UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Despliegue del núcleo de red 5G en Kubernetes sobre plataforma de bajo coste

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Iñesta Hernández, Borja

Tutor/a: Moltó Martínez, Germán

Director/a Experimental: Caballer Fernández, Miguel

CURSO ACADÉMICO: 2023/2024

Resumen

Este Trabajo de Fin de Máster se centrará en la instalación y configuración de un clúster de *Kubernetes* multinodo sobre una plataforma de bajo coste, en este caso *Raspberry Pi*. La instalación de *Kubernetes* y sus prerrequisitos será automatizada utilizando *roles* de *Ansible*. Tras la instalación, el clúster será utilizado para el despliegue de un núcleo de red 5G con *Helm*, utilizando el proyecto *Open5GS* sobre contenedores *Docker*.

Utilizando el núcleo de red 5G seremos capaces de verificar el correcto funcionamiento de todos los componentes del clúster y realizaremos pruebas de rendimiento. Estas pruebas serán utilizadas para realizar una comparativa de coste y rendimiento del mismo despliegue sobre la plataforma de bajo coste y sobre un proveedor de *cloud* público.

El objetivo del proyecto es familiarizarse con despliegues sobre *Kubernetes*, ofrecer una guía de instalación y un análisis de la viabilidad del uso de plataformas de bajo coste para este tipo de aplicaciones.

Palabras clave: Kubernetes, 5G, Docker, Raspberry Pi.

Abstract

This Master's Thesis focus is about the installation and configuration of a multi-node *Kubernetes* cluster on top of a low cost platform, in specific *Raspberry Pi*. The installation of *Kubernetes* and its prerequisites will be automated using *Ansible roles*. After the installation, the cluster will host a 5G core network deployed with *Helm*, using the *Open5GS* project running on *Docker* containers.

Using the 5G network core we will be able to verify the cluster components and carry performance tests. These tests will be used to create a comparison between costs and performance using the same deployment running on the low-cost platform versus a public *cloud* provider.

The goal of this project is to learn *Kubernetes* deployments, provide an installation guide and an analyse the feasibility of using low-cost platforms for this kind of applications.

Keywords: Kubernetes, 5G, Docker, Raspberry Pi.

Tabla de contenidos

Índice de figuras

1. Introducción

La virtualización es el paradigma que habilitó la creación de la computación en la nube. *IBM* fue pionero en este ámbito, desarrollando a finales de la década de 1960 lo que pueden ser considerados los primeros sistemas basados en máquinas virtuales [1]. Los avances en la virtualización, desde la evolución de los hipervisores hasta la virtualización de recursos *hardware* como redes o almacenamiento, junto con la mejora del *hardware* y la incorporación de instrucciones específicas para la virtualización en las *CPUs*, aumentaron la adopción de la virtualización y trajeron consigo la creación de la computación en el *cloud*.

Muchas arquitecturas de computación distribuida han precedido la creación de la computación en el *cloud*, aunque este concepto no se popularizó hasta que *Amazon* con *Amazon Web Services* [2] (creada en 2002) lanzó en 2006 sus servicios de *S3* [3] para el almacenamiento en el *cloud* y *EC2* [4] para la creación de máquinas virtuales. Este hito puede ser considerado el nacimiento de la primera *Infrastructure as a Service* (*IaaS*), uno de los modelos de servicio de la computación *cloud*.

Posteriormente, otras empresas como *Google* con *Google Cloud Platform* [5] (creada en 2008) o *Microsoft* con *Azure* [6] (creada en 2010) surgieron en el mercado de la computación en el *cloud* ofreciendo productos y servicios similares. Un hito importante fue la creación de *OpenStack* [7] en 2010, un proyecto de código abierto diseñado para la creación de *clouds* privados.

Otro concepto que tiene sus raíces en el paradigma de la virtualización son los contenedores. Los contenedores, también llamados virtualización a nivel de sistema operativo o virtualización ligera, son un conjunto de tecnologías que utilizan varias técnicas de virtualización para aislar los recursos de aplicaciones que se ejecutan en un entorno compartido, donde comparten el núcleo del sistema operativo.

Aunque tecnologías relacionadas con los contenedores existen muchos años antes como *chroot* o *jails*, el concepto de contenedor no surgió hasta 2005 con *OpenVZ* y no se consolidó hasta 2008 con la aparición de los *Linux Containers* (*LXC*). Sin embargo, no fue hasta 2013, con la creación de *Docker* [8], cuando el concepto de contenedor se popularizó y sentó las bases de lo que conocemos actualmente como contenedores.

En 2014, con la popularidad alcanzada por los contenedores, *Google* lanzó *Kubernetes* [9], un proyecto diseñado para la orquestación y gestión de aplicaciones basadas en contenedores.

La virtualización ha desempeñado un papel fundamental en el desarrollo de 5G proporcionando avances clave en el despliegue de las redes, eficiencia en el uso de recursos y la escalabilidad. Facilita un despliegue ágil y flexible de las redes al ser implementadas en *software*. Reduce los gastos de gestión y mantenimiento gracias a compartir recursos *hardware* entre varios despliegues y al utilizar *hardware COTS* (*Commercial Off-the-self*). También permite el uso de herramientas *software* para la replicación de los despliegues proporcionando escalabilidad dinámica basada en la demanda y una mayor disponibilidad en caso de fallos.

Es por esto que este Trabajo de Fin de Máster (TFM) busca utilizar varias de las tecnologías mencionadas anteriormente para desplegar un núcleo de red 5G basado en contenedores en dos entornos diferentes. Un entorno *bare-metal on-premises* utilizando *SBCs* (*Single-board Computer*) y un entorno basado en máquinas virtuales utilizando un proveedor de *cloud* público. En ambos casos, la aplicación será gestionada por un orquestador de contenedores.

1.1 Motivación

Como miembro del grupo de comunicaciones móviles (*MCG*) del *iTEAM* [10] (*Instituto de Telecomunicaciones y Aplicaciones Multimedia*), la motivación de este TFM es la exploración de varias tecnologías de virtualización aplicadas a las redes móviles, en concreto al despliegue de un núcleo de red 5G.

Las redes de telefonía móvil también han evolucionado significativamente desde sus inicios a finales de la década de 1970. Uno de los cambios más recientes, impulsado por 5G, ha sido la transición de componentes basados en *hardware* en las tecnologías 2G, 3G y 4G a componentes basados en *software*. Esta implementación basada en *software* conocida como *Network Function Virtualization* (*NFV*) permite su despliegue en entornos virtualizados, utilizando máquinas virtuales o contenedores, lo que ofrece los beneficios de estos entornos como un mejor aprovechamiento de recursos y la reducción de costes, entre otros.

Estos cambios acercan las redes de telecomunicaciones a los entornos de desarrollo de *software* modernos, permitiendo aprovechar muchas de sus ventajas y herramientas existentes, aunque también suponen un gran cambio en el despliegue y la gestión de estas redes.

En términos generales, este TFM tiene como objetivo aplicar estas nuevas tecnologías en el ámbito de las comunicaciones móviles y documentar ese proceso.

1.2 Objetivos

El objetivo principal de este TFM es desplegar un núcleo de red 5G sobre una plataforma de gestión de contenedores. Para ello se hará uso de dos entornos diferentes. Un entorno *bare-metal on-premises* y un entorno basado en máquinas virtuales sobre un proveedor de *cloud* público, con los que se realizará una comparativa de rendimiento y coste.

Para alcanzar este objetivo se han definido los siguientes subobjetivos:

- Automatización de la instalación de la plataforma de gestión de contenedores
- Configuración de la plataforma de gestión de contenedores
- Integración y despliegue del núcleo de red 5G en la plataforma de gestión de contenedores
- Comparativa de rendimiento y coste entre los dos entornos en los que se desplegará la aplicación (*on-premises*, *cloud* público)

Cabe mencionar que además de estos objetivos se busca aprender las tecnologías seleccionadas para cumplir los objetivos, tratar de mejorar procesos como los despliegues de contenedores en mi entorno de trabajo y documentar todo este proceso.

1.3 Metodología

Para este proyecto se ha seguido una metodología de desarrollo iterativa en la que tras completar una iteración completa del proyecto para el entorno *on-premises*, se repite para el entorno basado en el *cloud* público. Esto permite completar como mínimo dos iteraciones del proyecto completo antes de ser finalizado y realizar las pruebas. También se ha iterado dos veces sobre la adaptación de la aplicación desplegada en Kubernetes. Una primera iteración para la adaptación de Docker Compose para ser desplegada en Kubernetes y una segunda iteración sobre el despliegue de Kubernetes para ser adaptado a Helm, así como otros aspectos del desarrollo en el que se han incluido mejoras conforme avanzaba el proyecto.

1.4 Estructura de la memoria

El siguiente apartado explica la estructura del presente documento:

- Capítulo 2 (Estado del Arte y Tecnologías Relacionadas): Menciona las tecnologías relevantes en el contexto de este TFM y muestra un análisis de proyectos actuales que las implementan. En él también se exponen las razones detrás de la elección de cada una de las tecnologías utilizadas.

- Capítulo 3 (Desarrollo): Explica el proceso de instalación de la plataforma de gestión de contenedores, su automatización, la integración del núcleo de red 5G a la plataforma y su despliegue. Este desarrollo se realiza tanto para la infraestructura *on-premises* así como para la infraestructura de *cloud* público.
- Capítulo 4 (Validación): Aborda las pruebas realizadas con herramientas de testeo para la validación del correcto funcionamiento del núcleo de red 5G y la obtención de métricas relevantes para la comparativa entre infraestructuras.
- Capítulo 5 (Conclusiones y Trabajos Futuros): Muestra las conclusiones obtenidas tras los resultados de las pruebas y menciona aspectos en los que continuar trabajando en el contexto de este proyecto.
- Capítulo 6 (Bibliografía): Recopila las referencias utilizadas a lo largo del TFM.
- Capítulo 7 (Anexo): Esta sección contiene información extra citada en los capítulos relevantes.

2. Estado del Arte y Tecnologías Relacionadas

2.1 Infraestructura

A nivel de infraestructura existen múltiples opciones que pueden ser objeto de estudio, desde configuraciones *on-premises*[1] basadas en servidores utilizando tecnologías de virtualización hasta despliegues en el *cloud*. Para abarcar el mayor número de escenarios posible, en este TFM se ha decidido implementar dos despliegues muy diferenciados: un despliegue *bare-metal*[2] *on-premises* y un despliegue virtualizado en un proveedor de *cloud* público. El concepto bare-metal implica la instalación del software haciendo uso

El despliegue *bare-metal on-premises* permite una interacción directa con el *hardware*, utilizando componentes de red para la interconexión y desplegando el *software* en almacenamiento local. Este enfoque proporciona una experiencia completa de gestión de infraestructuras, permitiendo un control granular sobre los recursos físicos y la optimización de su rendimiento en un entorno controlado.

El despliegue en *cloud* público proporciona la experiencia de gestionar máquinas virtuales alojadas en el *cloud*, creando una arquitectura basada en los recursos disponibles en la plataforma *cloud* seleccionada. Esta opción permite una administración simplificada y una escalabilidad flexible de los recursos, a costa de un control menos granular.

Esta decisión tiene como objetivo comparar las diferencias en la operación de estos dos tipos de despliegue.

Para la infraestructura *on-premises*, existen numerosas posibilidades según el presupuesto. Una opción sería adquirir un servidor capaz de ejecutar un hipervisor y basar la infraestructura en máquinas virtuales. Sin embargo, este despliegue busca no solo la ejecución *on-premises*, sino también sobre *bare-metal*, lo que implica la compra individual de cada uno de los equipos.

Para ello, se ha optado por basar esta infraestructura en computadores de placa única o *Single Board Computers* (*SBCs*), una opción económica que permite gestionar las máquinas sobre *bare-metal* ocupando un espacio muy reducido, permitiéndonos mantener esta infraestructura cómodamente en casa.

[1] El concepto *on-premises* implica la gestión del *hardware* en las propias instalaciones.
[2] El concepto *bare-metal* implica la instalación del *software* directo sobre una máquina física dedicada.

Aunque la adquisición de un servidor o una máquina más potente podría resultar más económica que la compra de cuatro *SBCs* se busca el despliegue de una infraestructura en la que se haga gestión de todo el *hardware*, incluido los elementos de interconexión para la red.

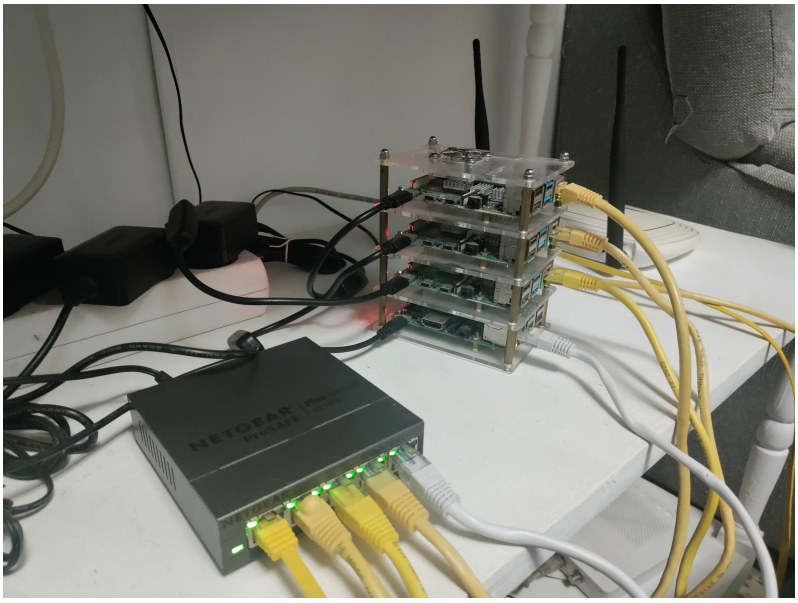

Figura 1: Infraestructura on-premises

Para la infraestructura basada en proveedores de *cloud* público, se han analizado los proveedores de *cloud* público más utilizados en la actualidad. En 2024, la suma de los tres principales proveedores *cloud* (*AWS*, *Azure*, *GCP*) representa un 67% del mercado [11] mientras que otros como *Alibaba Cloud* o *IBM Cloud*, representan un 4% y un 2% respectivamente.

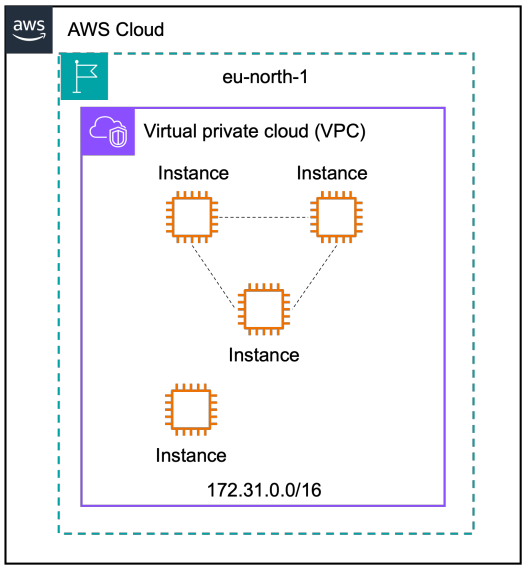

Figura 2: Infraestructura cloud

Entre los tres principales proveedores de *cloud* público, se ha escogido *AWS*. La decisión se ha basado en pequeños detalles, ya que todos ellos disponen

de los servicios necesarios para albergar nuestra infraestructura. *AWS* es actualmente el proveedor *cloud* líder con mayor cuota de mercado. Además, fue escogido en parte por la experiencia adquirida en el Máster, donde proporcionó la mejor experiencia de usuario entre los tres. Otro punto importante es la existencia de su plan gratuito de 12 meses para alguno de sus servicios, minimizando así el coste de las pruebas a realizar por este TFM.

2.2 SBCs

Los *SBCs* empezaron a popularizarse a partir de 2012 con la creación de *Raspberry Pi* [12]. *Raspberry Pi* es una empresa que desarrolla una familia de productos *SBC* de bajo coste y bajo consumo con *CPUs* basadas en la arquitectura *ARM*.

En la actualidad, existen muchas marcas con diferentes modelos de *SBCs* en el mercado, como pueden ser *Orange Pi* [13], *ODROID* [14] o *Nvidia Jetson* [15]. Algunos modelos de *Orange Pi* son más económicos, *ODROID* tiene modelos con *CPUs* y *GPUs* con un mayor rendimiento y los productos de *Nvidia Jetson* suelen ser utilizados para aplicaciones que hacen uso de la *Inteligencia Artificial* (*IA*) por contener *GPUs* más potentes.

La elección de *Raspberry Pi* como *hardware* para el clúster *on-premises* se ha basado en su madurez como proyecto. Como pionero en el campo de los *SBCs*, *Raspberry Pi* lleva años ofreciendo productos ampliamente testeados que garantizan su compatibilidad, además de proporcionar un sistema operativo propio basado en la distribución *GNU/Linux Debian* y una documentación muy útil para configurar sus dispositivos.

2.3 Plataformas de gestión y configuración

Un elemento fundamental para el desarrollo de este TFM es la plataforma de gestión y configuración. Esta plataforma permitirá automatizar de forma centralizada la configuración de las máquinas que conforman la infraestructura, sea *on-premises* o en el *cloud*. Mediante esta herramienta instalaremos la plataforma de gestión de contenedores en las máquinas que formarán parte de nuestro clúster. Existen varias herramientas destinadas a la gestión y configuración de máquinas remotas, entre las más populares de hoy en día se encuentran *Ansible* [16], *Salt* [17], *Chef* [18] o *Puppet* [19].

Las principales diferencias entre estas herramientas, además del lenguaje y la sintaxis que utilizan, residen en su arquitectura. *Ansible* utiliza una arquitectura sin agente, *Chef* y *Puppet* utilizan una arquitectura con agente y *Salt* puede ser configurado tanto con agente como sin agente. Tanto *Ansible* como *Salt* están basados en *Python* y utilizan *YAML*, *Chef* y *Puppet* están basados en *Ruby* y ambos utilizan su propio *Domain Specific Language* (*DSL*).

Se optará por una arquitectura sin agente para simplificar su instalación, puesto que las dos infraestructuras a gestionar son relativamente simples. Esto descarta el uso de *Chef* y *Puppet*. Entre las dos plataformas restantes se ha decidido utilizar *Ansible* ya que en la actualidad cuenta con una mayor comunidad lo que facilitará obtener un mayor soporte y documentación. *Ansible* requiere muy poca configuración inicial, mientras que *Salt*, al soportar varios modos de ejecución requiere de una mayor configuración previa.

2.4 Contenedores

Como ya se ha mencionado anteriormente, el concepto de contenedor nació con *OpenVZ* y ganó tracción con *LXC*, aunque no fue popularizado hasta la creación de *Docker* en 2013. Desde su inicio, *Docker* trajo consigo muchas innovaciones situándose como estándar de facto, sentando precedente en muchos aspectos tecnológicos de los contenedores.

Más tarde, se tomó la decisión de dividir su proyecto en varios para estandarizar y modularizar las tecnologías utilizadas por los contenedores, buscando fomentar la adopción e interoperabilidad de estas tecnologías.

En 2015, *Docker* donó su implementación del *container runtime de bajo nivel*, *runc*, a la *Open Container Initiative* (*OCI*). Desde entonces la *OCI* basó sus especificaciones en esta implementación y publicó sus especificaciones para *runtimes*, imágenes y para la distribución.

En 2017, *Docker* donó su implementación del *container runtime de alto nivel*, *containerd*, a la *Cloud Native Computing Foundation* (*CNCF*). En la actualidad, *containerd* es un proyecto independiente que implementa la especificación *Container Runtime Interface* (*CRI*) necesaria para la integración con *Kubernetes*.

Esta apertura provocó la irrupción de nuevos actores en el ecosistema de los contenedores y permitió conocer la verdadera complejidad de estas plataformas.

Tras el desuso de *dockershim* por parte del proyecto *Kubernetes* en la versión 1.24, se pudo ver la confusión causada en la comunidad. *dockershim* fue una herramienta temporal creada para permitir la integración de *Docker* con *Kubernetes*, puesto que *Docker* era una herramienta que no soportaba la especificación *CRI*.

Anteriormente los usuarios simplemente instalaban *Docker* y obtenían toda la funcionalidad que requerían, desde la creación de las imágenes, ejecución de los contenedores, creación de redes, entre otras. Sin embargo, ahora los usuarios deben escoger un *container runtime* para su plataforma de gestión de contenedores.

La arquitectura de *Docker* en la actualidad es la siguiente:

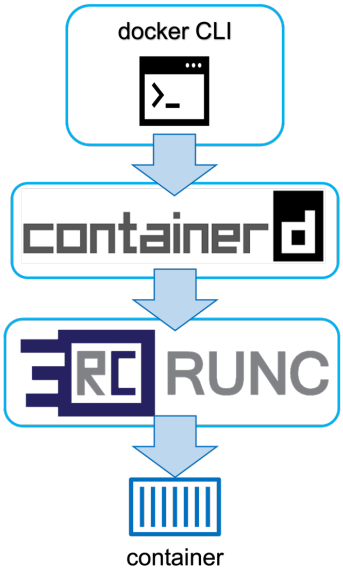

Figura 3: Arquitectura de Docker en la actualidad

Además de *containerd*, hoy en día existen varios proyectos con implementaciones de *container runtimes* de alto nivel como pueden ser *CRI-O* o *Podman*. *CRI-O* es una implementación minimalista del *CRI* de *Kubernetes* y por ello solo puede ser utilizada exclusivamente en entornos de *Kubernetes*. *Podman* surgió como alternativa a *Docker* pero con una arquitectura sin *daemon*, esta queda desestimada por no tener integración con *Kubernetes*.

En el ámbito de *container runtimes de bajo nivel*, además de *runc*, tenemos proyectos como *gVisor*, *Firecracker* o *crun*. En este ámbito, *runc* es el líder indiscutible, siendo la implementación de referencia del estándar *OCI*. Sin embargo, *gVisor* implementa características que mejoran la seguridad de los contenedores, *Firecracker* está enfocado en la ejecución de *microVMs* y *crun* en la ejecución de contenedores en sistemas con recursos limitados.

Debido a la madurez de ambos proyectos, tras haber nacido junto con *Docker* hasta convertirse en proyectos independientes, se han escogido *containerd* como *container runtime de alto nivel* y *runc* como *container runtime de bajo nivel* para este proyecto.

