Análisis del rendimiento del Subsistema Multimedia IP basado en NFV considerando microservicios

Trabajo de Grado

Juan Sebastian Orduz Vidal

Gabriel David Orozco Urrutia

Director: PhD. Oscar Mauricio Caicedo Rendón
Codirector: Msc. Carlos Hernan Tobar Arteaga

Departamento de Telemática
Facultad de Ingeniería Electrónica y Telecomunicaciones
Universidad del Cauca
Popayán, Cauca, 2019

Análisis del rendimiento del Subsistema Multimedia IP basado en NFV considerando microservicios

Juan Sebastian Orduz Vidal
Gabriel David Orozco Urrutia

Trabajo de grado presentado a la Facultad de Ingeniería
Electrónica y Telecomunicaciones de la
Universidad del Cauca para obtener el título de:
Ingeniero en Electrónica y Telecomunicaciones

Director: PhD. Oscar Mauricio Caicedo Rendón
Codirector: Msc. Carlos Hernan Tobar Arteaga

Departamento de Telemática
Facultad de Ingeniería Electrónica y Telecomunicaciones
Universidad del Cauca
Popayán, Cauca, 2019

Agradecimientos

En primera medida es imperativo agradecer a Dios, a nuestros familiares por su constante apoyo y entrega a lo largo del desarrollo de nuestra vida personal. A nuestros amigos y colegas del programa de Ingeniería Electrónica y Telecomunicaciones por su importante compañía durante la duración de este proyecto de investigación. A nuestro director Oscar Mauricio Caicedo y a nuestro codirector Carlos Tobar, quienes fueron nuestros guías y mentores a lo largo de este proyecto, y que junto al semillero de investigación COMSOCAUCA nos brindaron de todas las bases investigativas imprescindibles para su culminación. Mencionar además, a todo el equipo del laboratorio Telco 2.0 por proveernos con los recursos necesarios para el desarrollo del proyecto. Finalmente, queremos resaltar lo orgullosos que nos sentimos por haber realizado nuestro primer paso como ingenieros en la ilustre Universidad del Cauca, que nos proveyó de las mejores luces en nuestra formación.

Abstract

The steps towards all over IP have defined to the IMS (IP Multimedia Subsystem) as the de facto technology for end-to-end multimedia service provisioning in 5G. However, the unpredictable growth of users in 5G requires to improve IMS scalability to handle dynamic user traffic. Several works have addressed this issue by introducing auto-scaling mechanisms in vIMS (virtualized IMS) architectures. However, the current vIMS deployments use a monolithic design that does not allow a finer-scalability. In this monograph, we evaluate the performance of a microservice-based vIMS architecture called μvIMS, designed to provide finer-scalability, allowing more effective resources usage than regular monolithic design. To test our architecture, we evaluate μvIMS prototype regarding CPU usage, memory usage, SCR (Successful Call Rate), and latency metrics. Our test results reveal that μvIMS achieves a higher number of successful calls using the available resources effectively with a negligible latency increasing. Thus, we can state that divide the monolithic vIMS architecture in microservices enhances scalability to adapt to dynamic user traffic.

Resumen

Los pasos hacia todo sobre IP han definido a IMS (*IP Multimedia Subsystem*) como la tecnología de facto para el provisionamiento de servicios multimedia de extremo a extremo en 5G. Sin embargo, el crecimiento impredecible de usuarios en 5G requiere mejorar la escalabilidad de IMS para manejar el tráfico dinámico de estos usuarios. Varios trabajos han abordado este problema al introducir mecanismos de escalado automático en arquitecturas vIMS *(virtualized IMS*). Sin embargo, las implementaciones actuales de vIMS utilizan un diseño monolítico que no permite una escalabilidad fina. En esta monografía, se analiza el rendimiento de un vIMS basado en microservicios llamado μvIMS, diseñado para proporcionar una escalabilidad fina para permitir un uso más eficiente de los recursos que el diseño monolítico. Para probar esta arquitectura, se evaluó un prototipo de μvIMS con respecto al uso de la CPU, el uso de memoria, el SCR (*Successful Call Rate*) y la latencia. Los resultados de las evaluaciones revelan que μvIMS logra una mayor cantidad de llamadas exitosas al usar los recursos disponibles de manera más eficiente con un aumento de la latencia despreciable. Por lo tanto, se puede afirmar que dividir la arquitectura vIMS monolítica en microservicios mejora la escalabilidad, permitiendo adaptarse al tráfico dinámico de usuarios.

