   6.774 seconds
Complete requests:      120000
Failed requests:        0
Total transferred:      101400000 bytes
HTML transferred:       73440000 bytes
Requests per second:    17716.08 [#/sec] (mean)
Time per request:       1.806 [ms] (mean)
Time per request:       0.056 [ms] (mean, across all concurrent requests)
Transfer rate:          14619.22 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       2
Processing:     1    2   0.6      2      41
Waiting:        1    2   0.6      2      41
Total:          1    2   0.6      2      42

Percentage of the requests served within a certain time (ms)
  50%      2
  66%      2
  75%      2
  80%      2
  90%      2
  95%      2
  98%      2
  99%      3
 100%     42 (longest request)
```

Tabla 23: Pruebas de carga después de los cambios en kubelbs, 2 instancias

```
$ ab -c 32 -n 60000 -H "Host: benchmark.playground.svc.mad.tuenti.int" http://10.95.5.108/

Server Software:        nginx/1.13.1
Server Hostname:        10.95.5.108
Server Port:            80

Document Path:          /
Document Length:        612 bytes

Concurrency Level:      32
Time taken for tests:   2.830 seconds
Complete requests:      60000
Failed requests:        0
Total transferred:      50700000 bytes
HTML transferred:       36720000 bytes
Requests per second:    21203.88 [#/sec] (mean)
Time per request:       1.509 [ms] (mean)
Time per request:       0.047 [ms] (mean, across all concurrent requests)
Transfer rate:          17497.34 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       3
Processing:     1    1   1.5      1      64
Waiting:        1    1   1.5      1      64
Total:          1    1   1.5      1      64

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      2
  75%      2
  80%      2
  90%      2
  95%      2
  98%      2
  99%      2
 100%     64 (longest request)
```

Tabla 24: Pruebas de carga configurando el kubelb con host gateway, sin VXLAN