2.5 Plataformas de gestión de contenedores

En 2014, *Google* lanzó *Kubernetes*, un orquestador de aplicaciones basadas en contenedores. Unos meses más tarde, *Docker* presentó *Docker Compose* y *Docker Swarm*.

Docker Compose es una herramienta que facilita el despliegue de aplicaciones multicontenedor en una misma máquina y ha tenido una amplia adopción por

su facilidad de uso. Por otro lado, *Docker Swarm*[3], actualmente un proyecto descontinuado, ofrecía una funcionalidad similar con un enfoque a despliegues en varias máquinas. Las funcionalidades de *Docker Swarm* han sido integradas en el modo *swarm* de *Docker Engine*.

En 2015 *Google* se asoció con la *Linux Foundation* para formar la *Cloud Native Computing Foundation* y donó *Kubernetes* [20] como proyecto semilla.

En la actualidad, *Kubernetes* es considerado estándar de facto para la orquestación de contenedores. Existen muchas distribuciones de *Kubernetes* como *OpenShift* de *Red Hat*, *Rancher Kubernetes Engine* (*RKE*) de *Rancher Labs* o las diferentes distribuciones que se pueden encontrar en los proveedores *cloud* como *Elastic Kubernetes Service* (*EKS*) en *AWS*, *Azure Kubernetes Service* (*AKS*) en *Azure* o *Google Kubernetes Engine* (*GKE*) en *GCP*.

También existen múltiples herramientas de instalación y configuración de clústers como *kubeadm* [21], *k3s*, *kops* o *microk8s*. Cada una de estas herramientas tiene un propósito y unas limitaciones diferentes.

Un despliegue de un clúster de *Kubernetes* requiere al menos de una máquina que actúe como plano de control. Aunque se pueden configurar varios planos de control para obtener redundancia y equilibrado de carga. El plano de control de *Kubernetes* es el encargado de la gestión del estado del clúster, vigilando el estado de todas las máquinas que conforman el clúster y los contenedores desplegados en él. También gestiona la autorización de las peticiones que llegan al clúster, debido a que expone la *API*, interfaz principal de todos los recursos del clúster. Se podría definir el plano de control como el cerebro del clúster, ya que es el encargado de la toma de decisiones.

El clúster también puede contener máquinas destinadas únicamente a la ejecución de contenedores, llamadas nodos. Estas máquinas aportan recursos al clúster para poder desplegar las aplicaciones. Siguiendo la analogía, se podrían definir a los nodos como los músculos del clúster, encargados de la ejecución de las decisiones tomadas por el plano de control.

Un clúster de *Kubernetes* está compuesto por los siguientes componentes:

[3] *Docker Swarm* está descontinuado como el proyecto *classicswarm*. Por otra parte, *Mirantis* ha comprado la sección *Docker Enterprise* y ha relanzado *Swarm* como parte de su *Mirantis Kubernetes Engine* (*MKE*).

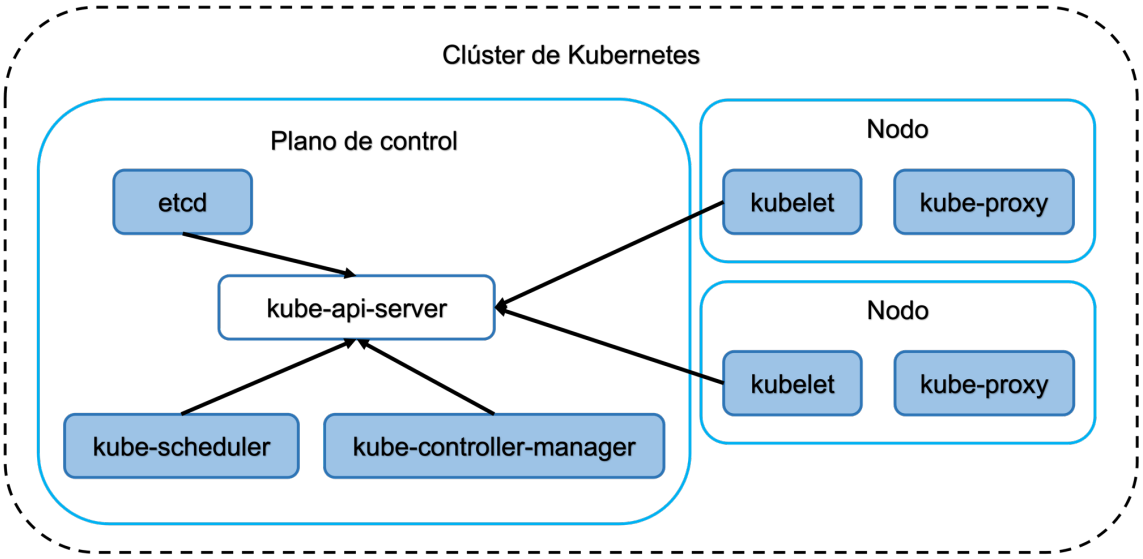

Figura 4: Componentes de Kubernetes

El plano de control contiene:

- ***kube-apiserver***: implementa la *API* de *Kubernetes*
- ***etcd***: base de datos distribuida que contiene la información del clúster
- ***kube-scheduler***: toma las decisiones acerca de la ejecución de los contenedores en los nodos
- ***kube-controller-manager***: encargado de ejecutar los *controllers* que supervisan el estado del clúster

Un nodo contiene:

- ***kubelet***: gestiona la ejecución de los contenedores en los *pods*
- ***kube-proxy***: encargado de mantener las reglas en los nodos que permiten la comunicación entre *pods* y *services* en el clúster

Los clústers también hacen uso de *addons* a través de los cuales ofrecen funcionalidades como *DNS*, *dashboards* o los *plugins de red* necesarios para la interconexión.

Con estos componentes *Kubernetes* es capaz de ejecutar aplicaciones basadas en contenedores con los objetos definidos por su *API* como son los *Pods*, *Services*, *Namespaces*, entre otros. Estos conceptos serán explicados con más detalle en el capítulo 3: Desarrollo.

Para la instalación del clúster utilizaremos la herramienta *kubeadm*, ya que nos permite configurar y adaptar el clúster a nuestras necesidades sin imponer el *plugin de red* a utilizar.

2.6 Redes móviles (5G)

En 2017, el *3GPP* (*3rd Generation Partnership Project*) completó la especificación inicial de lo que sentaría las bases de 5G, conocida como *Release 15*. Entre muchas tecnologías propuestas y decisiones de diseño, una de las decisiones fundamentales fue la inclusión de una arquitectura para el plano de control basada en servicios, conocida como *Service Based Architecture* (*SBA*). El *SBA* define la comunicación entre elementos del plano de control del núcleo de red 5G a través de *APIs REST* basadas en el protocolo *HTTP/2*.

Otro de los conceptos clave es el concepto de *Network Function Virtualization* (*NFV*), acuñado en 2012 por el *European Telecommunications Standards Institute* (*ETSI*), que proporciona un enfoque para la virtualización de elementos de red y una infraestructura para su ejecución, la *Network Function Virtualization Infraestructure*[4] (*NFVI*).

Estas tecnologías sentaron las bases del diseño de las *Network Functions* (*NFs*) en 5G, facilitando la adopción de la virtualización desde el principio. En el contexto de 5G, el concepto de *Network Function* se refiere a los diferentes componentes que conforman el núcleo de red 5G, tanto en el plano de control como en el plano de usuario. Existen muchas *Network Functions* distintas, cada una con un rol específico, algunas de ellas serán explicadas en el capítulo 3: Desarrollo.

Aunque ya en 4G existían, las implementaciones del núcleo de red basadas en *software* eran poco comunes, aun así existían algunos ejemplos de código abierto. Sin embargo, con la adopción de las tecnologías mencionadas anteriormente y el precedente establecido por 4G, 5G ha visto un aumento significativo en estos proyectos. Algunos ejemplos son *Open5GS* [22], *Free5GC* [23], *OpenAirInterface* [24] (*OAI*) o *SD-Core* [25], entre otros.

El cambio de paradigma en las redes móviles de *hardware* a *software* trajo consigo algunas de las metodologías y herramientas utilizadas en el desarrollo de *software*. Algunos ejemplos son *Continuous Integration and Continuous Delivery* (*CI/CD*), el *testing* con herramientas específicas o técnicas genéricas como el *fuzzing* o las arquitecturas basadas en microservicios.

En el ámbito de 5G, el *testing* ha evolucionado para incluir herramientas específicas que permiten la simulación de componentes de la red como son las estaciones base (*gNBs*) o los dispositivos móviles (*UEs*). Estas herramientas permiten realizar tests para comprobar el funcionamiento de la red, garantizar la interoperabilidad o medir su rendimiento. Algunos ejemplos son herramientas

[4] Cabe señalar que aunque para el TFM se va a desplegar un núcleo de red 5G en un entorno virtualizado, la infraestructura no puede ser considerada como *NFVI*. La *ETSI* define una arquitectura específica para esta infraestructura y una capa de orquestación específica llamada *MANO* (*Management and Orchestration*), no utilizadas en este TFM.

comerciales como *Landslide* [26] de *Spirent* o *LoadCore* [27] de *Keysight* o proyectos de código abierto como *UERANSIM* [28] o *PacketRusher* [29] de *HP*.

Para la realización de este TFM se ha escogido *Open5GS* como implementación del núcleo de red 5G por su amplia adopción, el número de funcionalidades implementadas y por la experiencia adquirida en mi entorno de trabajo. Para este proyecto, se utilizará *docker-open5gs* [30], una versión de *Open5GS* basada en contenedores que he desarrollado en el *iTEAM*, fuera del contexto de este TFM.

Para la validación del núcleo de red 5G desplegado, se utilizará *UERANSIM* entre otras herramientas. *UERANSIM* ha sido escogido por ser una alternativa de código abierto, la simplicidad de su instalación y que no requiere de *módulos del kernel* instalados como en el caso de *PacketRusher*.

3. Desarrollo

El proyecto comenzó con la puesta en marcha de la infraestructura *on-premises*. Esta decisión se tomó puesto que realizar las primeras pruebas en una red local resulta más sencillo y económico, lo que permite utilizar esta infraestructura para familiarizarse con el entorno antes de trasladar los avances a la infraestructura de *cloud* público.

Una de las ventajas de utilizar la infraestructura *on-premises* para comenzar es el control de los elementos de interconexión de la red. Esto elimina la presencia de *firewalls* u otros elementos que puedan alterar o bloquear las conexiones entre los dispositivos, facilitando un entorno más predecible.

El despliegue *on-premises* está formado por tres *Raspberry Pi 4 Model B* y una *Raspberry Pi 3 Model B*, todas conectadas a un *unmanaged switch NETGEAR GS105E-200PES* de cinco puertos con capacidad de 1 Gigabit. Uno de los puertos del *switch* está conectado a un *router NAT* (*Network Address Translation*) con conexión a Internet.

El sistema operativo seleccionado para los equipos de la infraestructura *on-premises* es *Raspberry Pi OS*, un sistema operativo *GNU/Linux* basado en la distribución *Debian*.

Tras la instalación de *Raspberry Pi OS* en los equipos y la comprobación del correcto funcionamiento de la red local, se realizó una instalación manual de *Kubernetes*, utilizando la herramienta *kubeadm*, siguiendo la guía de instalación [31].

En esta primera instalación se identifican los prerrequisitos de la instalación de *Kubernetes* para así adecuar los equipos. *Kubernetes* tiene los siguientes prerrequisitos a nivel *hardware*:

- Al menos 2 CPUs
- Al menos 2 GB de memoria RAM

A nivel *software*:

- Deshabilitar la *memoria swap*
- Habilitar los *módulos del kernel cgroups*, *br_netfiter* y *overlay*
- Habilitar el enrutamiento de *IPv4*/*IPv6*
- Permitir a *iptables* ver el tráfico dentro de los *bridges*

Para la instalación del clúster de *Kubernetes* utilizaremos las tres *Raspberry Pi 4 Model B*, sus características *hardware* son las siguientes:

- *CPU* de 64 bits *Broadcom BCM2711*, con cuatro núcleos *ARM v8 Cortex-A72* a 1,8 GHz
- 4 GB de memoria *RAM LPDDR4-3200 SDRAM*
- Interfaz de red Gigabit *Ethernet*
- *Micro-SD* con 32 GB de almacenamiento

3.1 Instalación de Kubernetes en Raspberry Pi

En el caso específico de *Raspberry Pi*, los *cgroups* han de ser habilitados en el *kernel* modificando la configuración del archivo *cmdline.txt* [32]. Este archivo se encuentra en */boot/firmware/cmdline.txt* y contiene los parámetros que el *kernel* utilizará en el arranque del sistema.

Añadiendo *cgroup_enable=cpuset cgroup_memory=1 cgroup_enable=memory* al final del contenido de este archivo en la misma línea, habilitaremos los *cgroups*.

```
console=serial0,115200 console=tty1 root=PARTUUID=<uuid>-02
rootfstype=ext4 elevator=deadline fsck.repair=yes rootwait quiet
splash plymouth.ignore-serial-consoles cgroup_enable=cpuset
cgroup_memory=1 cgroup_enable=memory
```

Fragmento 1: Archivo cmdline.txt tras los cambios

Los *cgroups* o *control groups* son un mecanismo integrado en el *kernel* de *Linux* que permite el control de recursos como *CPU*, memoria, recursos de red o la entrada/salida de disco de un grupo de procesos. Sus funcionalidades incluyen la limitación del uso de los recursos, priorización o la monitorización de recursos.

Por otra parte, los *módulos del kernel overlay* y *br_netfilter* pueden ser añadidos en tiempo de ejecución a diferencia de los *cgroups*. El módulo *overlay* permite el uso del *overlayFS*. El *filesystem overlay* es una implementación de *union filesystem* utilizado por las imágenes de los contenedores. Mediante este *filesystem* los contenedores son capaces de almacenar sus imágenes por capas que pueden ser reutilizadas por los distintos contenedores presentes en el sistema, optimizando el espacio utilizado por las imágenes y agilizando el proceso de construcción de las imágenes al reutilizar capas. *overlayFS* implementa el llamado *copy-on-write* (*CoW*), que permite que los ficheros sean compartidos entre capas de solo lectura y cuando un archivo de una capa es modificado, este es copiado de la capa de solo lectura a una capa que permite la escritura para su modificación.

El módulo *br_netfilter* permite la gestión del tráfico en los *bridges* de *Linux* mediante *iptables*. Los *bridges* de *Linux* son dispositivos virtuales de red que emulan el funcionamiento de un *switch* de capa 2 o capa 3 (dependiendo de la

configuración). Los *bridges* son utilizados para interconectar máquinas virtuales o contenedores en la misma red virtual.

iptables es un software que permite la gestión de reglas para el filtrado de los paquetes *IP* que atraviesan el *kernel*. *iptables* define los conceptos de tabla, cadena y regla para la gestión de los paquetes. Existen distintas tablas en *iptables* y estas tablas tienen asociadas cadenas y cada cadena a su vez tiene asignadas un conjunto de reglas. El conjunto de tabla y cadena identifica un punto específico en el procesamiento del paquete, como puede ser la modificación de un paquete antes de ser enrutado en la tabla de *NAT* para aplicar la regla de redirección de un puerto a otro diferente.

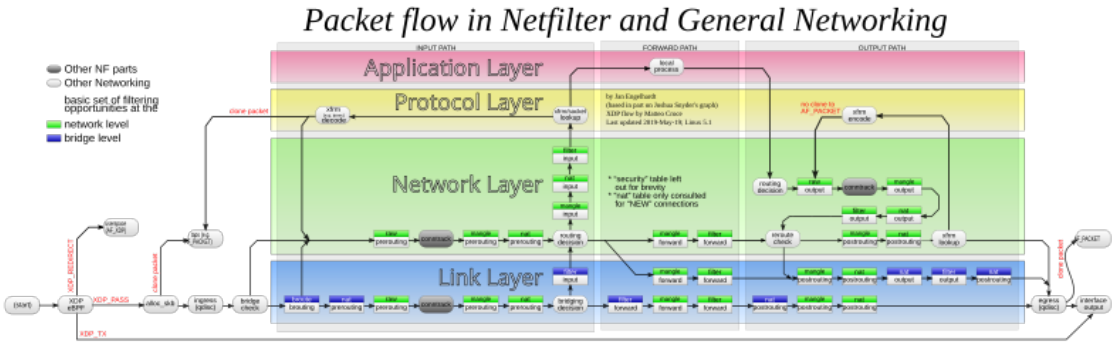

Figura 5: Arquitectura de iptables [33]

```
iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 8080
```

Fragmento 2: Ejemplo de regla en iptables

Con *-t* indicamos la tabla, en este caso *nat* como hemos indicado. Con *-A* añadimos la siguiente regla en la cadena *PREROUTING* (antes del enrutado) y la regla consiste en *REDIRECT* (redirigir) el tráfico *TCP* con puerto destino 80 al puerto 8080.

Las reglas que podemos aplicar son aceptar, rechazar, redirigir, enmascarar, entre otras.

br_netfilter permite añadir este tipo de reglas de *iptables* en el tráfico cursado por los *bridges*, utilizados por *Kubernetes* para el envío de tráfico entre los *pods* o contenedores.

Para que el sistema operativo cargue los *módulos del kernel* tras cada arranque, añadiremos estos módulos al archivo *modules*, que se encuentra en */etc/modules*.

La configuración para permitir el enrutamiento de *IPv4*/*IPv6* en el equipo y para permitir a *iptables* ver el tráfico cursado en los *bridges* se realiza modificando los parámetros del *kernel* con *sysctl*. *sysctl* permite cambiar la configuración de algunos parámetros del *kernel* en tiempo de ejecución, utilizando la

herramienta *sysctl* o modificando los valores del *filesystem* virtual ubicado en */proc/sys*.

sysctl también contiene un archivo, el *sysctl.conf* ubicado en */etc/sysctl.conf* el cual puede ser utilizado para añadir los cambios de manera persistente. A continuación, se mostrará cómo se habilita el enrutamiento de *IPv4* en la máquina de las tres maneras mencionadas:

- Con la herramienta *sysctl* en tiempo de ejecución: **sysctl -w net.ipv4.ip_forward=1**
- Modificando el fichero en el *filesystem* virtual: **echo 1 > /proc/sys/net/ipv4/ip_forward**
- Añadiendo la siguiente línea en el archivo */etc/sysctl.conf*: **net.ipv4.ip_forward = 1**

Las otras dos opciones a modificar con *sysctl* son **net.bridge.bridge-nf-call-iptables** y **net.bridge.bridge-nf-call-ip6tables**. Poniendo las dos opciones a **1**, es decir habilitándolas, permitiremos a *iptables* ver el tráfico dentro de los *bridges* tanto para *IPv4* como para *IPv6*.

La *memoria swap* ha de ser deshabilitada en dos pasos, por un lado, deshabilitar su uso y por otro lado eliminar su contenido y su mapeado en el sistema operativo. La *memoria swap* permite la ampliación de la memoria *RAM* instalada en el sistema. Para ello almacena contenido menos utilizado de la memoria *RAM* en una partición especial creada por el sistema operativo llamada *swap*. Esta técnica permite al sistema utilizar más memoria *RAM* de la que el sistema dispone en situaciones de sobrecarga a cambio de una penalización en la velocidad de acceso de estos datos, puesto que la velocidad de acceso a *RAM* es superior a la velocidad de acceso a la partición *swap* del disco.

Utilizando el comando **swapoff -a** podemos deshabilitar la memoria *swap* para la sesión actual. Si queremos deshabilitar la memoria *swap* de manera persistente tenemos que eliminar la entrada en el archivo *fstab*, ubicado en */etc/fstab*, este fichero contiene las particiones y los *filesystems* presentes en el equipo.

Comentar o eliminar la siguiente línea del fichero */etc/fstab*:

```
# UUID=XXXXXXX     none     swap     sw     0     0
```

Fragmento 3: Entrada de la partición swap en el fichero fstab

Tras la configuración de los equipos para cumplir los prerrequisitos de *Kubernetes* y tras realizar comprobaciones de la red, continuamos con la instalación de un *container runtime*.

Kubernetes es un orquestador de aplicaciones basadas en contenedores, para realizar su función necesita la instalación de un *container runtime* en cada uno de los nodos que forman el clúster, para así poder ejecutar los contenedores. Existen varios *container runtimes* disponibles, para ser utilizado por *Kubernetes* se necesita que cumpla con la especificación *CRI*. Para esta instalación se ha escogido *containerd* como *container runtime* debido a su madurez como proyecto.

Es necesaria una aclaración acerca de los *container runtimes* tras lo mencionado previamente en el capítulo 2: Estado del Arte y Tecnologías Relacionadas acerca de *Docker*. El *container runtime* que se va a instalar para *Kubernetes* es *containerd*, *containerd* realiza tareas como la gestión del ciclo de vida del contenedor o la transferencia y el almacenamiento de las imágenes o la ejecución y supervisión de los contenedores. Se omite por completo la existencia del *container runtime de bajo nivel*, en este caso el utilizado por *containerd* es *runc*. Un *container runtime de bajo nivel* se encarga de proporcionar una interfaz agnóstica del sistema operativo para la creación y ejecución de contenedores.