Contenido

Lista de Figuras

Lista de Tablas

Lista de Acrónimos

Acrónimo	Significado en inglés	Significado en español
AS	Aplication Server	Servidor de Aplicación
BM	Bare-Metal	Metal Desnudo
CPS	Calls Per Second	Llamadas Por Segundo
CPU	Central Processing Unit	Unidad de Procesamiento Central
CSCF	Call Session Control Function	Función de Control de Sesión de Llamada
CSP	Communications Service Provider	Proveedor de Servicios de Comunicaciones
CDF	Cumulative Distribution Function	Función de Distribución Acumulativa
CDR	Call Detail Record	Registro de Detalles de Llamada
ETSI	European Telecommunication Standards Institute	Instituto Europeo de Normas de Telecomunicaciones
GNF	Glasgow Network Functions	Función de Red Glasgow
HSS	Home Subscriber Server	Base de datos para los subscriptores
HTTP	Hypertext Transfer Protocol	Protocolo de Transferencia de Hipertexto
I-CSCF	Interrogating-CSCF	CSCF de Interrogación
IDS	Intrusion Detection System	Sistema de Detección de Intrusos
IMS	IP Multimedia Subsystem	Subsistema Multimedia IP
IoT	Internet of things	Internet de las Cosas
IFC	Initial Filter Criteria	Criterio de Filtrado Inicial
LxC	Linux Container	Contenedor Linux
MPS	Multimedia Priority Service	Servicio Prioritario Multimedia
MSA	Microservices Architecture	Arquitectura de Microservicios
NF	Network Function	Función de Red
NFV	Network Function Virtualization	Virtualización de Funciones de Red
NFVO	NFV Orchestrator	Orquestador NFV
NGN	Next Generation Networks	Redes de Nueva Generación
P-CSCF	Proxy-CSCF	CSCF Apoderado
QoS	Quality of Service	Calidad de Servicio
S-CSCF	Serving-CSCF	CSCF de Servicio
SCR	Successful Call Rate	Tasa de Llamadas Exitosas
SIP	Session Initiation Protocol	Protocolo de Inicio de Sesión
vIMS	virtualized IP Multimedia Subsystem	Subsistema Multimedia IP virtualizado
VIM	Virtualized Infrastructure Manager	Gestor de Infraestructura Virtualizada
VM	Virtual Machine	Máquina Virtual
VNF	Virtual Network Function	Función de Red Virtualizada
VNFM	VNF Manager	Gestor de VNF
WBS	Work Breakdown Structure	Estructura de Descomposición de Trabajo
WebRTC	Web Real-Time Communication	Comunicación Web en tiempo real
WP	Work Packages	Paquetes de Trabajo

Capítulo 1

Introducción

1.1 Definición del Problema

IMS (*IP Multimedia Subsystem*) es una arquitectura de telecomunicaciones que provee servicios de voz y multimedia sobre IP independiente del acceso a la red [1, 2]. IMS es importante por diversos aspectos. Primero, esta arquitectura es ampliamente utilizada por los CSP (*Communications Service Provider*) para desplegar redes de telecomunicaciones móviles 3G/4G [3] y nuevas tecnologías, por ejemplo, *Voice over LTE* [4] y *Voice over Wi-Fi* [5]. Segundo, IMS es un componente clave de las NGN (*Next Generation Networks*) encargándose del control de sesión de los servicios de voz y multimedia [6]. Tercero, los servicios basados en IMS cuentan con más de 190 millones de suscriptores alrededor del mundo [7]. Además, considerando las características de esta arquitectura, el ETSI (*European Telecommunications Standards Institute*) ha decidido continuar utilizando IMS para el provisionamiento de servicios multimedia en 5G [8].