La instalación de *containerd* se realiza a través de los repositorios *apt* de *Docker*, donde podemos encontrar una versión más actualizada. El nombre del paquete de *apt* es *containerd.io*. Tras la instalación se debe configurar *systemd* como *cgroup driver* [34]. Configuraremos *systemd* como *cgroup driver* en *kubelet* para tener un único punto de gestión de *cgroups* en el sistema, puesto que *Raspberry Pi OS* utiliza systemd como *init system*. Para ello tenemos que editar el archivo *config.toml* ubicado en */etc/containerd/config.toml* que contiene la configuración de *containerd* y añadir el siguiente parámetro:

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
    SystemdCgroup = true
```

Fragmento 4: Archivo config.toml tras los cambios

Tras la instalación y configuración de *containerd* como *container runtime*, el cúal utiliza *runc* como *container runtime de bajo nivel* (*OCI runtime*), podemos pasar a la instalación de *Kubernetes* utilizando *kubeadm*.

La instalación de *kubeadm* se realiza a través de los repositorios *apt* de *Kubernetes*, los nombres de los paquetes a instalar son *kubeadm*, *kubelet* y *kubectl*. *kubeadm* será el encargado de crear el clúster de *Kubernetes*, *kubelet* es el componente de *Kubernetes* encargado de gestionar los contenedores y *kubectl* es la herramienta utilizada por el usuario para interactuar con el clúster.

Para crear nuestro clúster de *Kubernetes*, ejecutaremos el comando **kubeadm init** en el nodo designado como plano de control del clúster. En nuestro caso

particular, indicaremos la dirección *IPv4* de la subred que vamos a utilizar para la interconexión de *pods*. El comando final quedará así:

```
kubeadm init --pod-network-cidr=10.13.0.0/24
```

Fragmento 5: Comando kubeadm en el plano de control

Tras haber iniciado el plano de control y copiar el contenido del fichero *admin.conf* generado por *kubeadm* como sugiere la salida del comando, el siguiente paso es añadir los demás nodos al clúster creado utilizando el comando **kubeadm join**:

```
kubeadm join <control-plane-host>:<control-plane-port> --token <token>
--discovery-token-ca-cert-hash sha256:<hash>
```

Fragmento 6: Comando kubeadm en los nodos

Tras la ejecución de estos comandos podemos comprobar la interconexión de los nodos del clúster ejecutando el comando **kubectl get nodes** y comprobando que la salida de este comando muestra todos los nodos añadidos en estado *Ready*.

Aunque si revisamos los elementos desplegados en el clúster con **kubectl get all -A** podremos comprobar que los elementos relacionados con el sistema *DNS* de *Kubernetes* (*CoreDNS*) no pueden llegar a desplegarse, esto es normal, puesto que para el correcto funcionamiento de nuestro clúster necesitamos un *network add-on* o *plugin de red*.

El siguiente paso para tener un clúster completamente funcional es instalar un *network add-on* para proporcionar conectividad a nuestros *pods*. *Kubernetes* no exige la instalación de un *pod network add-on* específico, simplemente el *network add-on* tiene que cumplir la especificación *Container Network Interface* (*CNI*) para poder ser utilizado junto a Kubernetes. Existen muchos *pod network add-ons* compatibles con esta especificación, para esta instalación se ha seleccionado *Calico* [35] por su amplia variedad de protocolos soportados [36].

La instalación de *Calico* es muy sencilla, sólo se ha de deshabilitar el *firewall*, instalar el *Tigera Operator* de la versión de *Calico* seleccionada y aplicar los cambios en la configuración pertinentes. En nuestro caso, los cambios en la configuración consisten en cambiar la red por defecto de *Calico* para la interconexión de los *pods*.

Vamos a instalar *Calico* utilizando el *operador* de *Kubernetes* llamado *Tigera Operator*. Un *operador* en *Kubernetes* se refiere a un controlador que amplía las funcionalidades de la *API* de *Kubernetes* y permite la creación y gestión de recursos personalizados [37].

Con el siguiente comando instalaremos el *Tigera Operator* versión v3.27.3:

```
kubectl create -f
https://raw.githubusercontent.com/projectcalico/calico/v3.27.3/manifest
s/tigera-operator.yaml
```

Fragmento 7: Instalación Tigera Operator v3.27.3

Tras esto, descargaremos los *custom resources* asociados a la versión:

```
wget
https://raw.githubusercontent.com/projectcalico/calico/v3.27.3/manifest
s/custom-resources.yaml
```

Fragmento 8: Descarga custom resources Calico v3.27.3

Modificaremos el fichero para añadir la subred utilizada para la interconexión de *pods* especificada en el comando *kubeadm init*. Tras esto, podemos crear la subred haciendo uso de los *custom resources* y el *Tigera Operator* previamente instalado.

```
kubectl create -f custom-resources.yaml
```

Fragmento 9: Instalación custom resources Calico v3.27.3

Tras esperar unos segundos, podemos comprobar que los recursos empiezan a ser desplegados en el clúster. Se podrá ver cómo *Calico* crea un *pod "calico-node"* por cada uno de los nodos del clúster[5].

```
kubectl taint nodes --all node-role.kubernetes.io/control-plane-
```

Fragmento 10: Comando para elimintar el taint del plano de control

Tras la eliminación del *taint*, se podrá ver como todos los *pods* de *Calico* son creados y tras esto se podrá ver la correcta configuración de los *pods* de *DNS* del clúster. Con ello finalizará la instalación básica del clúster de *Kubernetes*.

```
borjis@k8s-master:~ $ kubectl -n kube-system get all
NAME                                      READY   STATUS    RESTARTS      AGE
pod/coredns-7db6d8ff4d-8vtrr              1/1     Running   2 (17h ago)   20d
pod/coredns-7db6d8ff4d-9ksfg              1/1     Running   2 (17h ago)   20d
pod/etcd-k8s-master                       1/1     Running   2 (17h ago)   20d
pod/kube-apiserver-k8s-master             1/1     Running   2 (17h ago)   20d
pod/kube-controller-manager-k8s-master    1/1     Running   3 (17h ago)   20d
pod/kube-proxy-59ccv                      1/1     Running   2 (17h ago)   20d
pod/kube-proxy-gh6nx                      1/1     Running   2 (17h ago)   20d
pod/kube-proxy-h4sxp                      1/1     Running   2 (17h ago)   20d
pod/kube-scheduler-k8s-master             1/1     Running   2 (17h ago)   20d

NAME                                  TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
service/kube-dns                      ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   20d
```

[5] Para habilitar el despliegue de contenedores en el plano de control tenemos que eliminar el taint que añade kubeadm a Kubernetes en su instalación [49].

```
service/prometheus-coredns                    ClusterIP   None         <none>        9153/TCP                      4m
service/prometheus-kube-controller-manager    ClusterIP   None         <none>        10257/TCP                     4m
service/prometheus-kube-etcd                  ClusterIP   None         <none>        2381/TCP                      4m
service/prometheus-kube-proxy                 ClusterIP   None         <none>        10249/TCP                     4m
service/prometheus-kube-scheduler             ClusterIP   None         <none>        10259/TCP                     4m
service/prometheus-kubelet                    ClusterIP   None         <none>        10250/TCP,10255/TCP,4194/TCP  40m

NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/kube-proxy     3         3         3       3            3           kubernetes.io/os=linux   20d

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/coredns     2/2     2            2           20d

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/coredns-7db6d8ff4d      2         2         2       20d
```

Fragmento 11: Recursos de Kubernetes desplegados en el clúster

```
admin@ip-172-31-18-207:~$ kubectl -n calico-system get all
NAME                                       READY   STATUS    RESTARTS   AGE
pod/calico-kube-controllers-5945f69c66-cbz8f   1/1     Running   0          7m34s
pod/calico-node-759hw                      1/1     Running   0          4m52s
pod/calico-node-df8gq                      1/1     Running   0          4m21s
pod/calico-node-pg5hh                      1/1     Running   0          4m14s
pod/calico-typha-55cb8cb654-hmfp5          1/1     Running   0          7m34s
pod/calico-typha-55cb8cb654-qqrsl          1/1     Running   0          7m27s
pod/csi-node-driver-bg8vf                  2/2     Running   0          7m34s
pod/csi-node-driver-v75q7                  2/2     Running   0          7m34s
pod/csi-node-driver-zddx7                  2/2     Running   0          7m34s

NAME                                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/calico-kube-controllers-metrics   ClusterIP   None            <none>        9094/TCP   6m50s
service/calico-typha                    ClusterIP   10.104.94.107   <none>        5473/TCP   7m35s

NAME                              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/calico-node        3         3         3       3            3           kubernetes.io/os=linux   7m34s
daemonset.apps/csi-node-driver    3         3         3       3            3           kubernetes.io/os=linux   7m34s

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/calico-kube-controllers   1/1     1            1           7m34s
deployment.apps/calico-typha           2/2     2            2           7m34s

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/calico-kube-controllers-5945f69c66   1         1         1       7m34s
replicaset.apps/calico-typha-55cb8cb654           2         2         2       7m34s
```

Fragmento 12: Recursos de Calico desplegados en el cluster

3.2 Aplicación a desplegar en Kubernetes

El objetivo final del clúster de *Kubernetes* es albergar un núcleo de red 5G basado en contenedores. Para ello se va a utilizar el proyecto *Open5GS*, una implementación de código abierto del núcleo de red 4G y 5G, realizada en *C*.

Open5GS proporciona el código y las instrucciones para compilar los distintos componentes que conforman el núcleo de red. Estos componentes son transformados en ejecutables y el proyecto facilita una manera de desplegar estos componentes utilizando servicios de *systemd*. Los cambios en la configuración de estos componentes se llevan a cabo mediante la edición de unos ficheros de configuración en formato *YAML*.

Buscando una manera más sencilla de desplegar estos componentes, en el contexto del grupo de comunicaciones móviles del *iTEAM*, desarrollé imágenes de contenedores para ser utilizadas en proyectos de *Docker Compose* para probar nuevas funcionalidades. Este proyecto está disponible en *GitHub docker-open5gs* [30].

La primera modificación del proyecto para el TFM fue incorporar la arquitectura *ARM* para la creación de las imágenes, puesto que este proyecto se va a utilizar en *Raspberry Pi*. Esta modificación se llevó a cabo configurando un nuevo *builder* (ver anexo 7.1) utilizando *docker buildx*, un *plugin* para *Docker* que incorpora funcionalidades de *BuildKit.* Este builder se utilizó para crear las imágenes para la arquitectura nativa *x86-64* (también llamada *amd64*) y para *arm64* utilizando contenedores basados en el emulador *QEMU* con herramientas para realizar el *cross-compilation*.

También se aprovechó para migrar la herramienta de creación de contenedores de un *script* basado en *Make* con comandos de *Docker* a utilizar *Bake*. *Bake* es una funcionalidad de *docker buildx* que permite definir los parámetros para la creación de las imágenes de los contenedores utilizando un fichero, el *Bakefile*. *Bake* permite una manera sencilla de definir escenarios para la creación de imágenes y reducir los tiempos de creación de imágenes utilizando paralelismo (ver anexo 7.2).

Para poder utilizar la base de datos de *Open5GS* basada en *MongoDB* [38] versión 6 en las *Raspberry Pi* se ha utilizado una imagen para contenedores no oficial [39]. *MongoDB* no soporta *ARMv8.0* para sus versiones 5 o superior, estas solo están disponibles para *ARMv8.2* y las *Raspberry Pi 4 Model B* tienen un procesador basado en *ARMv8.0*.

Otra modificación del proyecto *docker-open5gs* fue la incorporación de la herramienta *WebUI* proporcionada por *Open5GS* para añadir usuarios en la base de datos del núcleo de red 5G de una manera cómoda y sencilla.

Previamente se hacía uso de un script proporcionado por *Open5GS* y en el proyecto *docker-open5gs* se exponía el puerto de la base de datos para añadir usuarios utilizando el *script* (ver anexo 7.3).

3.3 Objetos de Kubernetes

Para nuestra aplicación, desplegaremos los componentes del núcleo de red 5G del proyecto *docker-open5gs* utilizando los objetos proporcionados por la *API* de *Kubernetes*. *Kubernetes* define muchos objetos; en esta sección se introducirán algunos de los objetos básicos, junto con todos los que serán utilizados para el despliegue de nuestra aplicación.

El concepto más básico de *Kubernetes* es el *Pod*. Un *Pod* es un objeto de *Kubernetes* que encapsula uno o varios contenedores siendo ejecutados en un contexto compartido. *Kubernetes* gestiona *Pods*, no contenedores. Los *Pods* permiten la ejecución de varios contenedores que están estrechamente relacionados y precisan de una ejecución en la que comparten recursos como almacenamiento o red. Para casos de uso específicos puede ser útil la creación de *Pods* directamente, aunque por lo general se usan otros métodos de creación indirecta como la creación de *Controllers*.

Los *Controllers* son una capa de abstracción más que permiten definir el estado deseado para nuestros *Pods*. A través de los *Controllers*, el plano de control de *Kubernetes* es el encargado de gestionar que ese estado se cumpla. Algunos ejemplos de *Controllers* son *ReplicaSet* [40], *StatefulSet*, *DaemonSet* o los *Deployments*.

El objeto más común dentro de los *Controllers* es el *Deployment*. Mediante los *Deployments* podemos gestionar el estado de nuestros *Pods*, definiendo el número de réplicas que queremos utilizar. Los *Deployments* se pueden actualizar realizando operaciones de escalado o volviendo a una versión del despliegue anterior. A bajo nivel, un *Deployment* gestiona un objeto *ReplicaSet* que es el encargado de mantener el estado del número de réplicas seleccionadas. Como en el caso de los *Pods*, por lo general no se hace uso de un *ReplicaSet* pero pueden existir casos específicos en los que crear un objeto *ReplicaSet* directamente.

Los *StatefulSets* permiten la gestión de aplicaciones que requieren de algún tipo de estado. En un *Deployment*, los *Pods* creados son intercambiables, en el caso de un *StatefulSet* cada *Pod* tiene una identidad única. Un *StatefulSet* es similar a un *Deployment* que garantiza la persistencia de los datos durante la gestión de los *Pods*. Los identificadores únicos son utilizados para mantener una relación de los *Pods* con los datos que han de ser persistentes, como volúmenes o identificadores de red.

Los *DaemonSets* se ejecutan en un nodo específico dentro del clúster y suelen ser utilizados para la ejecución de funcionalidades extra en el clúster. Funcionalidades como proporcionar servicios de interconexión de red, almacenamiento o monitorización, entre otras.

Para permitir la interconexión entre los *Pods* existen los *Services*. Mediante los *Services* permitimos comunicarnos con otros *Pods* dentro del clúster o también podemos comunicarnos con elementos externos al clúster, existen varios tipos de objetos de tipo *Services*.

Los más comunes son *ClusterIP*, *NodePort*, *Ingress* o *LoadBalancer*. Cuando creamos un objeto *Service* sin especificar su tipo, se crea el tipo por defecto que es *ClusterIP*. *ClusterIP* asigna una *IP* del rango especificado para el clúster y permite la conexión entre los diferentes *Pods* del clúster.

El *NodePort* es un tipo de *Service* que permite el acceso a un *Pod* del clúster utilizando la *IP* de alguno de los nodos del clúster y un puerto específico. Por defecto, *Kubernetes* utiliza los puertos del 30000 al 32767 para este tipo de *Service*. Los nodos pertenecientes al clúster mapean ese puerto al *Pod* en cuestión.

Un *Ingress* permite mapear peticiones *HTTP* o *HTTPS* a una *URL* específica a *Pods* desplegados en un clúster. El servicio *Ingress* utiliza servicios *NodePort* o *LoadBalancer*, así como un *Ingress Controller* para llevar a cabo sus funciones. Cuando accedemos a una de las *URLs* definidas por un *Ingress* estaremos accediendo al servicio expuesto por uno de los *Pods* del clúster.

Un *LoadBalancer* es un objeto que crea un balanceador de carga, el cuál permite exponer una dirección *IP* y un puerto específico para que elementos externos al clúster puedan acceder a nuestros *Pods* a través del balanceador de carga. Es un objeto comúnmente utilizado en entornos *cloud*, ya que requiere de una implementación de *LoadBalancer* para utilizarlo, aunque se pueden instalar en entornos *on-premises* también a través de *Controllers*.

Kubernetes define una serie de objetos para el almacenamiento. De los múltiples objetos relacionados con el almacenamiento se van a comentar los conceptos de *volume*, *PersistentVolume* y *PersistentVolumeClaim*, *ConfigMap* [41], *Secret* y *StorageClass*.

Un *volume* en *Kubernetes* es un recurso de almacenamiento para un *Pod*, los diferentes tipos de *volume* definen cómo se acceden a esos datos y la tecnología utilizada para el almacenamiento. Un *Pod* puede hacer uso de múltiples *volumes* a la vez. Existen los *EphemerealVolumes* que son eliminados cuando el *Pod* es destruido o los *PersistentVolumes* que son

independientes a los *Pods* y mantienen su información tras la destrucción del *Pod*.

El aprovisionamiento de un *PersistentVolume*, puede ser realizado de manera estática por un administrador del clúster o puede ser realizado dinámicamente a través de los *StorageClasses*. Los *Pods* realizan peticiones de almacenamiento con parámetros como el tipo de acceso (lectura, escritura, …) o el tamaño de almacenamiento haciendo uso de los *PersistentVolumeClaims*. Estos *PersistentVolumeClaims* pueden ser satisfechos mediante un volumen pre aprovisionado por un administrador creando objetos de tipo *PersistentVolume* o dinámicamente mediante el uso de *StorageClasses*.

Los *StorageClasses* permiten definir varias clases de almacenamiento en el clúster y que este almacenamiento sea creado dinámicamente. Los *StorageClasses* utilizan *Controllers* para la creación dinámica de los *PersistentVolumes* solicitados.

Mediante el uso de *ConfigMaps* y *Secrets* podemos inyectar datos en los *Pods*. Los *ConfigMaps* son utilizados para información no confidencial como archivos de configuración y los *Secrets* para información confidencial como contraseñas.

Tras introducir la teoría acerca de los objetos de *Kubernetes* se planificará el despliegue del núcleo de red 5G basado en contenedores. El proyecto *docker-open5gs* proporciona ficheros para el despliegue del núcleo de red basado en *Docker Compose* por lo que hay que transformar esta configuración para poder utilizarla en *Kubernetes*.

Cada componente del núcleo de red será desplegado en *Kubernetes* como un *Deployment*, el cuál gestionará la creación de un *ReplicaSet* que gestionará el número de réplicas seleccionadas de los *Pods* a crear. Los archivos de configuración de cada componente serán desplegados como *ConfigMaps*. Se crearán diferentes *Services* para proporcionar la interconexión entre los distintos componentes y la exposición de algunos servicios al exterior del clúster. Componentes como la base de datos que requieren de almacenamiento persistente contarán con un *PersistentVolumeClaim* y un *PersistentVolume* asociados.

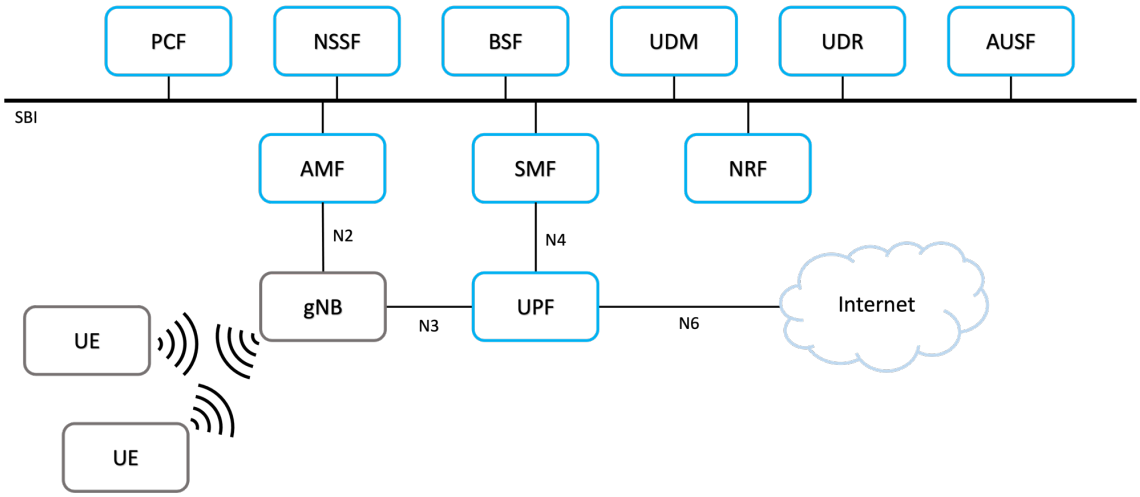

Figura 6: Arquitectura del núcleo de red 5G con un gNB y un UE

De cara a conexiones externas con el despliegue del núcleo de red 5G, sólo importan dos puntos de conexión, el *interfaz N2* del componente *AMF* que permite la conexión del plano de control de los *gNBs* utilizando el protocolo *NGAP* (*SCTP* puerto 38413) y el *interfaz N3* del componente *UPF* que permite la conexión del plano de usuario de los *gNBs* utilizando el protocolo *GTPU* (*UDP* puerto 2152).

Debido a que estos protocolos tienen que usar los puertos definidos se requiere de un tipo de servicio que pueda exponer estos puertos, por lo que se utilizará un balanceador de carga. Los *Services* para los protocolos *NGAP* y *GTPU* de *AMF* y *UPF* serán de tipo *LoadBalancer*.

Kubernetes permite la creación de *Services* de tipo *LoadBalancer* en proveedores de *cloud* públicos, en el caso particular de un despliegue *on-premises*, se ha de instalar un *software* que permita la instanciación de este tipo de servicios. Un ejemplo de este tipo de *software* es *MetalLB* que permite la creación de balanceadores de carga en clústers *bare-metal*.

La instalación de *MetalLB* es muy sencilla, solo consta de una pequeña modificación de la configuración de *kube-proxy* y aplicar el *manifest* de la versión seleccionada en el clúster. Tras esto hay que seleccionar la configuración a aplicar, puesto que *MetalLB* puede funcionar en varios modos diferentes y tras aplicar la configuración solo queda definir el primer *pool de direcciones IP* que el balanceador de carga asignará.

La configuración específica de *kube-proxy* es la siguiente:

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: "ipvs"
ipvs:
```

```
  strictARP: true
```

Fragmento 13: Configuración kube-proxy para MetalLB

En esta configuración se habilita el uso de *strict ARP* en caso de utilizar el modo *IPVS* de *kube-proxy*.

La instalación de *MetalLB* con el *manifest* de la versión v0.14.8:

```
kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.14.8/config/manife
sts/metallb-native.yaml
```

Fragmento 14: Instalación manifest MetalLB v0.14.8

El clúster *on-premises* funcionará con la configuración de *MetalLB* en capa dos del *modelo OSI*, puesto que la configuración de *Calico* utiliza el protocolo *BGP* que se utiliza para la configuración capa tres del *modelo OSI* de *MetalLB* y requiere de varios ajustes para hacerlo funcionar.

En la configuración de capa dos, *MetalLB* responde a las peticiones *ARP* para las *IPs* seleccionadas del balanceador de carga. La configuración requiere de la creación de dos recursos en el *namespace metallb-system*, la configuración para el anuncio de las direcciones *IP* en modo capa dos o *L2Advertisement* y la configuración del pool de *IPs* a utilizar o *IPAddressPool*:

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: example
  namespace: metallb-system
```

Fragmento 15: Ejemplo objeto L2Advertisement de MetalLB

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb-system
spec:
  addresses:
  - 192.168.0.140-192.168.0.150
```

Fragmento 16: Ejemplo objeto IPAddressPool de MetalLB

Las direcciones *IPs* configuradas por *MetalLB* serán alcanzables desde fuera del clúster por lo que utilizando la misma subred que los equipos que conforman el clúster nos permitirá alcanzar estos servicios desde la subred, sin necesidad de rutas de red adicionales.

Con esta configuración podemos crear servicios de tipo *LoadBalancer* en el clúster.

```
admin@ip-172-31-18-207:~$ kubectl -n metallb-system get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/controller-6dd967fdc7-sr8br     1/1     Running   0          8m23s
pod/speaker-sqwbn                   1/1     Running   0          8m23s
pod/speaker-svm4n                   1/1     Running   0          8m23s
pod/speaker-xdgcg                   1/1     Running   0          8m23s

NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/metallb-webhook-service     ClusterIP   10.110.74.249   <none>        443/TCP   8m23s

NAME                        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/speaker      3         3         3       3            3           kubernetes.io/os=linux   8m23s

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/controller      1/1     1            1           8m23s

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/controller-6dd967fdc7      1         1         1       8m23s
```

Fragmento 17: Recursos de MetalLB desplegados en el cluster

Antes de detallar la creación de los objetos de *Kubernetes* necesarios para el despliegue del núcleo de red 5G, se introducirán brevemente los distintos componentes desplegados y sus funciones.

3.4 Componentes del núcleo de red 5G

Existen múltiples configuraciones de un núcleo de red 5G en función de las funcionalidades que soporta. Cada *Network Function* desplegada aporta funcionalidades diferentes al núcleo de red, en el caso particular del núcleo de red 5G desplegado para este TFM se hará uso del mínimo número de componentes necesarios de los proporcionados por el proyecto *Open5GS*.

Se van a definir previamente los conceptos de *Network Slicing*, *PDU Session*, *handover* y *roaming*, conceptos cruciales para el entendimiento de la funcionalidad de las *Network Functions* que se explicarán más adelante.

Network Slicing es una de las novedades aportadas por 5G. *Network Slicing* permite la segmentación del núcleo de red 5G de forma que dentro de una red física puedan coexistir varias redes lógicas, denominados *slices*. Esto no es solo una forma de compartir recursos, sino que también permite dividir las redes según sus características y su funcionalidad.

En 5G se dividieron los casos de uso en tres grandes categorías: *Enhanced Mobile Broadband* (*eMBB*), *Massive Machine-Type Communications* (*mMTC*) y *Ultra-Reliable Low-Latency Communications* (*URLLC*).

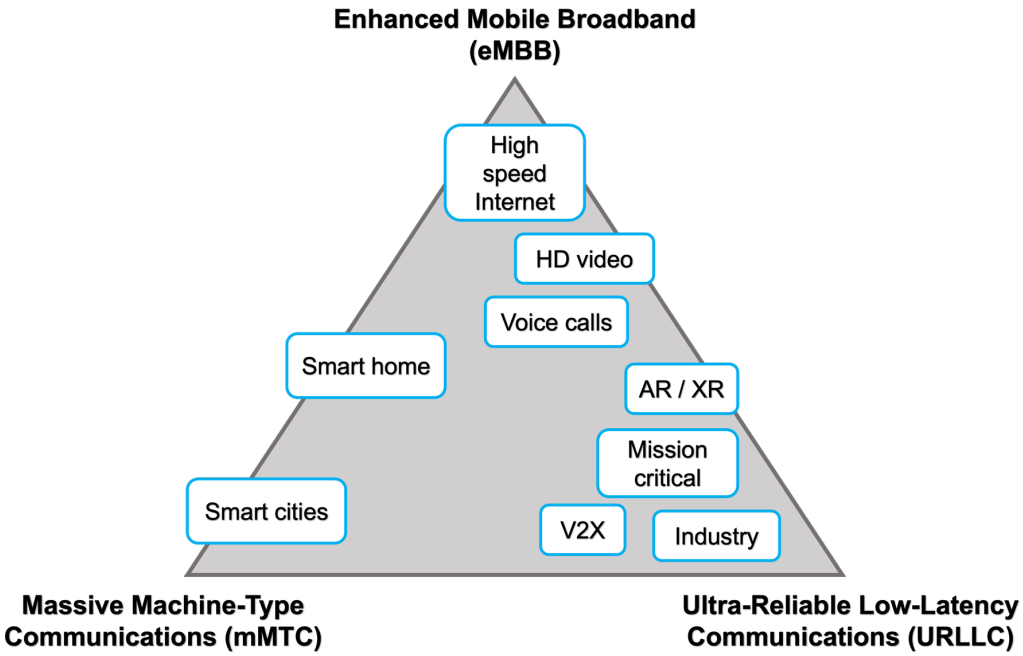

Figura 7: Verticales de 5G y sus casos de uso asociados

eMBB contiene las funcionalidades necesarias para mejorar las comunicaciones entre los usuarios, lo esperado en una nueva generación móvil. Mejoras en la latencia o las velocidades de subida y bajada.

mMTC contiene funcionalidades necesarias para la conexión masiva de dispositivos con requisitos de comunicación bajos, como puede ser el caso de dispositivos *IoT*.

Por último, *URLLC* trae consigo los servicios ultra fiables y de baja latencia, orientados a servicios como aplicaciones críticas, comunicaciones vehiculares (*V2X*), industria 4.0, entre otros.

La selección de los *Slices* se realiza utilizando dos parámetros, el *Slice Service Type* (*SST*) y el *Slice Differentiator* (*SD*). En la actualidad hay definidos seis *Slice Service Types*, estos son eMBB, URLLC, Massive IoT (MIoT), V2X, High-Performance Machine-Type Communications (HMTC) y H*igh data rate and Low Latency Communications* (*HDLLC*).

En la Figura 7: Verticales de 5G y sus casos de uso asociados, el servicio *MIoT* se sitúa con *mMTC* y los servicios *V2X*, *HMTC* y *HDLLC* se sitúan entre *mMTC* por la densidad de dispositivos conectados y *URLLC* por los requerimientos de conexión.

El *Slice Differentiator* es un parámetro extra que permite diferenciar distintos servicios dentro de un mismo *Slice Service Type*.

Para el funcionamiento de estos servicios se requieren ciertas *Network Functions* específicas dentro del núcleo de red. Para ello se crean diferentes redes lógicas que soportan estos servicios, cada uno con unos requisitos diferentes.

El concepto de sesión, en 5G llamado *PDU Session* (*PDU* significa *Packet Data Unit*), es un concepto que ya existía, aunque ha sido actualizado para 5G. La sesión identifica una conexión de un usuario a través de la red móvil. Un usuario puede mantener varias conexiones o *PDU Sessions* al mismo tiempo, estando conectado a diferentes redes, en 5G llamadas *Data Networks* (*DNs*). El concepto de sesión es utilizado en 5G para aplicar desde políticas de *QoS* (*Quality of Service*) hasta el pago por uso aplicado a los planes de telefonía móvil.

El *handover* es el mecanismo por el que un usuario puede cambiar de un lugar de la red a otro manteniendo la conexión. Este concepto es aplicable en muchos aspectos, desde el *handover* entre estaciones base cuando un usuario deja una estación base y salta a otra, hasta el *handover* en tecnologías cuando un usuario deja una zona que tiene cobertura de 5G y pasa a una zona con cobertura 4G. El *handover* es un mecanismo crucial y es lo que aporta el concepto de movilidad a las redes móviles.

El *roaming* es un concepto que permite el funcionamiento de los dispositivos móviles en otras zonas (comúnmente entre países) en los que un operador tiene un contrato con otro operador para proporcionar dicho servicio. Mediante el *roaming* podemos obtener conexión en una zona incluso si nuestro operador no proporciona servicio en la zona que visitamos. Esto se realiza mediante acuerdos entre operadores.

Para proporcionar su funcionalidad, el proyecto *Open5GS* proporciona las siguientes *Network* Functions: *Access and Mobility Management Function* (*AMF*), *Authentication Server Function* (*AUSF*), *Binding Support Function* (*BSF*), *Network Repository Function* (*NRF*), *Network Slice Selection Function* (*NSSF*), *Policy Control Function* (*PCF*), *Service Communication Proxy* (*SCP*), *Security Edge Protection Proxy* (*SEPP*), *Session Management Function* (*SMF*), *Unified Data Management* (*UDM*), *Unified Data Repository* (*UDR*) y el *User Plane Function* (*UPF*).

El *AMF* es el punto de acceso al exterior del plano de control a través del *interfaz N2*. También implementa las funcionalidades para la gestión de la movilidad del usuario. Es una de las *Network Functions* más importantes del plano de control al centralizar toda la comunicación con *gNBs* y los *UEs* a través de ella.

El *AUSF* proporciona los métodos de autenticación y autorización de usuarios en la red. El *AUSF* accede a la información del usuario a través del *UDM*.

El *BSF* permite determinar qué *PCF* está utilizando una sesión, puesto que puede haber varios *PCFs* desplegados.

El *NRF* es el punto de información centralizado de las *Network Functions*. Es utilizada por las *Network Functions* del plano de control para registrarse y para descubrir las direcciones de las demás, antes de ser contactadas para hacer uso de alguno de sus servicios.

El *NSSF* proporciona la información acerca de los *Network Slices* definidos en nuestro núcleo de red, mapeando los parámetros *Slice Service Type* y *Slice Differentiator* a los *Network Slice Instances* (*NSI*) que contienen la información acerca de las *Network Functions* disponibles en cada *Slice*.

El *PCF* es la *Network Function* encargada de obtener la información acerca de las políticas de *QoS* de una sesión, para ello hace uso del *UDR*.

El *SCP* es un *proxy* que proporciona funcionalidades de interoperabilidad y escalabilidad, permitiendo gestionar las comunicaciones entre las *Network Functions*.

El *SEPP* se utiliza en entornos con *roaming*. Para el funcionamiento del *roaming* dos núcleos de red distintos han de interoperar para gestionar la conexión de un usuario. El *SEPP* desplegado en un núcleo de red es el encargado de comunicarse con el *SEPP* desplegado en el otro núcleo de red y a través de esta comunicación gestionar el intercambio de información.

El *SMF* es la función que gestiona los aspectos del plano de control de la sesión y configura el plano de usuario. Junto con el *AMF* es una de las *Network Functions* más importantes del plano de control debido a sus funcionalidades.

El *UDM* es un *frontend* que gestiona datos del usuario de manera centralizada, para ello hace de interfaz de otras *Network Functions* y se comunica con el *UDR*.

El *UDR* es la *Network Function* que contiene la información del usuario, actúa como base de datos.

El *UPF* es el punto de acceso al exterior del plano de usuario a través del *interfaz N3*. Realiza las funciones de enrutamiento de los datos. Es la única *Network Function* encargada del plano de usuario y por ello es fundamental.

De las *Network Functions* nombradas se omitirá el despliegue del *SCP* por no requerir los servicios de interoperabilidad y escalabilidad y del *SEPP* al no operar en *roaming*.

3.5 Núcleo de red 5G en Kubernetes

Como se ha mencionado anteriormente, cada componente del núcleo de red 5G a desplegar requerirá de un objeto *Deployment* distinto. A continuación, se van a mostrar objetos de *Kubernetes* creados para diferentes componentes, con el fin de ilustrar cada una de las partes explicadas.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ausf-deployment
  labels:
    nf: ausf
spec:
  replicas: 1
  selector:
    matchLabels:
      nf: ausf
  template:
    metadata:
      labels:
        nf: ausf
    spec:
      containers:
      - name: ausf
        image: borieher/ausf:v2.7.2
        args: ["-c", "/open5gs/config/ausf.yaml"]
        ports:
        - name: sbi
          containerPort: 80
        volumeMounts:
        - name: ausf-volume-config
          mountPath: /open5gs/config/ausf.yaml
          subPath: ausf.yaml
      volumes:
      - name: ausf-volume-config
        configMap:
          name: ausf-configmap
```

Fragmento 18: Objeto Deployment para la Network Function AUSF

En este *Deployment* se pueden ver el número de réplicas configuradas (**1**), la imagen del contenedor que se está utilizando (**borieher/ausf:v2.7.2**) y los *volumes* asociados al contenedor, en este caso un ConfigMap nombrado ***ausf-volume-config***, mostrado en el siguiente espacio:

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: ausf-configmap
data:
  ausf.yaml: |
    logger:
      file:
        path: /open5gs/install/var/log/open5gs/ausf.log

    global:

    ausf:
      sbi:
        server:
          - dev: eth0
            port: 80
        client:
          nrf:
            - uri: http://nrf-service-sbi.default.svc.cluster.local:80
```

Fragmento 19: Objeto ConfigMap para la Network Function AUSF

En 5G los componentes del plano de control utilizan la llamada *Service Based Architecture* para comunicarse entre sí. Como ya se ha mencionado, el *SBA* se trata de un conjunto de *APIs REST* definidas por los estándares del *3GPP* para la comunicación utilizando *HTTP/2*. Para la comunicación entre los componentes del plano de control, utilizando los llamados *Service Based Interfaces* (*SBI*) se crearán *Services* de *Kubernetes* para permitir la comunicación, en nuestro caso, utilizando el puerto 80.

```
apiVersion: v1
kind: Service
metadata:
  name: ausf-service-sbi
spec:
  selector:
    nf: ausf
  ports:
    - name: sbi
      port: 80
      targetPort: 80
  type: ClusterIP
```

Fragmento 20: Objeto Service ClusterIP para SBI en la Network Function AUSF

Algunos componentes del núcleo de red 5G desplegado (*PCF*[6] y *UDR*[7]) requieren de una comunicación directa con la base de datos *MongoDB* desplegada. Para ello crearemos un *Service* que será utilizado por la base de datos para permitir estas conexiones utilizando el puerto 27017.

En 5G para comunicar los parámetros de configuración del plano de control al plano de usuario existe un protocolo llamado *Packet Forwarding Control Protocol* (*PFCP*). Este protocolo es utilizado entre los componentes *SMF* y *UPF*. Mediante *PFCP* se obtiene la denominada *Control and User Plane Separation* (*CUPS*) y es lo que permite al *SMF* "programar" el plano de usuario del *UPF* utilizando un método parecido al existente en *Software Defined Networks* (*SDN*). El protocolo *PFCP* utiliza *UDP* en el puerto 8805, para su uso crearemos un *Service* también.

Todos los *Services* anteriormente mencionados han sido creados con el tipo por defecto de *Kubernetes*, este es *ClusterIP* para obtener una *IP* alcanzable dentro del clúster de *Kubernetes*, lo que es suficiente para la comunicación interna de estos componentes.

A continuación, se describirán los dos *Services* de tipo *LoadBalancer* que han propiciado la instalación de *MetalLB* en el clúster y que han de ser alcanzables por componentes externos al clúster de *Kubernetes*.

La conexión entre el núcleo de red 5G en el plano de control y los componentes externos (*gNBs*, *N3IWF* y demás) es a través del *interfaz N2*. El *interfaz N2* utiliza el protocolo *NGAP* que está basado sobre el protocolo de transporte *SCTP* en el puerto 38412. *SCTP* es un protocolo de transporte situado entre *TCP* y *UDP*, que implementa algunas de las funcionalidades de *TCP* como la fiabilidad y que está orientado a conexión con algunas de las funcionalidades de *UDP* como permitir la transmisión desordenada de paquetes. Es un protocolo utilizado generalmente en el mundo de las telecomunicaciones, creado en sus inicios para el transporte de señalización telefónica sobre *IP* [42].

Este punto de entrada entre el exterior y el núcleo de red 5G está situado en el componente *AMF*, por lo que crearemos un *Service* de *Kubernetes* de tipo *LoadBalancer* para utilizar el protocolo *NGAP* y lo utilizaremos en el *Deployment* del *AMF*.

```
apiVersion: v1
kind: Service
metadata:
```

[6] El PCF de Open5GS es un caso específico en cuanto a su arquitectura, en el que esta Network Function interactúa directamente con la base de datos sin comunicarse previamente con el UDR.
[7] El UDR de Open5GS no contiene los datos en sí, interactúa con la base de datos MongoDB desplegada para obtenerlos.

```
  name: amf-service-ngap
  annotations:
    metallb.universe.tf/loadBalancerIPs: 192.168.0.142
spec:
  selector:
    nf: amf
  ports:
    - name: ngap
      port: 38412
      targetPort: 38412
      protocol: SCTP
  type: LoadBalancer
```

Fragmento 21: Objeto LoadBalancer para NGAP en la Network Function AMF

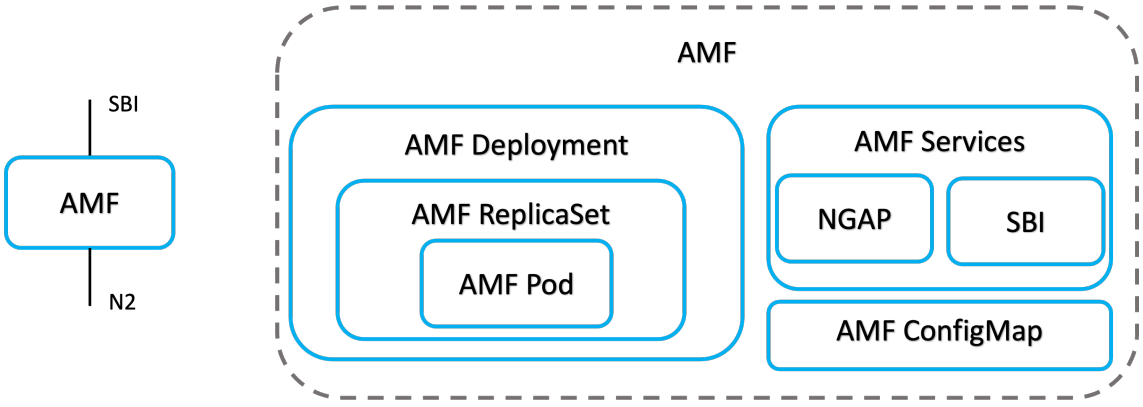

Figura 8: Ejemplo objetos de Kubernetes para Network Function AMF

La conexión entre el núcleo de red 5G en el plano de usuario y los componentes externos es a través del *interfaz N3*. El *interfaz N3* utiliza el protocolo *GTPU* que está basado sobre el protocolo de transporte *UDP* en el puerto 2152. GTPU es un subconjunto dentro del protocolo GTP (*GPRS Tunnelling Protocol*) originado para la red *GPRS*. *GTPU* se utiliza para encapsular el tráfico de los usuarios en las redes móviles. Cada usuario generará un túnel por sesión.

Este punto de entrada entre el exterior y el núcleo de red 5G está situado en el componente *UPF*, por lo que crearemos un *Service* de *Kubernetes* de tipo *LoadBalancer* para utilizar el protocolo *GTPU* y lo utilizaremos en el *Deployment* del *UPF*.

En un primer lugar el aprovisionamiento de usuarios en la base de datos *MongoDB* se realizaba a través de un *Service* de tipo *NodePort* conectado a la base de datos. A través de este *Service* y utilizando un *script* (*open5gs-dbctl,* ver anexo 7.3) proporcionado por el proyecto *Open5GS* se añadían o eliminaban usuarios en la red.

Este método quedó obsoleto rápidamente, cuando se comprobó la comodidad de utilizar el componente *WebUI* proporcionado también por el proyecto

Open5GS. Tras una prueba inicial, se añadió el componente al proyecto *docker-open5gs* como un contenedor más para poder ser utilizado como los demás componentes de este proyecto.

Para el acceso a la interfaz web del componente *WebUI* se creará un *Service* de tipo *NodePort*. El *Service NodePort* permite al clúster abrir un puerto estático, accesible desde la *IP* de los nodos del clúster, que permitirá el acceso a ese servicio a través de la *IP* de cualquier nodo del clúster.

```
apiVersion: v1
kind: Service
metadata:
  name: webui-service
spec:
  selector:
    app: webui
  ports:
    - name: webui
      port: 9999
      targetPort: 9999
      nodePort: 30999
  type: NodePort
```

Fragmento 22: Objeto Service NodePort para acceso a WebUI

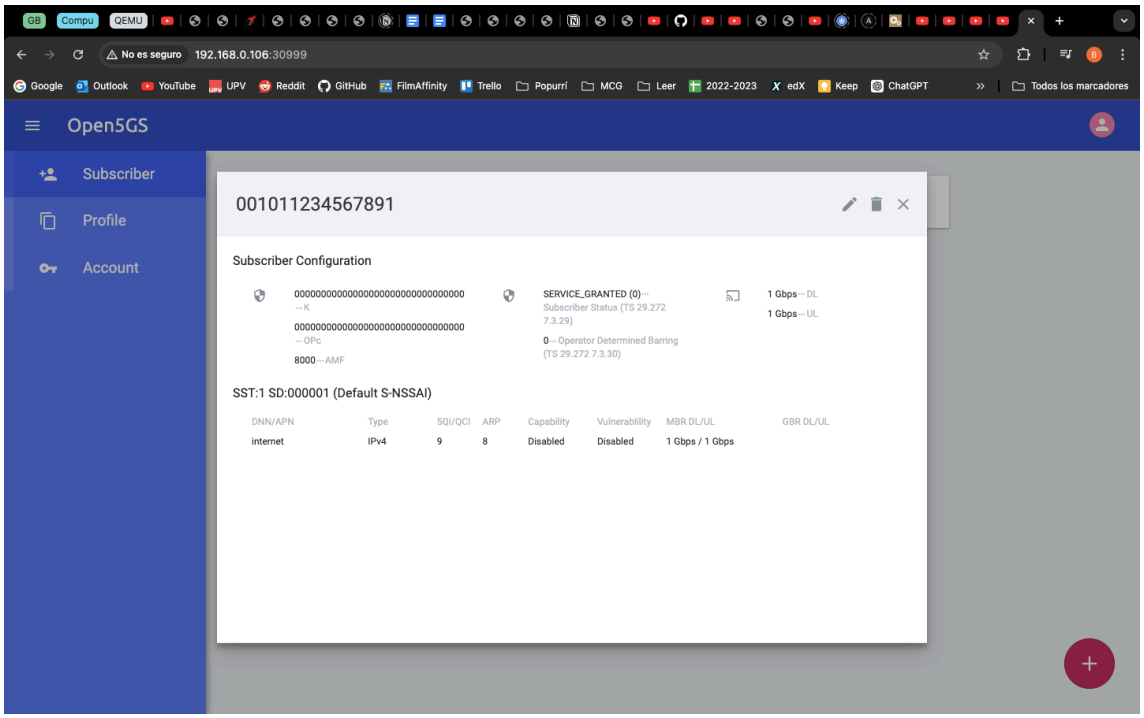

Figura 9: Acceso a WebUI

La base de datos *MongoDB* de nuestro sistema requiere de almacenamiento persistente. Para ello crearemos un *PersistentVolumeClaim* para solicitar almacenamiento mediante un *PersistentVolume*.