Teniendo en cuenta la importancia de IMS, se considera fundamental analizar su rendimiento. En los trabajos [9–11] se hizo este análisis considerando métricas como latencia, *throughput*[1], consumo de CPU (*Central Processing Unit*), consumo

[1] *Throughput* es la tasa máxima de las capacidades de transmisión y recepción de información de una red [9]

de memoria y número de sesiones simultáneas. Los resultados de estos trabajos evidencian que las arquitecturas tradicionales de IMS poseen problemas de rendimiento debido al uso de *hardware* específico [12]. Estos problemas ocurren debido a que el rendimiento de IMS está limitado por el número o características de los equipos *hardware*, para poder mejorar las características de la arquitectura se requiere adquirir nuevos equipos.

Los CSP optaron por virtualizar IMS utilizando NFV (*Network Function Virtualization*) [13] con el objetivo de mejorar la calidad de provisión de sus servicios y disminuir los costos asociados al despliegue y mantenimiento de los mismos [7]. NFV consiste en una separación e independencia entre el *software* y *hardware* de los *middlebox*[2], lo que conlleva a que funciones de red originalmente vinculadas a un *hardware* específico pasen a ser ejecutadas en *software* sin afectar su funcionalidad [15]. El desacople entre *hardware* y *software* trae beneficios en aspectos de portabilidad, facilidad de gestión, flexibilidad de despliegue y evolución [16, 17]. Además, NFV ofrece a los CSP la capacidad de añadir réplicas de una función de red en dispositivos de uso general, lo que implica una mejora importante en la escalabilidad de la arquitectura. Al utilizar NFV en IMS se obtiene un vIMS (*virtualized IP Multimedia Subsystem*).

Aun con las mejoras que brinda NFV, vIMS se enfrenta a retos importantes en términos de escalabilidad [18] y gestión de recursos [19] porque vIMS utiliza las mismas NF (*Network Function*) monolíticas de las arquitecturas IMS tradicionales. Una NF monolítica es aquella que ejecuta varias funciones que no son dependientes entre sí. El uso de NF monolíticas no permite una escalabilidad fina. En este documento, escalabilidad fina se la define como la asignación específica de recursos a las funciones que manejan el tráfico. Esta asignación específica de recursos permitiría a vIMS atender mayor cantidad de usuarios, utilizando eficientemente los recursos disponibles. La escalabilidad fina puede lograrse diseñando el núcleo de vIMS[3] a partir del concepto de microservicios. Este concepto consiste en dividir aplicaciones monolíticas en distintos componentes, cada uno con una función específica e inde-

[2] Un *middlebox* es cualquier dispositivo intermediario que realice funciones distintas a las funciones estándar de un enrutador entre un *host* de origen y un *host* de destino [14]

[3] El *núcleo de vIMS* son los componentes más importantes de IMS, los cuales son las CSCF y el HSS

pendiente de las demás [20]. Esta división es ideal para compañías con un alto número de usuarios [21] que requieren una mejora en escalabilidad [22] y gestión de recursos [19]. Es importante mencionar que una de las características fundamentales de microservicios es el uso de virtualización de sistema operativo [23] en la cual se utilizan contenedores, diferente de la virtualización por *hardware* en la cual se utilizan VM (*Virtual Machines*) y es la más utilizada para los despliegues de vIMS. La *virtualización de sistema operativo*, también llamada virtualización por *software*, comparte los recursos del *kernel* y del sistema operativo a múltiples contenedores. La *virtualización por hardware* toma parte de los recursos de la máquina física y se los asignan a múltiples VM.