Como prueba inicial, el *PersistentVolume* fue creado estáticamente para cumplir los requisitos del *PersistentVolumeClaim* solicitado por la base de datos. Mediante un *PersistentVolumeClaim* un componente puede solicitar el aprovisionamiento de almacenamiento con distintas características como pueden ser la capacidad de almacenamiento, rendimiento o los modos de acceso.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-pvc-data
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
```

Fragmento 23: Objeto PersistentVolumeClaim para la base de datos

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: db-pv-data
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  hostPath:
    path: /mnt/db_data
```

Fragmento 24: Objeto PersistentVolume para la base de datos

Tras las pruebas iniciales se comprobó la necesidad de la instalación de un sistema de aprovisionamiento dinámico como son las *StorageClasses* de *Kubernetes* para facilitar el aprovisionamiento de *PersistentVolumes*.

Una vez comprobados los requisitos de la instalación de *Kubernetes* y los requisitos específicos para el despliegue de nuestra aplicación se comenzó a trabajar en la automatización de la instalación de *Kubernetes* utilizando *Ansible*.

3.6 Creación de roles de Ansible

Ansible es la herramienta seleccionada para la configuración y gestión de máquinas de manera remota.

Para la gestión se utilizará una máquina externa en la que instalaremos *Ansible* utilizando *Python*. Se recomienda esta forma de instalación para asociar una versión de *Python* a la versión de *Ansible* instalada. Utilizando el gestor de paquetes de *Python pip* podremos realizar la instalación y las actualizaciones posteriores.

Tras la instalación el siguiente paso es habilitar *SSH* en las máquinas a gestionar para poder ejecutar comandos.

Para el envío de comandos *Ansible* utiliza un fichero llamado *inventory*. En el *inventory* se encuentra la información necesaria para conectar con las máquinas, así como otros parámetros relevantes que podemos utilizar.

```
---
control_plane:
  hosts:
    master:
      ansible_host: k8s-master
      ansible_user: borjis
      ansible_ssh_private_key_file: ~/.ssh/borjis-rsa
      kubernetes_role: control_plane

nodes:
  hosts:
    node1:
      ansible_host: k8s-node-1
      ansible_user: borjis
      ansible_ssh_private_key_file: ~/.ssh/borjis-rsa
      kubernetes_role: node

    node2:
      ansible_host: k8s-node-2
      ansible_user: borjis
      ansible_ssh_private_key_file: ~/.ssh/borjis-rsa
      kubernetes_role: node

k8s_cluster:
  children:
    control_plane:
    nodes:
```

Fragmento 25: Fichero inventory para la infraestructura on-premises

En el ejemplo de *inventory*, además de los nombres de *host*, nombres de usuario y la clave privada, hay dos parámetros relevantes mostrados. El primero se trata de la asignación de las máquinas en grupos. Bajo del todo, en el ejemplo aparece el grupo llamado **k8s_cluster** que contiene a los grupos **control_plane** y **nodes**, definidos arriba. En los *playbooks* se puede

referenciar a varias máquinas utilizando los nombres de grupo para así ejecutar comandos solo en los grupos seleccionados.

Otro detalle es la variable **kubernetes_role** utilizada en el *rol de Ansible* de *Kubernetes* que mencionaremos más adelante.

Una manera sencilla de comenzar con *Ansible* es a través del comando **ansible-playbook**. Este comando sirve para ejecutar un *playbook*, aunque para simplificar el entendimiento reduciremos el *playbook* a su expresión mínima, una única *task*.

```
---
- name: Update servers
  hosts: servers


  tasks:
  - name: Update the apt package index
    ansible.builtin.apt:
      update_cache: yes
      cache_valid_time: 3600
```

Fragmento 26: Ejemplo básico de Ansible playbook

En *Ansible* una *task* es una operación realizada sobre una máquina o grupo de máquinas del *inventory*. Para poder ejecutar estas operaciones *Ansible* define los *modules*. Los *modules* son los elementos encargados de realizar la ejecución en las máquinas, están creados por *Ansible* o por la comunidad. A través de los modules realizamos operaciones en las máquinas como actualizar el gestor de paquetes *apt* (en el ejemplo mostrado arriba, utilizando **ansible.builtin.apt**) o la gestión de servicios de *systemd* a través del *module* **ansible.builtin.service**.

Los *modules* se agrupan en *collections*. Los *modules* que comienzan por *ansible.builtin* vienen por defecto con *Ansible*, mientras que otros *modules* como los pertenecientes a la *collection amazon.aws* pueden ser instalados para añadir nueva funcionalidad.

Una agrupación de *tasks* que se ejecutan sobre una máquina o grupo se le denomina *play* y un conjunto de *plays* es un *playbook*. Un *playbook* puede contener uno o varios *plays* que pueden ser ejecutados en diferentes máquinas.

Un concepto fundamental en *Ansible* es el concepto de idempotencia. Cuando una máquina ha sido configurada, por ejemplo, utilizando un *playbook*, volver a ejecutar ese *playbook* debe dejar a la máquina en el mismo estado en el que estaba. Este concepto ha de estar presente a la hora de crear *playbooks* con *Ansible*.

En casos en los que las *tasks* son más complejas de estructurar y para permitir su reutilización, *Ansible* define los *roles*. Los *roles* en *Ansible* son una estructura de ficheros que permite organizar variables, *tasks*, ficheros, *handlers*, entre otros.

La estructura de un *rol de Ansible* es la siguiente:

```
example/
  tasks/
  handlers/
  templates/
  files/
  vars/
  defaults/
  meta/
  library/
  module_utils/
  lookup_plugins/
```

Fragmento 27: Estructura de un rol de Ansible

La estructura del ejemplo muestra los directorios existentes para un *rol de Ansible* llamado "*example*".

En el interior del directorio *tasks* encontraremos los ficheros *YAML* que contienen las *tasks* a ejecutar por el *rol*. Otras *tasks* se pueden encontrar también en el directorio *handlers*, que tienen un uso especial y es que se ejecutan tras dar el aviso desde otra *task*. Un ejemplo de *handler* sería el reinicio de una máquina para aplicar la configuración modificada por una *task*.

Los directorios *vars* y *defaults* contienen variables que pueden ser utilizadas en las *tasks*, mientras que los directorios *templates* y *files* contienen plantillas y ficheros de apoyo necesarios para nuestras *tasks*.

Por último, los directorios *meta*, *library*, *module_utils* y *lookup_plugins* son para definir las dependencias del *rol* en cuanto a otros *roles*, *modules*, *module*_utils o lookup_*plugins* respectivamente.

Tras la instalación de *Kubernetes* manual se identificaron cuántos *roles de Ansible* vamos a crear para el TFM. Se identificaron cuatro *roles* diferentes: *raspberry-pi, containerd, kubernetes* y *metallb*.

La función principal de cada uno de los *roles* es la siguiente:

- **raspberry-pi**: Sólo será utilizado en el contexto del clúster *on-premises*. Se encargará de configurar los parámetros del *kernel* necesarios para la activación de los *cgroups* en las *Raspberry Pi*.
- **containerd**: Utilizado para instalar *containerd* como *container runtime* para *Kubernetes*.

- **kubernetes**: Se encargará de instalar Kubernetes, así como un *network plugin* para el funcionamiento de la red. En el caso de este TFM, será *Calico*.
- **metallb**: Instalará *MetalLB* sobre el clúster de *Kubernetes*.

Se utilizaron los pasos mostrados en la instalación para dividir en *tasks* los pasos a seguir por cada *rol de Ansible*. Cada vez que se identificaba una nueva *task*, ésta era definida y ejecutada en un *playbook* independiente para verificar el correcto funcionamiento de la *task* y los parámetros del *module* seleccionado.

A continuación, se van a poner como ejemplo algunas *tasks* diseñadas y utilizadas en los *roles* mencionados.

```
- name: Install kubelet, kubeadm and kubectl using apt
  become: yes
  ansible.builtin.apt:
    name:
      - kubelet
      - kubeadm
      - kubectl
    state: present
```

Fragmento 28: Ansible task para instalar kubelet, kubeadm y kubectl utilizando apt

Por lo general, con una selección del *module* correcto a utilizar, la creación de las *tasks* es una traducción de los comandos utilizados “a mano”. La dificultad reside en la definición de variables para permitir la configuración del *rol* por parte del usuario.

A continuación se comentarán algunas de las variables utilizadas para la configuración de los *roles* por parte del usuario.

Para el *rol* de *containerd* se permite la selección del estado del servicio de *systemd* de *containerd* mediante el uso de las variables **service_containerd_state** y **service_containerd_enabled**. También se permite la selección del canal de *release* del repositorio que vamos a instalar utilizando la variable **apt_docker_release_channel**, siendo el por defecto *stable*.

En el caso del *rol* de *kubernetes* se permite la selección de la versión de *Kubernetes* con la variable **apt_kubernetes_version**, el *network plugin* con la variable **kubernetes_pod_network_add_on** (actualmente solo soporta *Calico*) o el rango de dirección de red a utilizar por el *network plugin* con **kubernetes_pod_network_cidr**.

Una de las dificultades a la hora de diseñar las *tasks* se presentó en la creación del clúster de *Kubernetes*. Como se ha mostrado previamente, la creación del clúster de *Kubernetes* comienza por la configuración del plano de control. Tras su configuración este, se utilizan una serie de parámetros como son la dirección *IP* del plano de control o el *token* para permitir que las demás máquinas, que serán nodos, se unan al clúster.

Ansible facilita mucho la ejecución de comandos en máquinas remotas desde la máquina central, pero en este caso particular que requiere de hacer uso de parámetros almacenados en la máquina de control desde los nodos, hay que ponerse creativos. Tras la ejecución del comando **kubeadm init** en el plano de control, solicitaremos la información requerida al plano de control con el comando **kubeadm token create --print-join-command**. El *task* es el siguiente:

```
- name: Get kubeadm join command
  ansible.builtin.command: "kubeadm token create --print-join-
command"
  when: kubernetes_role == "control_plane"
  register: join_command_result
```

Fragmento 29: Obteniendo el token desde el plano de control

Con la cláusula *when* limitamos qué clase de máquinas pueden ejecutar este comando, en este caso las que tienen la variable **kubernetes_role** como control plane. Y tras ejecutar el comando, guardamos la salida del comando en la variable **join_command_result**.

Tras esto, vamos a repartir la información a los nodos. *Ansible* define el concepto de *fact*, un *fact* es información acerca de una máquina en concreto. Hay información que se puede extraer de la conexión con la máquina y también se puede añadir información extra que pueda resultar útil. En este caso se va a compartir el resultado del comando añadiendo como *fact* el contenido de la variable utilizando el *module set_fact*, de este modo, cada nodo tendrá una copia del contenido de la variable. El *task* es el siguiente:

```
- name: Set kubeadm join command as ansible fact in the nodes
  ansible.builtin.set_fact:
    kubernetes_join_command: "{{ join_command_result.stdout }}"
  when: join_command_result.stdout is defined
  delegate_to: "{{ item }}"
  delegate_facts: true
  with_items: "{{ groups['nodes'] }}"
```

Fragmento 30: Repartiendo el token a los nodos desde el plano de control

Con la cláusula *when* forzamos a que solo se ejecute este *task* cuando el contenido de la variable **join_command_result** está definido. Vamos a ejecutar este *task* sobre todos los *items* del grupo *nodes*, que contiene a todas las máquinas que serán nodos de nuestro clúster. Con la combinación de la cláusula *when* y los *items* garantizamos que esto solo se ejecutará desde el plano de control a todos los nodos.

Por último, para que los nodos ejecuten el comando recibido a través de los *facts*, se crea la siguiente *task*:

```
- name: Setup kubernetes node
  become: yes
  ansible.builtin.command: "{{ kubernetes_join_command }}"
  when: kubernetes_role == "node"
```

Fragmento 31: Ejecución del comando join con el token desde los nodos

Otra dificultad encontrada con *Ansible* ha sido la ejecución de comandos dentro de *Kubernetes*, requerida en el caso de las instalaciones como las de *Calico* o de *MetalLB*.

La creación de algunos recursos de *Kubernetes* puede tardar un tiempo y para garantizar la creación del recurso se ha llevado a cabo la siguiente *task*:

```
- name: Install MetalLB
  ansible.builtin.command: "kubectl apply -f {{ metallb_manifest_url }}"
  register: metallb_result
  changed_when: "'created' in metallb_result.stdout or 'already exists' in
metallb_result.stdout"
  until: metallb_result is not failed
  retries: 10
  delay: 5
```

Fragmento 32: Creación de recursos en Kubernetes desde Ansible

La *task* ejecuta el comando utilizando *kubectl*, utilizando la variable **metallb_manifest_url** que tiene la *URL* del *manifest* de *MetalLB* a instalar y comprueba diez veces, esperando cinco segundos tras cada intento que la salida del comando tenga en sus contenidos "*created*" o "*already exists*".

Estos son algunos ejemplos de *tasks* que han supuesto una dificultad mayor, mientras que el resto de *tasks* o repiten patrones mostrados o son más básicas solo requiriendo del uso de diferentes modules. Para una visualización de todas las *tasks* creadas para los *roles* de este TFM, visitar el repositorio en GitHub de la siguiente referencia [43].

Una vez creados los *roles*, su ejecución se realiza a través de un *playbook* en el que definimos los *roles* a ejecutar. El *playbook* final que contiene todos los *roles* para la instalación *on-premises* es el siguiente:

```
---
- hosts: k8s_cluster
  roles:
    - raspberry-pi
    - containerd
    - kubernetes
    - metallb
```

Fragmento 33: Ansible playbook utilizado en la infraestructura on-premises

Este *playbook* ejecuta los cuatro *roles* creados sobre el grupo **k8s_cluster** definido en el *inventory* mostrado.

Tras el desarrollo de los *roles de Ansible* se pasó a la creación de los *Helm Charts* para facilitar el despliegue del núcleo de red 5G en *Kubernetes*.

3.7 Creación de los Helm Charts

Helm es una herramienta que permite el despliegue y la gestión de aplicaciones en *Kubernetes* de una manera sencilla, ocultando gran parte de la complejidad del despliegue y permitiendo al usuario la edición de parámetros relativos al despliegue.

El objetivo de empaquetar el núcleo de red 5G en un *Helm Chart* es facilitar su despliegue y configuración, puesto que tras las primeras pruebas se comprobó la incomodidad de tener que desplegar los cuarenta objetos de *Kubernetes* necesarios para desplegar nuestra aplicación. En un principio se optó por un *script,* pero tras el descubrimiento de *Helm* se optó por portar la aplicación al formato para empaquetarlo en un *Chart*.

En *Helm* un *Chart* es el resultado del empaquetado de una aplicación de *Kubernetes*. Utilizaremos un *Chart* para desplegar una aplicación. Desde el punto de vista del usuario, *Helm* proporciona un *Chart* específico cuya configuración se puede modificar a través de un fichero llamado *values.yaml.* Con los comandos del ejecutable *helm*, se despliega el *Chart* con una configuración específica, lo que en el contexto de *Helm* se conoce como una *release*.

Los *Helm Charts* son un conjunto de ficheros que contienen los objetos utilizados por *Kubernetes* utilizando una sintaxis de plantillas para sustituir los valores configurables.

La estructura de ficheros un *Helm Chart* es la siguiente:

```
example/
  Chart.yaml
  LICENSE
  README.md
  values.yaml
  values.schema.json
  charts/
  crds/
  templates/
```

Fragmento 34: Estructura de un Helm Chart

La estructura del ejemplo muestra los directorios y ficheros existentes para un *Helm Chart* llamado "*example*".

El fichero *Chart.yaml* contiene información acerca del *Chart*, como la versión del *Chart* o la versión de la aplicación empaquetada en el *Chart*, así como las dependencias del *Chart* o el tipo. Existen dos tipos de *Charts*, *application* o *library*, los *Chart* de tipo *library* son especiales y proveen utilidades para la construcción de otros *Charts*, mientras que los *Chart* de tipo *application* son los más comunes y sirven para el empaquetamiento de aplicaciones.

Los ficheros *LICENSE*, *README.md* y *values.schema.json* son opcionales y contienen la licencia que aplica al *Chart*, información extra acerca del *Chart* para usuarios o el *JSON schema* para validar la estructura y el contenido del fichero *values.yaml* proporcionado al *Chart* por el usuario.

El fichero *values.yaml* contiene los valores por defecto del *Chart*, existe una precedencia en la aplicación de los parámetros de configuración según cómo son proporcionados por el usuario que será brevemente mencionado en la sección donde se explica la construcción de las *Charts* de este TFM.

El directorio *charts* contiene los *Charts* de los que depende el *Chart example* mientras que el directorio *templates* contiene plantillas que tras aplicar los parámetros de configuración del *values.yaml* generarán los *manifests* a utilizar en *Kubernetes*. *Helm* proporciona una sintaxis basada en el *Go Template language* y otras funciones que añade *Helm*, para renderizar los valores de los *values.yaml* en las plantillas. En el directorio *template* se pueden añadir ficheros *tpl* que contienen funciones y herramientas customizadas para utilizar en nuestras plantillas.

Por último, el directorio *crds* contiene los *Custom Resource Definitions* de *Kubernetes* para nuestra aplicación, en caso de hacer uso de alguno. Estos objetos son creados los primeros y no pueden incluir la sintaxis de plantillas.

Para la creación de nuestra aplicación se crearon *Charts* independientes por cada uno de los elementos del núcleo de red 5G y se creó un *Chart* general, llamado *open5gs* que contiene los elementos del núcleo de red 5G como *subcharts* y permite la configuración de todos los parámetros de una manera centralizada, facilitando su despliegue. Este *Chart* centralizado permite también habilitar o deshabilitar distintos componentes del núcleo de red si no queremos que sean utilizados en ese despliegue.

Como ejemplo se van a utilizar algunos de los ficheros incluidos en el *Helm Chart* del *AUSF*:

```
{{/*
Return the proper ausf image name
*/}}
{{- define "ausf.image" -}}
{{ include "common.images.image" (dict "imageRoot" .Values.image "global"
.Values.global) }}
{{- end -}}


{{/*
Return the proper ausf config file
*/}}
{{- define "ausf.config" -}}
{{- if .Values.config }}
{{- toYaml .Values.config | nindent 4 }}
{{- else }}
{{ tpl (.Files.Get "configs/ausf.yaml") . | indent 4 }}
{{- end }}
{{- end -}}
```

Fragmento 35: Fichero _helpers.tpl dentro del directorio templates del Chart ausf

El fichero *_helpers.tpl* mostrado define dos utilidades, **ausf.image** y **ausf.config**. Estas utilidades nos facilitan la obtención de la imagen del contenedor utilizado por el *Chart ausf*, así como su configuración. Estas utilidades serán utilizadas en otras plantillas del *Chart*.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "common.names.fullname" . }}-deployment
  namespace: {{ include "common.names.namespace" . }}
  labels:
    nf: ausf
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      nf: ausf
  template:
    metadata:
      labels:
        nf: ausf
    spec:
      containers:
      - name: {{ include "common.names.fullname" . }}
        image: {{ template "ausf.image" . }}
        args: ["-c", "/open5gs/config/ausf.yaml"]
        ports:
        - name: sbi
          containerPort: {{ .Values.global.containerPorts.sbi }}
        volumeMounts:
        - name: ausf-volume-config
          mountPath: /open5gs/config/ausf.yaml
          subPath: ausf.yaml
      volumes:
      - name: ausf-volume-config
        configMap:
          name: {{ include "common.names.fullname" . }}-configmap
```

Fragmento 36: Plantilla deployment.yaml dentro del directorio templates/ del Chart ausf

Esta plantilla es la que será renderizada por *Helm*, tomando los valores del *values.yaml* y aplicando las funciones que podemos ver en el fichero para crear el objeto *Deployment* de *Kubernetes*.

Otras plantillas utilizadas en el *Chart* del *ausf* son la plantilla para su configuración que se transformará en el objeto *ConfigMap* y la plantilla para el servicio del *SBI* que se transformará en el objeto *Service*. Para configurar estos valores, el *Chart* del *ausf* define el siguiente *values.yaml*:

```
image:
  registry: ghcr.io
  repository: borjis131/ausf
  tag: "v2.7.2"
  pullPolicy: IfNotPresent

config: {}

global:
  containerPorts:
    sbi: 80
```

Fragmento 37: Fichero values.yaml del Chart ausf

En el fichero se pueden ver los valores seleccionados de la imagen del contenedor para el *AUSF*, un objeto **config** que sobreescribe por completo la configuración por defecto añadida al *ConfigMap* y en la sección **global** el puerto utilizado por el *Service SBI*.

Utilizando los objetos de *Kubernetes* creados para el núcleo de red 5G previamente como base se crean los demás *Charts*. La selección de los parámetros que van tras la sección *global* se hizo tras la creación del *Chart* general y será explicada más adelante.

Un apartado importante en *Kubernetes* que no se ha mencionado hasta el momento son los *Namespaces* y los *selectors*. En *Kubernetes* los *Namespaces* nos permiten agrupar y gestionar los recursos de manera aislada. Con la migración a *Helm*, se añadieron *Namespaces* a los objetos creados para el núcleo de red, permitiendo así desplegar varios *Charts* en *Namespaces* diferentes y obtener una mayor organización.

Por otro lado, los *selectors* son un mecanismo que nos permite emparejar recursos. En los *Charts* se ha hecho uso de *label selectors* para emparejar los *Pods* desplegados con el objeto *Service* y también para vincular el *ReplicaSet* creado por el *Deployment* con el *Pod* desplegado.

A continuación, se van mostrar los aspectos más relevantes de la creación del *Chart* general *open5gs*. Este *Chart* ha sido creado para la gestión centralizada de los diferentes *subcharts* que componen el núcleo de red 5G. Este *Chart* solo contiene el archivo *Chart.yaml*, en el que listamos todos los *subcharts* como dependencias y el archivo *values.yaml* desde el que configuramos los parámetros de todos los *Charts* que componen nuestro núcleo de red.

En el *Chart.yaml* habilitamos una condición en los *subcharts* y es que desde el *values.yaml* podemos deshabilitar el uso de los *subcharts* que no interesen. Así existe una manera centralizada de desplegar varios *Charts* desde el *Chart* general *open5gs* sin la necesidad de las demás dependencias. El *values.yaml* del *Chart open5gs* queda así:

```
global:
  containerPorts:
    db: 27017
    sbi: 80
    webui: 9999
  mobileNetwork:
    name: Open5GS
    plmn:
      mcc: "001"
      mnc: "01"
      tac: 1
```

```
      s_nssai:
        - sst: 1
          sd: "000001"
    dataNetwork:
      subnet: 10.45.0.0/16
      gateway: 10.45.0.1
      dnn: internet

db:
  enabled: true
  services:
    db:
      type: NodePort
      nodePort: 30007

webui:
  enabled: true
  services:
    webui:
      type: NodePort
      nodePort: 30999

nrf:
  enabled: true

ausf:
  enabled: true

bsf:
  enabled: true

nssf:
  enabled: true

pcf:
  enabled: true

udm:
  enabled: true

udr:
  enabled: true

amf:
  enabled: true
  services:
```

```
    ngap:
      type: LoadBalancer
      loadBalancerIP: 10.33.0.2
      provider: MetalLB

smf:
  enabled: true

upf:
  enabled: true
  services:
    gtpu:
      type: LoadBalancer
      loadBalancerIP: 10.33.0.3
      provider: MetalLB
```

Fragmento 38: Fichero values.yaml del Chart open5gs

Desde la sección **global** podemos cambiar parámetros de configuración comunes a varios *subcharts* desde un único punto. Valores como el puerto utilizado por los *Services SBI*, *WebUI* o *DB*. También se pueden configurar parámetros de configuración de la red móvil como el nombre de la red, parámetros de *Network Slicing* modificando los valores tras **mobileNetwork**. Dentro de **mobileNetwork** se encuentra la sección **dataNetwork** con los parámetros de configuración de la red a la que los usuarios se van a conectar.

Fuera de la sección **global** tenemos secciones por cada uno de los *subcharts* incluidos, indicando si ese *Chart* está habilitado o no con el parámetro **enabled** o modificando parámetros más específicos de cada *Chart* como pueden ser los *Services NGAP* en el caso del *Chart amf* o *GTPU* en el caso del *Chart upf*.

La precedencia de valores en *Helm* permite modificar los valores de un *subchart* a través del *Chart* que las contiene, en este caso *open5gs* puede modificar los valores por defecto de todos los *subcharts* de los componentes del núcleo de red 5G. En el caso de valores que han de ser modificados desde un único punto en el *Chart* y ser aplicados a todos los *subcharts* correspondientes, hacemos uso de la sección ***global***, esta sección es accesible por todos los *Charts*.

Desde el *Chart* general podemos cambiar todos los valores de los *subcharts* uno a uno o especificar valores comunes que son compartidos por varios *subcharts*.

Tras la creación del *Chart* general, se puede instanciar el núcleo de red 5G utilizando un único *Helm Chart* y un archivo *values.yaml* con los parámetros de configuración específicos para el despliegue.

Para desplegar el *Chart open5gs* haremos uso de los siguientes comandos:

```
helm pull oci://registry-1.docker.io/borieher/open5gs --version 0.3.1

helm install --create-namespace -n upv core1 -f values.yaml ./open5gs-0.3.1.tgz
```

Fragmento 39: Comandos para desplegar el Chart open5gs

El primer comando descarga el *Chart open5gs* versión 0.3.1 del repositorio ubicado en *DockerHub* y el segundo comando despliega un *release* llamado *core1* en el *Namespace* creado *upv*.

```
admin@ip-172-31-18-207:~$ kubectl -n upv get all
NAME                                        READY   STATUS    RESTARTS      AGE
pod/core1-amf-deployment-5d4765b857-qq2rm   1/1     Running   0             47m
pod/core1-ausf-deployment-d7864f77f-k2kp6   1/1     Running   0             47m
pod/core1-bsf-deployment-5bd485c997-44nbn   1/1     Running   0             47m
pod/core1-db-deployment-58dd669655-htgqr    1/1     Running   0             47m
pod/core1-nrf-deployment-688c9c57d4-wkdq8   1/1     Running   0             47m
pod/core1-nssf-deployment-5564ffd7c8-nwrxn  1/1     Running   0             47m
pod/core1-pcf-deployment-5965ff6455-g59kt   1/1     Running   3 (46m ago)   47m
pod/core1-smf-deployment-8654d59688-ntn9m   1/1     Running   0             47m
pod/core1-udm-deployment-6c87bcd58b-gpjcc   1/1     Running   0             47m
pod/core1-udr-deployment-849849b4dc-xcjbw   1/1     Running   3 (46m ago)   47m
pod/core1-upf-deployment-6697446d8b-fvjr6   1/1     Running   0             47m
pod/core1-webui-deployment-65d959cdbd-jqgh7 1/1     Running   0             43m

NAME                              TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)             AGE
service/core1-amf-service-ngap    LoadBalancer   10.99.252.205    172.31.9.9    38412:31735/SCTP    47m
service/core1-amf-service-sbi     ClusterIP      10.106.252.10    <none>        80/TCP              47m
service/core1-ausf-service-sbi    ClusterIP      10.98.44.42      <none>        80/TCP              47m
service/core1-bsf-service-sbi     ClusterIP      10.105.207.38    <none>        80/TCP              47m
service/core1-db-service          NodePort       10.99.213.90     <none>        27017:30007/TCP     47m
service/core1-nrf-service-sbi     ClusterIP      10.96.3.78       <none>        80/TCP              47m
service/core1-nssf-service-sbi    ClusterIP      10.98.215.5      <none>        80/TCP              47m
service/core1-pcf-service-sbi     ClusterIP      10.110.215.235   <none>        80/TCP              47m
service/core1-smf-service-pfcp    ClusterIP      10.108.126.75    <none>        8805/UDP            47m
service/core1-smf-service-sbi     ClusterIP      10.109.247.147   <none>        80/TCP              47m
service/core1-udm-service-sbi     ClusterIP      10.99.160.235    <none>        80/TCP              47m
service/core1-udr-service-sbi     ClusterIP      10.98.19.245     <none>        80/TCP              47m
service/core1-upf-service-gtpu    LoadBalancer   10.100.166.95    172.31.9.10   2152:31039/UDP      47m
service/core1-upf-service-pfcp    ClusterIP      10.107.87.199    <none>        8805/UDP            47m
service/core1-webui-service       NodePort       10.96.56.201     <none>        9999:30999/TCP      47m

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/core1-amf-deployment    1/1     1            1           47m
deployment.apps/core1-ausf-deployment   1/1     1            1           47m
deployment.apps/core1-bsf-deployment    1/1     1            1           47m
```

```
deployment.apps/core1-db-deployment      1/1     1            1           47m
deployment.apps/core1-nrf-deployment     1/1     1            1           47m
deployment.apps/core1-nssf-deployment    1/1     1            1           47m
deployment.apps/core1-pcf-deployment     1/1     1            1           47m
deployment.apps/core1-smf-deployment     1/1     1            1           47m
deployment.apps/core1-udm-deployment     1/1     1            1           47m
deployment.apps/core1-udr-deployment     1/1     1            1           47m
deployment.apps/core1-upf-deployment     1/1     1            1           47m
deployment.apps/core1-webui-deployment   1/1     1            1           47m

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/core1-amf-deployment-5d4765b857   1         1         1       47m
replicaset.apps/core1-ausf-deployment-d7864f77f   1         1         1       47m
replicaset.apps/core1-bsf-deployment-5bd485c997   1         1         1       47m
replicaset.apps/core1-db-deployment-58dd669655    1         1         1       47m
replicaset.apps/core1-nrf-deployment-688c9c57d4   1         1         1       47m
replicaset.apps/core1-nssf-deployment-5564ffd7c8  1         1         1       47m
replicaset.apps/core1-pcf-deployment-5965ff6455   1         1         1       47m
replicaset.apps/core1-smf-deployment-8654d59688   1         1         1       47m
replicaset.apps/core1-udm-deployment-6c87bcd58b   1         1         1       47m
replicaset.apps/core1-udr-deployment-849849b4dc   1         1         1       47m
replicaset.apps/core1-upf-deployment-6697446d8b   1         1         1       47m
replicaset.apps/core1-webui-deployment-65d959cdbd 1         1         1       47m
```

Fragmento 40: Recursos desplegados para el núcleo de red 5G

Tras comprobar la facilidad para desplegar la aplicación utilizando el *Helm Chart*, se buscó la manera de evitar tener que crear de manera manual antes de cada despliegue el *PersistentVolume* para la base de datos.

Para realizarlo se analizaron los *StorageClasses* de *Kubernetes*, en busca de un *StorageClass* que permita crear un *PersistentVolume* de manera dinámica para la base de datos.

En un primer vistazo todos los *storageClasses* existentes parecían precisar de algún tipo de almacenamiento especial como *NFS*, lo cual requiere de un montaje de una infraestructura especial para ello, o almacenamiento en el *cloud*, que en el caso de nuestro clúster *on-premises* no disponemos.

Se encontró el *storageClass local-path-provisioner* de la empresa *Rancher*, el cúal permite el aprovisionamiento de manera dinámica de almacenamiento local, exactamente lo que estábamos haciendo de manera estática con la creación del *PersistentVolume*.

Con la instalación de *local-path-provisioner* se crea un *storageClass* llamado *local-path* el cual permite crear *PersistentVolumes* dinámicamente cuando se realizan las solicitudes mediante un *PersistentVolumeClaim*. Estos *PersistentVolumes* creados son de almacenamiento local al clúster. Una

limitación existente es que en la versión actual del *local-path-provisioner* no permite poner límites a la capacidad del *volume*.

Instalación de la versión v0.0.28 de *local-path-provisioner*:

```
kubectl apply -f https://raw.githubusercontent.com/rancher/local-path-provisioner/v0.0.28/deploy/local-path-storage.yaml
```

Fragmento 41: Instalación de local-path-provisioner v0.0.28

```
admin@ip-172-31-18-207:~$ kubectl -n local-path-storage get all
NAME                                      READY   STATUS    RESTARTS   AGE
pod/local-path-provisioner-567b8559b8-r8zht   1/1     Running   0          47m

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/local-path-provisioner   1/1     1            1           47m

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/local-path-provisioner-567b8559b8   1         1         1       47m
```

Fragmento 42: Despliegue de los recursos del local-path-provisioner

Otra mejora que se incorporó al clúster es la inclusión del *Helm Chart kube-prometheus-stack* para la monitorización del clúster.

El *Helm Chart kube-prometheus-stack* incorpora un servidor de métricas de *Prometheus* y *AlertManager* preconfigurado para obtener métricas relevantes de un clúster de *Kubernetes* como el consumo de *CPU*, *RAM* o recursos de red por parte de los nodos o de aplicaciones desplegadas en el clúster. También incorpora un servidor *Grafana* para mostrar las gráficas de todas las métricas obtenidas.

Para instanciar una *release* del *Chart kube-prometheus-stack*, añadiremos el repositorio de *Helm* de *prometheus-community* y modificaremos el fichero *values.yaml* para añadir nuestra configuración.

Para el acceso a *Grafana*, el *Chart kube-prometheus-stack* permite el despliegue de un *Ingress*. Como se mencionó anteriormente, los *Ingress* requieren de un *Ingress Controller*. En este caso se ha seleccionado el *Ingress Controller* llamado *Ingress Nginx*, proporcionado por *Kubernetes*. La instalación se realiza con un *Helm Chart* también:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

helm repo update

helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx --namespace ingress-nginx --create-namespace
```

Fragmento 43: Instalación de Ingress Nginx

Tras la instalación de *Ingress Nginx* en su *Namespace ingress-nginx* y en conjunto con la instalación previa de *MetalLB*, podremos instanciar el *Ingress* necesario para acceder a *Grafana*.

La instalación de *kube-prometheus-stack* y aplicación del *values.yaml* se realiza así:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts


helm repo update


helm upgrade --install --namespace monitoring prometheus prometheus-community/kube-prometheus-stack -f values.yaml --create-namespace
```

Fragmento 44: Instalación de kube-prometheus-stack y aplicación de values.yaml

El contenido del fichero *values.yaml* se encuentra en el anexo sección 7.4, así como los *Secrets* configurados para el acceso a *Grafana* en el anexo sección 7.5. Tras la instalación el acceso a *Grafana* y los datos recolectados por *Prometheus* se pueden observar desde la interfaz *web*.

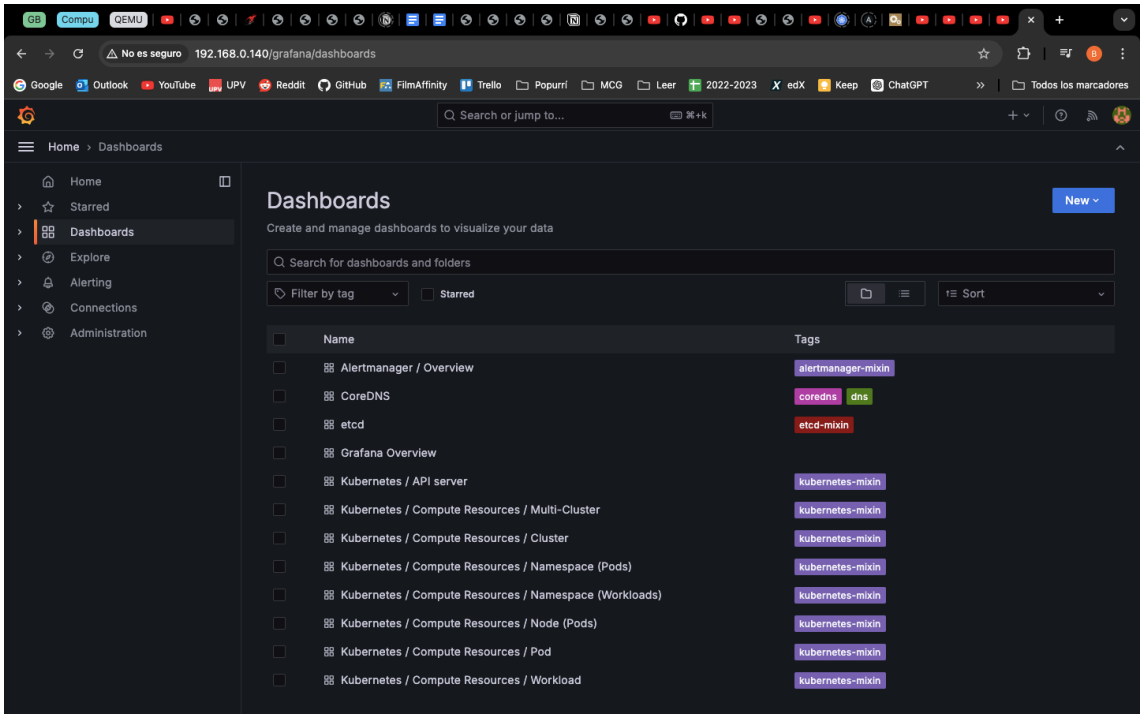

Figura 10: Acceso principal a Grafana

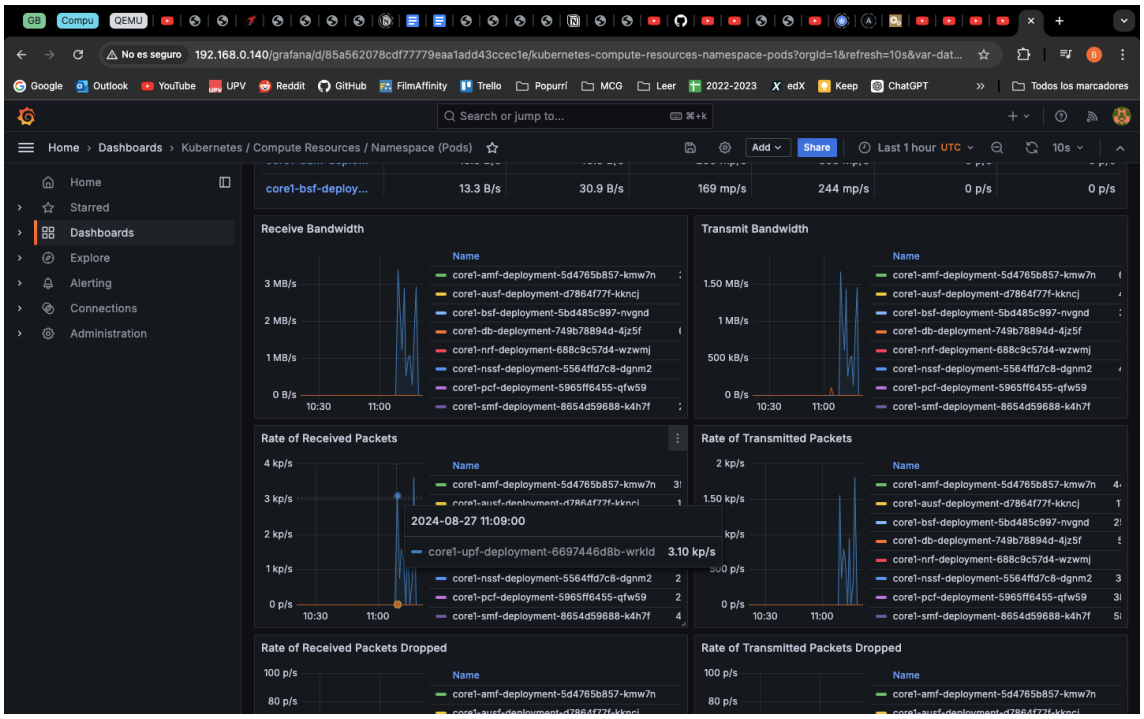

Figura 11: Información de red del UPF con kube-prometheus-stack

3.8 Verificación del funcionamiento

Para verificar el correcto funcionamiento de todos los componentes desplegados hasta ahora se decidió utilizar un simulador de *gNBs* y *UEs* en la cuarta *Raspberry Pi*, hasta ahora sin uso.

Los *gNBs* en 5G son las estaciones base utilizadas para permitir las conexiones de los móviles o *UEs*. Con *UERANSIM* [8] podemos simular conexiones para verificar el correcto funcionamiento del núcleo de red 5G desplegado.

Tras la instalación de *UERANSIM* y la configuración de un *gNB* y un *UE* con los mismos parámetros que los existentes en el despliegue del núcleo de red 5G, se puede verificar el correcto funcionamiento de los diferentes componentes de la red. Una prueba simple es conectar el *gNB* a la red, conectar un *UE* y verificar que el tráfico es capaz de llegar a Internet a través de su conexión por la red móvil. Se puede consultar la configuración del *gNB* y *UE* desplegados en *UERANSIM* en la sección del anexo 7.6.

3.9 Instalación de Kubernetes en AWS

[8] El proceso de conexión del simulador UERANSIM y los detalles se especifican en el capítulo 4: Validación de la memoria.

Tras la verificación del despliegue *on-premises* se comenzó a trabajar utilizando el proveedor de *cloud* público. Como proveedor de *cloud* público se ha escogido *Amazon Web Services*.

Tras crear la cuenta raíz de *AWS* lo siguiente fue crear un usuario de *IAM* con privilegios limitados. A través de este usuario se creó un usuario de *IAM* con acceso a la consola para poder utilizar aws cli. Con el usuario de IAM obtendremos el access key y el *secret access key* necesarios para utilizar *aws cli*. Estos fueron guardados en un archivo *credentials* dentro del directorio *aws* para ser utilizados desde el terminal.

Para poder acceder a las instancias creadas en *AWS* se creó un *key pair* para el acceso mediante *SSH*. También se creó un *security group* para administrar las conexiones de las instancias, únicamente permitiendo el tráfico de entrada *SSH* desde la *IP WAN* del *router* de casa y permitiendo el tráfico de salida desde las instancias a Internet para poder instalar lo necesario.

El *VPC* utilizado fue el creado por defecto por *AWS*, utilizando la subred privada 172.31.0.0/16 en la *región eu-north-1* situada en Estocolmo.

El sistema operativo seleccionado para los equipos de la infraestructura *cloud* es *Debian*, una distribución del sistema operativo *GNU/Linux*, en concreto la versión 12 también llamada *Bookworm*.

Se comenzaron a realizar pequeñas pruebas utilizando instancias basadas en la *AMI* de Debian 12 con arquitectura x86-64 y tamaño *t3.micro* para poder ser utilizadas dentro de la capa gratuita.

Las primeras pruebas consistieron en la creación de instancias asignadas al *security group* definido previamente y con el *key pair* creado.

```
aws ec2 run-instances --image-id ami-0b27735385ddf20e8 --count 1 --instance-
type t3.micro --key-name Borjis-AWS --security-group-ids sg-0a5a9fb43ef7fd5fd
```

Fragmento 45: Ejemplo de creación de una AMI de tipo t3.micro utilizando aws cli

Se realizaron más pruebas con el objetivo de obtener los comandos necesarios para obtener las *IPs públicas* de las instancias, así como los identificadores de las instancias para poder lanzar el comando **terminate-instances** para eliminarlas. El listado de los comandos utilizados con *aws cli* se puede encontrar en la sección del anexo 7.7.

Tras familiarizarse con *aws cli*, se utilizó el *module* de *Ansible* **Amazon.aws** para crear *plays* lanzados con el comando *ansible-playbook* que permitieran la creación y la destrucción de las instancias que serán utilizadas para el clúster de *Kubernetes* en *AWS*. Estos plays se pueden encontrar en la sección del anexo 7.8.

Tras la creación de las instancias, se modificó el *inventory* de *Ansible* para apuntar a estas nuevas instancias creadas en el *cloud* y se ejecutaron los *roles* creados para la instalación de *containerd*, *kubernetes* y *metallb*.

Una de las principales diferencias entre una instancia de *AWS* basada en *Debian* y una máquina *on-premises* basada en *Debian* es que las instancias de *AWS* tienen repositorios *apt* diferentes, cambiando el nombre de algunos de los paquetes a instalar o incluso no ofreciendo algunos paquetes. Esta diferencia nos llevó a modificar algunos *tasks* existentes en los *roles* para poder instalar todo.

Un ejemplo que muestra las diferencias claramente fue añadiendo la clave *GPG* para los repositorios de *apt*, tanto en el *rol containerd* como en el *rol kubernetes*. Previamente se hacía uso del *module* **ansible.builtin.apt_key** que requería del paquete *apt* **gnupg2** instalado. Este paquete no se encuentra con el mismo nombre en las instancias de *AWS* por lo que se migró la *task* a utilizar **ansible.builtin.get_url** para descargar la clave *GPG*.