En la literatura existen trabajos [24–28] que realizan una comparación de rendimiento entre VM y contenedores para VNF (*Virtual Network Functions*)[4]. Para esta comparación se utilizaron métricas como latencia, consumo de CPU o consumo de memoria. Por otro lado, se encontraron trabajos [2, 13, 29, 30] que realizan un análisis del rendimiento del núcleo de un vIMS con el objetivo de mejorarlo en aspectos de gestión de recursos, escalabilidad, confiabilidad. Estos trabajos utilizan métricas como CPS (*Calls Per Second*), latencia o consumo de recursos[5]. Cabe aclarar que en ninguno de los trabajos mencionados se realiza un análisis que permita evidenciar el efecto del tipo de virtualización en el rendimiento de vIMS. Este análisis fue realizado en otro trabajo [31], pero no consideró microservicios para el diseño del núcleo vIMS. Por último, se encontraron dos trabajos [19, 32] en los que se analiza el rendimiento de vIMS diseñado con microservicios. Sin embargo, en [19] solo se considera las funciones de autorización, autenticación y registro del IMS, por lo que este es insuficiente para proveer todas las funcionalidades del núcleo de IMS. En [32] se despliegan todas las CSCF (*Call Session Control Function*) en un solo microservicio, lo cual no es coherente con el enfoque de microservicios.

El análisis de la literatura evidencia que la evaluación del rendimiento de un núcleo vIMS diseñado con microservicios no ha sido lo suficientemente explorado. Los microservicios implican una mejora significativa en términos de escalabilidad y gestión de recursos, retos actuales de vIMS. Además, se debe tener en cuenta el tipo de vir-

[4] *VNF* es una función de red virtualizada que inicialmente estaba vinculada a *hardware* específico
[5] El *consumo de recursos* es la cantidad de memoria y CPU utilizadas

tualización porque los microservicios utilizan virtualización por *software*, y los vIMS tradicionales utilizan virtualización por *hardware* [7]. Por lo tanto, esta monografía toma como objeto de estudio[6] analizar el rendimiento del núcleo vIMS diseñado con microservicios y desplegado sobre contenedores, comparándolo con el rendimiento del núcleo vIMS desplegado sobre VM. El rendimiento debe analizarse considerando las métricas de CPS, latencia, consumo de CPU y consumo de memoria, puesto que estas son las métricas más utilizadas para analizar la arquitectura vIMS. De esta manera se busca responder la siguiente pregunta:

¿Cuál es el rendimiento de un núcleo vIMS diseñado con microservicios considerando las métricas de CPS, latencia, consumo de CPU y consumo de memoria?

Para responder esta pregunta de investigación los objetivos presentados en el siguiente numeral son considerados.

[6]Un *objeto de estudio* es la tecnología base sobre la cual se desarrolla un proyecto

1.2 Objetivos

1.2.1 Objetivo General

Analizar el rendimiento de un núcleo vIMS diseñado con microservicios.

1.2.2 Objetivos Específicos

- Diseñar un núcleo vIMS bajo un enfoque de microservicios.
- Adaptar un núcleo vIMS de fuente abierta que soporte microservicios desplegado con contenedores.
- Evaluar el rendimiento de la adaptación realizada a través de las métricas de CPS, latencia, consumo de CPU y consumo de memoria comparándolas con las de un núcleo vIMS desplegado sobre VM.

1.3 Aportes Investigativos

- Diseño de un núcleo vIMS bajo un enfoque de microservicios.
- Prototipo de fuente abierta de un núcleo vIMS diseñado con microservicios desplegado con contenedores.
- Evaluación del rendimiento del prototipo del núcleo vIMS diseñado con microservicios comparado con un núcleo vIMS desplegado con VM, considerando las métricas de CPS, latencia, consumo de CPU y consumo de memoria.

1.4 Actividades y Cronograma

Este trabajo de grado tomó como referencia WBS (*Work Breakdown Structure*) [33], una herramienta que facilita la definición y organización de actividades necesarias para llevar acabo un proyecto. WBS define una estructura jerárquica y vertical donde se descompone un proyecto en distintos componentes denominados WP (*Work Packages*), los cuales a su vez se dividen en tareas y actividades [33].