```
- name: Add Docker's official GPG key
  become: yes
  ansible.builtin.apt_key:
    id: 9DC858229FC7DD38854AE2D88D81803C0EBFCD88
    url: "https://download.docker.com/linux/{{ ansible_distribution | lower }}/gpg"
    keyring: "{{ apt_keyrings_dir }}/docker.gpg"
    state: present
```

Fragmento 46: Versión del task utilizando ansible.builtin.apt_key

```
- name: Add Docker's official GPG key
  become: yes
  ansible.builtin.get_url:
    url: "https://download.docker.com/linux/{{ ansible_distribution | lower }}/gpg"
    dest: "{{ apt_keyrings_dir }}/docker.asc"
    mode: "0644"
    force: true
```

Fragmento 47: Versión del task utilizando ansible.builtin.get_url

De esta manera garantizamos que las tasks funcionan tanto en el entorno *on-premises* como en *cloud*.

La siguiente modificación fue utilizar instancias *t3.small* en vez de las *t3.micro*, puesto que *kubeadm* requiere de al menos 2 GB de memoria *RAM* y las instancias *t3.micro* solo tienen 1 GB de memoria *RAM* y no era posible su instalación.

Para la instalación del clúster de *Kubernetes* en el *cloud* público utilizaremos instancias *t3.small* de *AWS*, las cuales tienen las siguientes características *hardware* [44]:

- 2 *vCPU* de 64 bits *x86-64* al 20% de 3,1 GHz
- 2 GB de memoria *RAM LPDDR4-3200 SDRAM*
- Interfaz de red que soporta ráfagas de 5 Gbps
- *EBS* con 10 GB de almacenamiento

Tras superar los errores debido a la falta de recursos de las instancias *t3.micro*, comenzaron a surgir otros problemas existentes con el tipo de despliegue.

El *security group* de *AWS* tuvo que ser configurado iterativamente, comprobando paso a paso el uso de los diferentes protocolos. En el despliegue *on-premises*, como se comentó, no había ninguna clase de *firewall* o elemento de red que pudiera bloquear las comunicaciones. En el caso de *AWS*, no solo hacemos uso de los *security groups* para habilitar el tipo de tráfico que dejamos salir y entrar a nuestras instancias, sino que no tenemos control de lo que hay realmente desplegado en el *VPC*, es parte es la infraestructura de *AWS*.

El despliegue realizado con los *roles de Ansible* con los parámetros utilizados para el despliegue *on-premises* no funcionaba, la configuración de *Calico* no llegaba a desplegarse correctamente. Se identificó la causa del problema en el protocolo *BGP*.

Por alguna razón los recursos desplegados por *Calico* no eran capaces de establecer una conexión *BGP* entre ellos. Se probó a abrir el puerto relacionado con *BGP* en el *VPC* sin ningún éxito. En definitiva, se optó por la configuración de *Calico* en modo *VXLAN* en vez de en modo *BGP*.

En vez de la configuración utilizada *on-premises*:

```
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  calicoNetwork:
    ipPools:
    - name: default-ipv4-ippool
      blockSize: 26
      cidr: 10.13.0.0/24
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()
```

Fragmento 48: Configuración Calico en la infraestructura on-premises

Se utilizó la siguiente configuración para el *cloud*:

```
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  calicoNetwork:
    bgp: Disabled
    ipPools:
      - cidr: 10.13.0.0/24
        encapsulation: VXLAN
```

Fragmento 49: Configuración Calico en la infraestructura cloud

En el caso del despliegue de *MetalLB*, el problema era evidente desde el principio. Según la documentación de *MetalLB*, este no funciona en entornos *cloud* [45] y recomienda el uso de los balanceadores de carga existentes en estas plataformas.

Se decidió probar a instalar *MetalLB* y comprobarlo. Al configurar *MetalLB* en modo capa dos, no tendríamos más problemas con *BGP*. Se pudo desplegar todo y el único problema existente, se pudo solventar añadiendo rutas a las direcciones del balanceador de carga a través de los nodos del clúster de *Kubernetes*. Al parecer, el mecanismo por el que *MetalLB* retransmite las direcciones que usa no funciona correctamente, por lo que añadiendo rutas estáticas funciona correctamente.

Una configuración necesaria para hacer funcionar *MetalLB* en las instancias de *AWS* fue deshabilitar el *source/dest check* que realizan las instancias. Con el *source/dest check* activo la instancia comprueba que las direcciones *IP* de origen y destino coinciden con las propias antes de recibir y enviar tráfico. Haciendo uso de *MetalLB* en el que configuramos nuevas direcciones *IP* a las que la instancia ha de responder se ha de deshabilitar para permitir este tipo de tráfico.

Con todo lo anterior solventado, se procedió a verificar el correcto funcionamiento de todo lo instalado anteriormente utilizando el despliegue del núcleo de red 5G y el simulador *UERANSIM* como se hizo con el despliegue *on-premises*.

Con todas las comprobaciones realizadas se puede pasar a la validación y comparación de ambos despliegues en el siguiente apartado de la memoria.

4. Validación

La validación de la aplicación desplegada se llevó a cabo realizando pruebas de ancho de banda, tanto en el despliegue *on-premises* como en el *cloud*.

Las pruebas de ancho de banda se realizaron utilizando la conexión proporcionada por el núcleo de red 5G a la herramienta de simulación *UERANSIM*, siguiendo los pasos detallados a continuación.

Estas pruebas requieren de un clúster de Kubernetes en el que desplegar el núcleo de red 5G utilizando el *Helm Chart open5gs*. Tras este despliegue, se añade un usuario a la base de datos del núcleo de red 5G, usuario que utilizaremos para testear la conexión.

En una máquina con acceso a la subred del clúster de Kubernetes (la *Raspberry Pi 3 Model B* en el caso del despliegue *on-premises* y una instancia *t3.micro* en el caso del despliegue *cloud*) y utilizando los mismos parámetros que en el núcleo de red 5G, desplegaremos un *gNB* simulado utilizando la herramienta *UERANSIM*. Este *gNB* simulado proporcionará el punto de acceso para nuestro usuario.

En la misma máquina en la que se ha desplegado el *gNB* y utilizando los parámetros configurados en la base de datos del núcleo de red 5G, desplegaremos un *UE* simulado utilizando *UERANSIM*. Este usuario será capaz de registrarse y autenticarse en el núcleo de red 5G y recibirá los parámetros necesarios para establecer una conexión y tener acceso a Internet.

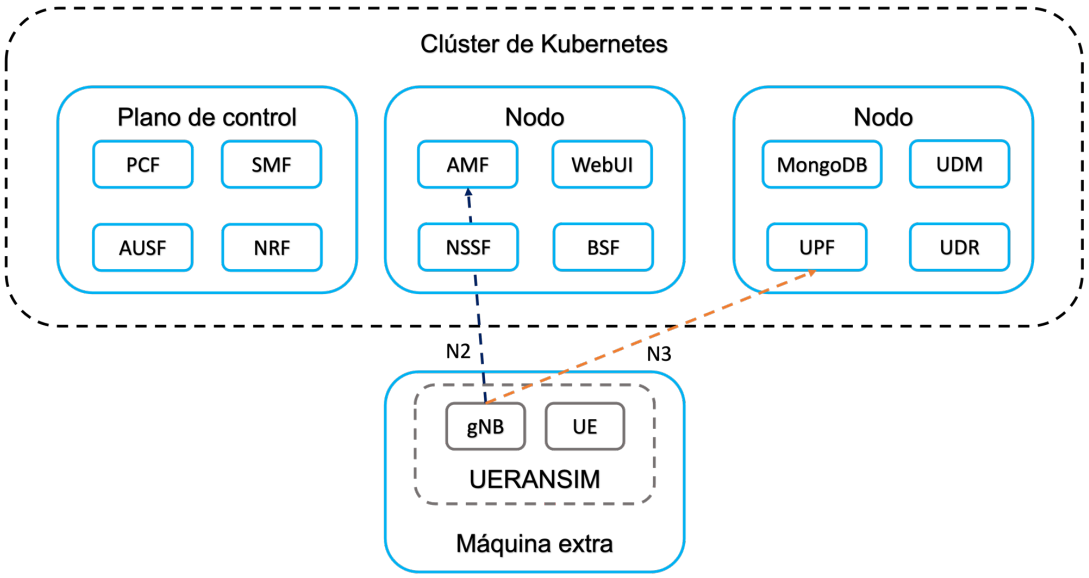

Figura 12: Diagrama de conexión con UERANSIM

Tras una conexión exitosa, el simulador creará un interfaz *TUN* en la máquina en la que se encuentra el *UE* simulado. El tráfico cursado por este interfaz será enviado del *UE* al *gNB* simulando la conexión que en un contexto real sería inalámbrica y el *gNB* enviará este tráfico al *UPF*, el cual será el encargado de cursar el tráfico a su destino.

En la verificación, esta conexión es la que se ha utilizado para realizar la comprobación de que todo lo desplegado funciona correctamente, utilizando la herramienta *ping* y alcanzando un servidor accesible desde Internet, en nuestro caso el 8.8.8.8, servidor *DNS* de *Google*.

Para esta comparativa se tienen en cuenta las diferencias existentes entre la infraestructura de cada despliegue. Testeando las infraestructuras se pudo observar una diferencia significativa en el tiempo de ida y vuelta (*RTT*) de las peticiones *ping* al servidor 8.8.8.8 de *Google*. Cuando realicemos comparativas utilizando este servidor, la medida importante será la diferencia entre el *RTT* obtenido por el *ping* realizado por la máquina y el *RTT* obtenido por el *ping* realizado a través de la red móvil, lo que nos dará el retardo en atravesar la red y nos permitirá eliminar el retardo desde nuestra máquina al servidor 8.8.8.8.

```
borjis@pi3:~ $ ping -I uesimtun0 8.8.8.8
PING 8.8.8.8 (8.8.8.8) from 10.45.0.5 uesimtun0: 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=11.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=117 time=10.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=117 time=8.94 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=117 time=8.99 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=117 time=9.27 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=117 time=8.86 ms
^C
--- 8.8.8.8 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5007ms
rtt min/avg/max/mdev = 8.856/9.576/11.066/0.831 ms
```

Fragmento 50: Ping a través de la red 5G on-premises

```
borjis@pi3:~/UERANSIM/build $ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=119 time=8.70 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=119 time=7.34 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=119 time=7.15 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=119 time=7.13 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=119 time=7.16 ms
^C
```

```
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 7.134/7.497/8.699/0.605 ms
```

Fragmento 51: Ping a través de la máquina on-premises

```
admin@ip-172-31-16-243:~$ ping -I uesimtun0 8.8.8.8
PING 8.8.8.8 (8.8.8.8) from 10.45.0.2 uesimtun0: 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=53 time=5.13 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=53 time=5.41 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=53 time=5.56 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=53 time=5.00 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=53 time=4.91 ms
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 4.911/5.202/5.559/0.245 ms
```

Fragmento 52: Ping a través de la red 5G cloud

```
admin@ip-172-31-16-243:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=55 time=3.66 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=55 time=3.69 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=55 time=3.67 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=55 time=3.65 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=55 time=3.64 ms
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 3.638/3.660/3.688/0.016 ms
```

Fragmento 53: Ping a través de la instancia cloud

Se pueden apreciar las diferencias de aproximadamente alrededor de 1,5 ms en el tiempo de ida y vuelta obtenidas tras hacer *ping* desde el interfaz principal de la máquina y desde el interfaz *TUN* creado por el simulador. Estas diferencias se deben al retardo de atravesar los diferentes componentes de la red 5G para salir a Internet.

Tras la validación, se realizaron unas pruebas de tráfico para medir el ancho de banda máximo aceptado por el núcleo de red 5G para un usuario, para estas pruebas se utilizó la herramienta *iPerf3*.

También se realizaron pruebas de tráfico utilizando las máquinas desplegadas (sin involucrar el núcleo de red 5G), pruebas de *CPU* y de memoria *RAM* para realizar la comparativa de las infraestructuras.

4.1 Pruebas de red con iPerf3

Utilizaremos *iPerf3* con el interfaz *TUN* creado para el usuario para medir el tráfico máximo que la red es capaz de soportar minimizando las pérdidas para un único usuario. Para evitar las diferencias en cuanto a la localización de los despliegues, como ocurre cuando utilizamos el servidor de *Google* para el *ping*, se utilizó un servidor público situado en Francia para las pruebas de ancho de banda de la red. Se utilizó iPerf para estas pruebas realizando medidas tanto con el protocolo *TCP* como *UDP*.

Las pruebas realizadas en UDP nos reportan los siguientes resultados:

```
borjis@pi3:~ $ iperf3 -c ping.online.net -p 5203 -u -b 70M --bind 10.45.0.7
Connecting to host ping.online.net, port 5203
[  5] local 10.45.0.7 port 48014 connected to 51.158.1.21 port 5203
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec  8.34 MBytes  69.9 Mbits/sec  6486
[  5]   1.00-2.00   sec  8.35 MBytes  70.0 Mbits/sec  6493
[  5]   2.00-3.00   sec  8.34 MBytes  70.0 Mbits/sec  6491
[  5]   3.00-4.00   sec  8.34 MBytes  70.0 Mbits/sec  6490
[  5]   4.00-5.00   sec  8.34 MBytes  70.0 Mbits/sec  6487
[  5]   5.00-6.00   sec  8.35 MBytes  70.0 Mbits/sec  6494
[  5]   6.00-7.00   sec  8.35 MBytes  70.1 Mbits/sec  6496
[  5]   7.00-8.00   sec  8.34 MBytes  70.0 Mbits/sec  6488
[  5]   8.00-9.00   sec  8.34 MBytes  70.0 Mbits/sec  6491
[  5]   9.00-10.00  sec  8.34 MBytes  70.0 Mbits/sec  6487
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-10.00  sec  83.4 MBytes  70.0 Mbits/sec  0.000 ms  0/64903 (0%)  sender
[  5]   0.00-10.44  sec  79.8 MBytes  64.1 Mbits/sec  0.328 ms  2807/64903 (4.3%)  receiver

iperf Done.
```

Fragmento 54: iPerf3 a través de la red 5G on-premises (Raspberry Pi 3 Model B)

```
borjis@pi3:~ $ iperf3 -c ping.online.net -u -b 300M
Connecting to host ping.online.net, port 5201
[  5] local 192.168.0.109 port 34615 connected to 51.158.1.21 port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec  11.5 MBytes  96.2 Mbits/sec  8307
[  5]   1.00-2.00   sec  11.4 MBytes  95.6 Mbits/sec  8257
[  5]   2.00-3.00   sec  11.4 MBytes  95.6 Mbits/sec  8256
[  5]   3.00-4.00   sec  11.4 MBytes  95.6 Mbits/sec  8256
```

```
[  5]   4.00-5.00   sec  11.4 MBytes  95.6 Mbits/sec  8256
[  5]   5.00-6.00   sec  11.4 MBytes  95.6 Mbits/sec  8256
[  5]   6.00-7.00   sec  11.4 MBytes  95.6 Mbits/sec  8256
[  5]   7.00-8.00   sec  11.4 MBytes  95.6 Mbits/sec  8256
[  5]   8.00-9.00   sec  11.4 MBytes  95.6 Mbits/sec  8256
[  5]   9.00-10.00  sec  11.4 MBytes  95.6 Mbits/sec  8256
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-10.00  sec   114 MBytes  95.7 Mbits/sec  0.000 ms  0/82612 (0%)  sender
[  5]   0.00-10.06  sec   114 MBytes  95.1 Mbits/sec  0.126 ms  0/82612 (0%)  receiver

iperf Done.
```

Fragmento 55: iPerf3 desde la máquina on-premises (Raspberry Pi 3 Model B)

```
admin@ip-172-31-16-243:~$ iperf3 -c ping.online.net --port 5202 --bind-dev uesimtun0 -u -b 100M
Connecting to host ping.online.net, port 5202
[  5] local 10.45.0.7 port 49083 connected to 51.158.1.21 port 5202
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec  11.9 MBytes  99.9 Mbits/sec  9265
[  5]   1.00-2.00   sec  11.9 MBytes   100 Mbits/sec  9273
[  5]   2.00-3.00   sec  11.9 MBytes   100 Mbits/sec  9277
[  5]   3.00-4.00   sec  11.9 MBytes   100 Mbits/sec  9269
[  5]   4.00-5.00   sec  11.9 MBytes   100 Mbits/sec  9273
[  5]   5.00-6.00   sec  11.9 MBytes   100 Mbits/sec  9273
[  5]   6.00-7.00   sec  11.9 MBytes   100 Mbits/sec  9273
[  5]   7.00-8.00   sec  11.9 MBytes   100 Mbits/sec  9273
[  5]   8.00-9.00   sec  11.9 MBytes   100 Mbits/sec  9273
[  5]   9.00-10.00  sec  11.9 MBytes   100 Mbits/sec  9273
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-10.00  sec   119 MBytes   100 Mbits/sec  0.000 ms  0/92722 (0%)  sender
[  5]   0.00-10.07  sec   119 MBytes  98.9 Mbits/sec  0.049 ms  310/92722 (0.33%)  receiver

iperf Done.
```

Fragmento 56: iPerf3 a través de la red 5G cloud (t3.micro)

```
admin@ip-172-31-25-109:~$ iperf3 -p 5202 -c ping.online.net -u -b 4G
Connecting to host ping.online.net, port 5202
[  5] local 172.31.25.109 port 41617 connected to 51.158.1.21 port 5202
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   399 MBytes  3.34 Gbits/sec  288690
[  5]   1.00-2.00   sec   395 MBytes  3.31 Gbits/sec  285721
[  5]   2.00-3.00   sec   308 MBytes  2.58 Gbits/sec  222929
[  5]   3.00-4.00   sec   290 MBytes  2.43 Gbits/sec  209875
[  5]   4.00-5.00   sec   313 MBytes  2.63 Gbits/sec  226906
[  5]   5.00-6.00   sec   344 MBytes  2.88 Gbits/sec  248927
```

```
[  5]   6.00-7.00   sec   374 MBytes  3.14 Gbits/sec  270941
[  5]   7.00-8.00   sec   363 MBytes  3.04 Gbits/sec  262662
[  5]   8.00-9.00   sec   402 MBytes  3.38 Gbits/sec  291391
[  5]   9.00-10.00  sec   366 MBytes  3.07 Gbits/sec  265353
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-10.00  sec  3.47 GBytes  2.98 Gbits/sec  0.000 ms  0/2573395 (0%)  sender
[  5]   0.00-10.07  sec  3.47 GBytes  2.96 Gbits/sec  0.001 ms  2451/2573395 (0.095%)  receiver

iperf Done.
```

Fragmento 57: iPerf3 a través de la instancia cloud (t3.micro)

Hay una gran diferencia entre el ancho de banda alcanzado por las máquinas y el alcanzado cuando se atraviesa el núcleo de red 5G, también existe una diferencia muy grande entre el ancho de banda alcanzado por la máquina *on-premises* y la instancia *cloud* sin usar la red 5G.

La diferencia entre la máquina *on-premises* y la instancia *cloud* se debe al uso de *Raspberry Pi 3 Model B*, este *SBC* en concreto tiene un interfaz de red inferior al de la propia *Raspberry Pi 4 Model B*. La *Raspberry Pi 3 Model B* cuenta con un interfaz que puede alcanzar 100 Mbit por segundo, mientras que la *Raspberry Pi 4 Model B* cuenta con un interfaz Gigabit *Ethernet*. Incluso en el caso de haber utilizado una *Raspberry Pi 4 Model B* para las pruebas de red, la instancia *cloud* es superior en este aspecto, llegando a alcanzar los casi 3 Gbit por segundo.

Pero la diferencia verdaderamente relevante para este proyecto y que ocurre en ambas infraestructuras es la diferencia entre el ancho de banda obtenido por la máquina/instancia frente al obtenido por el núcleo de red 5G desplegado. El cuello de botella se encuentra en el componente *UPF* desplegado en contenedores. Este componente tiene una limitación debido a su diseño, dado que utiliza un único hilo para realizar el enrutado de los paquetes. Esto supone una gran limitación en el tráfico cursado. *Open5GS* es una implementación muy interesante para probar funcionalidades, probar despliegues y tomar medidas, pero no es un núcleo de red preparado para un despliegue en producción. Este tipo de proyectos de código abierto en entornos de producción, suelen tener al *UPF* como el factor más limitante y las alternativas implicarían la instalación de otros *UPFs* que llevarían un trabajo de pruebas de interoperabilidad y serán contemplados como posible trabajo futuro.

Para realizar esta prueba, el punto de referencia para validarla fue obtener el mayor ancho de banda que el usuario podía cursar sin perder la conexión. Puesto que durante la validación probando anchos de banda más altos que los mostrados, los usuarios perdían la conexión con el núcleo de red 5G y tenían que volver a conectarse para poder funcionar.

4.2 Pruebas de rendimiento con sysbench

Para la comparación entre las infraestructuras utilizadas se realizó una prueba de *CPU* y de memoria *RAM* en las máquinas de cada infraestructura para así poder realizar una comparativa de coste y rendimiento.

Para las pruebas de *CPU* y memoria *RAM* se ha utilizado la herramienta *sysbench*.