1.4.1 WP1 Generación de la base de conocimiento

- Revisión del estado del arte. Se realizó una revisión del estado del arte para identificar brechas de investigación.
- Síntesis de información. Se sintetizó la información de la revisión del estado del arte en una base teórica, útil para desarrollar el análisis de rendimiento del Subsistema Multimedia IP basado en NFV considerando microservicios.

1.4.2 WP2 Diseño del Subsistema Multimedia IP Virtualizado basado en Microservicios

- Exploración de herramientas de fuente abierta para virtualizar IMS. Se exploraron distintas herramientas con las cuales virtualizar IMS.
- Diseño orientado a microservicios del núcleo de un vIMS. Se realizó un diseño orientado a microservicios de IMS con las bases teóricas obtenidas en el WP1.

1.4.3 WP3 Implementación del Subsistema Multimedia IP Virtualizado basado en Microservicios

- Implementación del vIMS utilizando VM. Se implementó un vIMS con VM como referencia del rendimiento de un vIMS sin microservicios utilizando

Clearwater, el cual es un vIMS de *software* libre.

- Implementación del vIMS diseñado con microservicios utilizando contenedores. Se implementó un prototipo del diseño del vIMS basado en microservicios adaptando Clearwater.

1.4.4 WP4 Pruebas y análisis de resultados

- Exploración y selección de herramientas para realizar las pruebas del rendimiento de vIMS. Se seleccionó SIPp para evaluar el rendimiento del vIMS sin microservicios y el vIMS basado en microservicios.
- Recolección de datos de rendimiento de los vIMS implementados. Se realizaron y recolectaron los resultados de la comparación de rendimiento del vIMS sin microservicios y vIMS basado en microservicios, utilizando las herramientas previamente seleccionadas.
- Análisis de los resultados obtenidos. Se analizó el rendimiento de vIMS sin microservicios y el vIMS basado en microservicios considerando las métricas de CPS, consumo de CPU, consumo de memoria y latencia.

1.4.5 WP5 Divulgación

- Elaboración del artículo. Se elaboró un artículo para conferencia llamado "*μvIMS: A Finer-Scalable Architecture Based on Microservices*"
- Elaboración de la monografía. Se elaboró la monografía detallando el análisis de rendimiento de un vIMS basado en microservicios.

La Figura 1.1 muestra el cronograma planteado con el que se desarrolló las actividades propuestas en este trabajo de grado.

Table 1.1: Cronograma

Actividades	**Meses**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
WP1 Generación de la base de conocimiento									
Revisión del estado del arte.									
Síntesis de información.									
WP2 Diseño del Subsistema Multimedia IP Virtualizado basado en Microservicios									
Exploración de herramientas de fuente abierta para virtualizar IMS.									
Diseño orientado a microservicios del núcleo de un vIMS.									
WP3 Implementación del Subsistema Multimedia IP Virtualizado basado en Microservicios									
Implementación del vIMS utilizando VM.									
Implementación del vIMS diseñado con microservicios utilizando contenedores.									
WP4 Pruebas y análisis de resultados									
Exploración y selección de herramientas para realizar las pruebas del rendimiento de vIMS.									
Recolección de datos de rendimiento de los vIMS implementados.									
Análisis de los resultados obtenidos.									
WP5 Divulgación									
Elaboración del artículo.									
Elaboración de la monografía.									

1.5 Publicaciones

El trabajo presentado en esta monografía fue enviado a la comunidad científica a través de un artículo a una conferencia de renombre (Anexo B)

- **Juan S. Orduz, Gabriel D Orozco**, Carlos H. Tobar-Arteaga and Oscar Mauricio Caicedo Rendon. **μvIMS: A Finer-Scalable Architecture Based on Microservices** IEEE Local Computer Networks (LCN 2019), Octubre 14-17
  - Estado: Aceptado.
  - Índice H: 45 Scimago
  - Rango: A Core

1.6 Estructura del Documento

Este documento se divide en los capítulos descritos a continuación.