```
borjis@k8s-node-2:~ $ sysbench cpu run --threads=4
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 4
Initializing random number generator from current time


Prime numbers limit: 10000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:  7064.71

General statistics:
    total time:                          10.0005s
    total number of events:              70675

Latency (ms):
         min:                                    0.56
         avg:                                    0.57
         max:                                    3.59
         95th percentile:                        0.57
         sum:                                39981.33

Threads fairness:
    events (avg/stddev):           17668.7500/4.44
    execution time (avg/stddev):   9.9953/0.00
```

Fragmento 58: Prueba sysbench CPU max threads on-premises (Raspberry Pi 4 Model B)

```
borjis@k8s-node-2:~ $ sysbench memory run
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
```

```
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 1KiB
  total size: 102400MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 24746666 (2473788.21 per second)

24166.67 MiB transferred (2415.81 MiB/sec)


General statistics:
    total time:                          10.0001s
    total number of events:              24746666

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                    0.11
         95th percentile:                        0.00
         sum:                                 4865.65

Threads fairness:
    events (avg/stddev):           24746666.0000/0.00
    execution time (avg/stddev):   4.8657/0.00
```

Fragmento 59: Prueba sysbench memoria RAM on-premises (Raspberry Pi 4 Model B)

```
admin@ip-172-31-16-243:~$ sysbench cpu run --threads=2
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Prime numbers limit: 10000

Initializing worker threads...
```

```
Threads started!

CPU speed:
    events per second:  1448.33

General statistics:
    total time:                          10.0008s
    total number of events:              14487

Latency (ms):
         min:                                    0.97
         avg:                                    1.38
         max:                                    9.47
         95th percentile:                        1.93
         sum:                                19988.12

Threads fairness:
    events (avg/stddev):           7243.5000/0.50
    execution time (avg/stddev):   9.9941/0.00
```

Fragmento 60: Prueba sysbench CPU max threads cloud (t3.micro)

```
admin@ip-172-31-16-243:~$ sysbench memory run
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 1KiB
  total size: 102400MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 47119349 (4711055.03 per second)

46014.99 MiB transferred (4600.64 MiB/sec)


General statistics:
```

```
    total time:                          10.0001s
    total number of events:              47119349

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                    9.93
         95th percentile:                        0.00
         sum:                                 4645.19

Threads fairness:
    events (avg/stddev):           47119349.0000/0.00
    execution time (avg/stddev):   4.6452/0.00
```

Fragmento 61: Prueba sysbench memoria RAM cloud (t3.micro)

En la sección del anexo 7.9 se pueden ver más pruebas realizadas utilizando un único *thread* para ambas infraestructuras.

De estas pruebas se puede ver que las CPUs de las *Raspberry Pi* son superiores tanto en número de núcleos como el número de eventos que puede procesar cada *thread*. La *Raspberry Pi 4 Model B* obtiene 1766 eventos por segundo por cada *thread* mientras que la instancia *t3.micro* obtiene 826 eventos por segundo. Cuando comparamos con el número de *threads* máximo la diferencia se hace aún más notable, llegando la *Raspberry Pi 4 Model B* a 7064 eventos por segundo frente a los 1448 eventos por segundo de la instancia *t3.micro*.

En cuanto a memoria *RAM* las *Raspberry Pi 4 Model B* tienen una mayor cantidad con sus 4 GB pero obtienen una velocidad menor con 2415 MiB por segundo, frente a los 2 GB de *RAM* de las instancias *t3.micro* pero con una velocidad de 4600 MiB por segundo.

Se evitaron las pruebas de almacenamiento puesto que ambos despliegues podrían ser mejorados fácilmente en este aspecto, utilizando *SSD* en el despliegue *on-premises* y utilizando los volúmenes optimizados para el almacenamiento en *AWS*.

5. Conclusiones y Trabajos Futuros

Tras la comparativa realizada se puede ver que en cuanto a *hardware* el rendimiento se divide por categorías donde comparando los modelos seleccionados obtenemos unas *Raspberry Pi 4 Model B* con unas especificaciones mejores en cuanto a *CPU* y una mayor memoria *RAM*, frente a una *RAM* con mayores velocidades y una red con un rendimiento mucho mayor en el caso de las instancias *t3.micro* de *AWS*.

Para el caso de la aplicación seleccionada para este TFM, el despliegue del núcleo de red 5G, el rendimiento del *UPF* ha sido el que ha marcado la prueba. Debido a las limitaciones del *UPF*, haciendo uso de un único *thread*, el uso de la red no ha impactado demasiado en las pruebas, lo que acorta las diferencias pese a las características de la red disponible en *AWS*. El núcleo de red 5G ha sido capaz de cursar 100 Mbit por segundo en el caso del despliegue *cloud* y 70 Mbit por segundo en el caso del despliegue *on-premises*.

En cuanto a la comparativa de costes, actualmente la *Raspberry Pi 4 Model B* con 4 GB de *RAM* cuesta alrededor de 65 € [46] mientras que en el entorno *cloud* de pago por uso, una instancia *t3.small* bajo demanda con 20 GB de almacenamiento en *EBS* cuesta alrededor de 13 € al mes, según *AWS pricing calculator* [47].

Con estos datos se puede ver que el precio del clúster *on-premises* (con una *Raspberry Pi 3 Model B* de 40 €) es de 195 € en total mientras que el clúster en el *cloud* costaría unos 39 € al mes (sin la instancia *t3.micro* de la capa gratuita). Esto implica que la compra de las *Raspberry Pis* estaría amortizada en unos 5 meses.

También hay que señalar que al igual que no se han utilizado las instancias más potentes en *AWS*, tampoco se han escogido los *SBCs* con mejores especificaciones con respecto al precio.

Se pueden encontrar alternativas para el clúster *on-premises* con un periodo de amortización más corto, aunque teniendo en cuenta que todas las mejoras de infraestructura llevadas a cabo precisarán de un equipo de trabajadores mayor y una inversión en *hardware* mayor también.

En cambio, si lo que se busca es tener un gran rendimiento disponible para nuestras aplicaciones sin la necesidad de tener un gran equipo para el

mantenimiento de la infraestructura, con la desventaja de que esto implica un pago recurrente por uso, se puede optar por el *cloud* público.

En el contexto del núcleo de red 5G desplegado y con las limitaciones existentes del *UPF*, el despliegue *on-premises* es una muy buena opción. El caso sería muy diferente realizando un despliegue con un *UPF* preparado para producción donde el entorno *cloud* tendría mucho que aportar en cuanto a sus recursos.

Para la continuación y evolución de las tareas realizadas en este TFM se listan los siguientes trabajos futuros:

- Soportar otras distribuciones *GNU/Linux* en los *roles de Ansible* para la instalación
- Utilización y optimización de un *UPF* preparado para producción
- Investigar el impacto en las métricas obtenidas utilizando un despliegue híbrido, con el plano de control en el *cloud* y el UPF en un servidor *on-premises*

6. Bibliografía

[1] IBM CP-40, "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/IBM_CP-40.

[2] Amazon, "Amazon Web Services," [Online]. Available: https://aws.amazon.com/.

[3] AWS S3, "Amazon Web Services," [Online]. Available: https://aws.amazon.com/es/about-aws/whats-new/2006/03/13/announcing-amazon-s3---simple-storage-service/.

[4] AWS EC2, "Amazon Web Services," [Online]. Available: https://aws.amazon.com/es/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/.

[5] Google, "Google Cloud Platform," [Online]. Available: https://cloud.google.com/.

[6] Microsoft, "Azure," [Online]. Available: https://azure.microsoft.com/.

[7] OpenStack, "OpenStack," [Online]. Available: https://www.openstack.org/.

[8] Docker, "Docker," [Online]. Available: https://www.docker.com/.

[9] Kubernetes, "Kubernetes," [Online]. Available: https://kubernetes.io/.

[10] iTEAM, "iTEAM UPV," [Online]. Available: https://www.iteam.upv.es/.

[11] Synergy Research Group, "Huge Cloud Market Sees a Strong Bounce in Growth Rate for the Second Consecutive Quarter," [Online]. Available: https://www.srgresearch.com/articles/huge-cloud-market-sees-a-strong-bounce-in-growth-rate-for-the-second-consecutive-quarter.

[12] Raspberry Pi, "Raspberry Pi," [Online]. Available: https://www.raspberrypi.com/.

[13] Orange Pi, "Orange Pi," [Online]. Available: http://www.orangepi.org/.

[14] Hardkernel, "ODROID by Hardkernel," [Online]. Available: https://www.hardkernel.com/.

[15] NVIDIA, "Nvidia Jetson," [Online]. Available: https://www.nvidia.com/es-

es/autonomous-machines/embedded-systems/.

[16] Ansible, "Ansible," [Online]. Available: https://www.ansible.com/.

[17] Salt, "Salt," [Online]. Available: https://saltproject.io/.

[18] Chef, "Chef," [Online]. Available: https://www.chef.io/.

[19] Puppet, "Puppet," [Online]. Available: https://www.puppet.com/.

[20] Wikipedia, "Kubernetes," [Online]. Available: https://en.wikipedia.org/wiki/Kubernetes.

[21] Kubernetes, "kubeadm," [Online]. Available: https://kubernetes.io/es/docs/reference/setup-tools/kubeadm/.

[22] Open5GS, "Open5GS," [Online]. Available: https://open5gs.org/.

[23] free5GC, "free5GC," [Online]. Available: https://free5gc.org/.

[24] OpenAirInterface, "OpenAirInterface," [Online]. Available: https://openairinterface.org/.

[25] Open Networking Foundation, "SD-Core," [Online]. Available: https://opennetworking.org/sd-core/.

[26] Spirent, "Landslide," [Online]. Available: https://www.spirent.com/products/core-network-test-5g-lte-ims-wifi-diameter-landslide.

[27] Keysight, "LoadCore," [Online]. Available: https://www.keysight.com/us/en/product/P8900S/loadcore-core-network-solutions.html.

[28] GitHub UERANSIM, "UERANSIM," [Online]. Available: https://github.com/aligungr/UERANSIM.

[29] GitHub PacketRusher, "PacketRusher," [Online]. Available: https://github.com/HewlettPackard/PacketRusher.

[30] docker-open5gs, "GitHub docker-open5gs," [Online]. Available: https://github.com/Borjis131/docker-open5gs.

[31] Kubernetes, "Installing kubeadm," [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[32] Raspberry Pi, "cmdline.txt," [Online]. Available: https://www.raspberrypi.com/documentation/computers/config_txt.html#cmdline.

[33] Wikimedia, "Packet flow in Netfilter and General Networking," [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg.

[34] Kubernetes, "systemd cgroup driver," [Online]. Available: https://kubernetes.io/docs/setup/production-environment/container-runtimes/#systemd-cgroup-driver.

[35] Tigera, "Calico," [Online]. Available: https://www.tigera.io/project-calico/.

[36] Medium, "Choosing a Container Network Interface Plugin in Kubernetes," [Online]. Available: https://overcast.blog/choosing-the-right-container-network-interface-plugin-in-kubernetes-45391c7d4cc8.

[37] Red Hat, "¿Qué es un operador de Kubernetes?," [Online]. Available: https://www.redhat.com/es/topics/containers/what-is-a-kubernetes-operator.

[38] MongoDB, "MongoDB," [Online]. Available: https://www.mongodb.com/.

[39] GiHub, "Imágenes Docker MongoDB ARMv8.0," [Online]. Available: https://github.com/themattman/mongodb-raspberrypi-docker.

[40] Kubernetes, "When to use a ReplicaSet," [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#when-to-use-a-replicaset.

[41] Kubernetes, "ConfigMaps," [Online]. Available: https://kubernetes.io/es/docs/concepts/configuration/configmap/.

[42] Wikipedia, "Stream Control Transmission Protocol," [Online]. Available: https://es.wikipedia.org/wiki/Stream_Control_Transmission_Protocol.

[43] GitHub, "Repositorio TFM," [Online]. Available: https://github.com/Borjis131/TFM.

[44] Amazon, "Instancias T3 de Amazon EC2," [Online]. Available: https://aws.amazon.com/es/ec2/instance-types/t3/.

[45] MetalLB, "Cloud Compatibility," [Online]. Available: https://metallb.universe.tf/installation/clouds/index.html.

[46] raspipc.es, "Raspberry Pi 4 Model B 4 GB," [Online]. Available: https://www.raspipc.es/index.php?ver=tienda&accion=verArticulo&idProducto=1752&src=raspberrypi.

[47] Amazon, "AWS Pricing Calculator," [Online]. Available: https://calculator.aws/#/.

[48] GitHub, "docker-open5gs Bakefile," [Online]. Available: https://github.com/Borjis131/docker-open5gs/blob/main/docker-bake.hcl.

[49] Kubernetes, "Taints and Tolerations," [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/.

7. Anexo

7.1 Creación del builder en Docker buildx

```
docker buildx create --name example --driver docker-container --use
--bootstrap
```

7.2 Archivo Bakefile

```
variable "OPEN5GS_VERSION" {
  default = "v2.7.2"
}

variable "UBUNTU_VERSION" {
  default = "jammy"
}

group "default" {
  targets = ["base-open5gs", "amf"]
}

target "base-open5gs" {
  context = "./images/base-open5gs"
  tags = ["base-open5gs:${OPEN5GS_VERSION}"]
  output = ["type=image"]
}

target "amf" {
  context = "./images/amf"
  contexts = {
    "base-open5gs:${OPEN5GS_VERSION}" = "target:base-open5gs"
  }
  tags = ["amf:${OPEN5GS_VERSION}"]
  output = ["type=image"]
}
```

Este ejemplo solo contiene un segmento. Archivo completo [48].

7.3 Comando script open5gs-dbctl

```
./open5gs-dbctl --db_uri=192.168.0.106:30007/open5gs
```

```
add_ue_with_slice 001011234567891 00000000000000000000000000000000
00000000000000000000000000000000 internet 1 000001
```

7.4 kube-prometheus-stack values.yaml

```
fullnameOverride: prometheus


alertmanager:
  fullnameOverride: alertmanager
  enabled: true
  ingress:
    enabled: false
  alertmanagerSpec:
    storage:
      volumeClaimTemplate:
        spec:
          storageClassName: local-path
          resources:
            requests:
              storage: 1Gi
          accessModes:
            - ReadWriteOnce


grafana:
  fullnameOverride: grafana
  enabled: true
  defaultDashboardsEnabled: true
  defaultDashboardsTimezone: utc
  ingress:
    enabled: true
    ingressClassName: nginx
    hosts: []
    path: /
  serviceMonitor:
    enabled: true
  admin:
    existingSecret: grafana-credentials
    userKey: grafana-user
    passwordKey: grafana-password
  grafana.ini:
    server:
      domain: "{{ if (and .Values.ingress.enabled .Values.ingress.hosts) }}{{
.Values.ingress.hosts | first }}{{ else }}''{{ end }}"
      root_url: http://localhost:3000/grafana
      serve_from_sub_path: true


kubeApiServer:
```

```
  enabled: true

kubelet:
  enabled: true

kubeControllerManager:
  enabled: true

coreDns:
  enabled: true

kubeDns:
  enabled: false

kubeEtcd:
  enabled: true

kubeScheduler:
  enabled: true
  endpoints:

kubeProxy:
  enabled: true
  endpoints:

kubeStateMetrics:
  enabled: true

kube-state-metrics:
  fullnameOverride: kube-state-metrics
  selfMonitor:
    enabled: true
  prometheus:
    monitor:
      enabled: true

nodeExporter:
  enabled: true

prometheus-node-exporter:
  fullnameOverride: node-exporter
  prometheus:
    monitor:
      enabled: true

prometheusOperator:
```

```
  enabled: true


prometheus:
  enabled: true
  prometheusSpec:
    replicas: 1
    ruleSelectorNilUsesHelmValues: false
    serviceMonitorSelectorNilUsesHelmValues: false
    podMonitorSelectorNilUsesHelmValues: false
    probeSelectorNilUsesHelmValues: false
    retention: 2d
    storageSpec:
      volumeClaimTemplate:
        spec:
          storageClassName: local-path
          resources:
            requests:
              storage: 4Gi
          accessModes:
            - ReadWriteOnce

thanosRuler:
  enabled: false
```

7.5 Grafana Secrets

```
echo -n "admin" > ./grafana-user
echo -n "grafana" > ./grafana-password


kubectl create secret generic grafana-credentials --from-
file=./grafana-user --from-file=grafana-password -n monitoring
```

7.6 Configuración de UERANSIM

```
mcc: '001'
mnc: '01'

nci: '0x000000010'
idLength: 32
tac: 1

linkIp: gnb.ueransim.org
ngapIp: gnb.ueransim.org
gtpIp: gnb.ueransim.org

amfConfigs:
```

```
  - address: amf.open5gs.org
    port: 38412

slices:
  - sst: 1
    sd: 000001

ignoreStreamIds: true
```

Fichero de configuración del gNB para UERANSIM

```
supi: 'imsi-001011234567891'
mcc: '001'
mnc: '01'

key: '00000000000000000000000000000000'
op: '00000000000000000000000000000000'
opType: 'OPC'
amf: '8000'
imei: '356938035643803'
imeiSv: '4370816125816151'

gnbSearchList:
  - gnb.ueransim.org

uacAic:
  mps: false
  mcs: false

uacAcc:
  normalClass: 0
  class11: false
  class12: false
  class13: false
  class14: false
  class15: false

sessions:
  - type: 'IPv4'
    apn: 'internet'
    slice:
      sst: 1
      sd: 000001

configured-nssai:
  - sst: 1
    sd: 000001
```

```
default-nssai:
  - sst: 1
    sd: 000001

integrity:
  IA1: true
  IA2: true
  IA3: true

ciphering:
  EA1: true
  EA2: true
  EA3: true

integrityMaxRate:
  uplink: 'full'
  downlink: 'full'
```

Fichero de configuración del UE para UERANSIM

7.7 Comandos de aws cli

```
# Obtén los instance ids
aws ec2 describe-instances --query "Reservations[].Instances[].InstanceId"

# Obtén el nombre y la IP pública
aws ec2 describe-instances \
--filters "Name=instance-state-name,Values=running" \
--query "Reservations[].Instances[].[PublicIpAddress, Tags[].Value]"

# Elimina una instancia con <id>
aws ec2 terminate-instances --instance-ids <id>

# Elimina el source/dst check de una instancia con <id>
aws ec2 modify-instance-attribute --no-source-dest-check --instance-id <id>
```

7.8 Ansible Amazon.aws module plays

```
---
- hosts: localhost
  gather_facts: False
  vars_files:
    - ../vars/aws.yaml
  module_defaults:
```

```
    group/aws:
      aws_access_key: "{{ aws_access_key }}"
      aws_secret_key: "{{ aws_secret_key }}"
      region: "eu-north-1"

  tasks:
    - name: "Provision the EC2 instances for the k8s cluster"
      amazon.aws.ec2_instance:
        name: "{{ item }}"
        state: running
        key_name: "Borjis-AWS"
        instance_type: t3.small
        security_groups:
          - sg-xxxxxxxxxxxxxxxxx
        network:
          assign_public_ip: true
          delete_on_termination: true
        volumes:
          - device_name: /dev/xvda
            ebs:
              volume_size: 10
        image_id: ami-0b27735385ddf20e8
      register: result
      with_items:
        - aws-k8s-master
        - aws-k8s-node-1
        - aws-k8s-node-2
```

Ansible play para crear el clúster en AWS

```
---
- hosts: localhost
  gather_facts: False
  vars_files:
    - ../vars/aws.yaml
  module_defaults:
    group/aws:
      aws_access_key: "{{ aws_access_key }}"
      aws_secret_key: "{{ aws_secret_key }}"
      region: "eu-north-1"

  tasks:
    - name: "Delete the EC2 instances for the k8s cluster"
      amazon.aws.ec2_instance:
        name: "{{ item}}"
        state: terminated
      register: result
```

```
      with_items:
        - aws-k8s-master
        - aws-k8s-node-1
        - aws-k8s-node-2
```

Ansible play para destruir el clúster en AWS

7.9 Resultados sysbench CPU 1 thread

```
borjis@k8s-node-2:~ $ sysbench cpu run
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Prime numbers limit: 10000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:  1766.69

General statistics:
    total time:                          10.0006s
    total number of events:              17674

Latency (ms):
         min:                                    0.56
         avg:                                    0.57
         max:                                    1.09
         95th percentile:                        0.57
         sum:                                 9995.47

Threads fairness:
    events (avg/stddev):           17674.0000/0.00
    execution time (avg/stddev):   9.9955/0.00
```

Prueba sysbench CPU 1 thread on-premises (Raspberry Pi 4 Model B)

```
admin@ip-172-31-25-109:~$ sysbench cpu run
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
```

```
Initializing random number generator from current time


Prime numbers limit: 10000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:   826.12

General statistics:
    total time:                          10.0003s
    total number of events:              8263

Latency (ms):
         min:                                    0.97
         avg:                                    1.21
         max:                                   10.67
         95th percentile:                        1.79
         sum:                                 9991.29

Threads fairness:
    events (avg/stddev):           8263.0000/0.00
    execution time (avg/stddev):   9.9913/0.00
```

*Prueba sysbench CPU 1 thread cloud (t3 micro)*Resultados sysbench CPU 1 thread

7.10 Relación con los Objetivos de Desarrollo Sostenibles (ODS)

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Proceede**
ODS 1. **Fin de la pobreza.**				**X**
ODS 2. **Hambre cero.**				**X**
ODS 3. **Salud y bienestar.**				**X**
ODS 4. **Educación de calidad.**				**X**
ODS 5. **Igualdad de género.**				**X**
ODS 6. **Agua limpia y saneamiento.**				**X**
ODS 7. **Energía asequible y no contaminante.**				**X**
ODS 8. **Trabajo decente y crecimiento económico.**				**X**

ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.			X	
ODS 11. **Ciudades y comunidades sostenibles**.			X	
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

El presente TFM analiza y utiliza tecnologías del estado del arte que permiten una innovación y una mejora de las infraestructuras de las telecomunicaciones con el impacto que eso supone en la sociedad. Mejorando las redes que interconexión nacen nuevos casos de uso que mejoran nuestra vida en el día a día, como puede ser la creación de ciudades con una mayor recolección de datos para el análisis climatológico a través de sensores conectados a las redes móviles.

Algunas de las mejoras presentadas también impactan el ámbito económico de estas redes, desde su coste inicial como su coste de operación, reduciendo las desigualdades económicas a la hora de desplegar redes como puede ser en ámbitos rurales donde las infraestructuras se resienten por el bajo número de usuarios. Reduciendo costes acercamos la tecnología a esos ámbitos.

Algunas de las mejoras en la reducción de costes también se deben a una mejor utilización de los recursos existentes, lo cual nos lleva a la creación de comunidades más sostenibles e impacta en el clima gracias a evitar el desperdicio recursos.