- Capítulo 1 presenta la **Introducción** que contiene definición del problema, objetivos, aportes investigativos, actividades y cronograma, publicaciones y la estructura de este documento.
- Capítulo 2 presenta los **Conceptos Fundamentales** de la investigación. Estos conceptos son esenciales para entender el desarrollo del presente trabajo.
- Capítulo 3 presenta los **Trabajos Relacionados** que describen investigaciones cercanas a la de este trabajo.
- Capítulo 4 introduce al **Subsistema Multimedia IP Virtualizado basado en Microservicios**.
- Capítulo 5 presenta la **Evaluación del Subsistema Multimedia IP Virtualizado basado en Microservicios**. Para esta evaluación se define un prototipo, escenario de prueba y el análisis de los resultados.
- Capítulo 6 presenta **Conclusiones y Trabajos Futuros**. Se provee una conclusión principal del trabajo realizado y sus implicaciones.

Capítulo 2

Conceptos Fundamentales

2.1 Núcleo del Subsistema Multimedia IP Virtualizado

El núcleo de IMS se encarga del control de sesión multimedia. Este núcleo está compuesto de las CSCF que son: P-CSCF (*Proxy-CSCF*), S-CSCF (*Serving-CSCF*) e I-CSCF (*Interrogating-CSCF*). Estos elementos utilizan al HSS (*Home Subscriber Server*) como base de datos y a los AS (*Application Server*) para proveer servicios. Las funciones de los elementos anteriormente nombrados son las siguientes:

- P-CSCF es el primer punto de contacto entre los usuarios y el núcleo de IMS, valida las solicitudes externas y las enruta a su destino.
- S-CSCF controla las sesiones multimedia, el registro de nuevos usuarios y reenvía las solicitudes al elemento IMS indicado, por ejemplo, se lo puede enviar a un P-CSCF diferente al que está atendiendo al emisor.
- I-CSCF revisa el perfil del usuario, asigna un usuario a un S-CSCF, y enruta las solicitudes a otros IMS.
- HSS es una base de datos que contiene la información de los usuarios (*e.g.*, perfil del subscriptor y localización). Es importante mencionar que HSS no es

solo utilizado por los elementos de IMS sino también por otras arquitecturas de telecomunicaciones como el *Long Term Evolution*.

- AS (*Application Server*) es un elemento para el aprovisionamiento de servicios que tiene la función de ejecutar y hospedar servicios para los clientes. Por ejemplo, servicios de correo de voz, reenvío de llamadas y encolamiento de llamadas.

Dada la importancia de IMS en las redes actuales y futuras. Los CSP han optado por virtualizar IMS utilizando NFV [7]. Una arquitectura IMS basada en NFV se le denomina vIMS. Diferentes proyectos se han enfocado en virtualizar IMS [34, 35] con el objetivo de aprovechar las ventajas que conlleva virtualizar las funciones de red. Entre estas ventajas sobresalen la disponibilidad, confiabilidad y flexibilidad de despliegue [36]. Por ejemplo, con dos instancias virtuales del S-CSCF y un balanceador de carga es posible mover los usuarios asignados de un S-CSCF principal a uno de respaldo de manera transparente y distribuir el tráfico entre las distintas funciones del núcleo, ofreciendo una mejor QoS (*Quality of Service*) a los usuarios [10].

En la literatura se mencionan herramientas para virtualizar IMS, clasificadas en dos vertientes importantes:

- La virtualización por *hardware* en la cual se utilizan VM.
- la virtualización por *software* en la cual se utilizan contenedores.

Subsección 2.1.1 describe el vIMS virtualizado con VM y Subsección 2.1.2 describe el vIMS virtualizado con contenedores.

2.1.1 Subsistema Multimedia IP Virtualizado con Máquinas Virtuales

Cuando se utiliza VM como elemento de virtualización, los recursos disponibles de una máquina física se comparten a múltiples VM. Estas a su vez suelen ser de dos tipos: las BM (*Bare-Metal*) y VM hospedadas.

- BM consiste en un hipervisor que se instala directamente en una máquina física y permite ejecutar nativamente múltiples VM [37], por ejemplo, ESXi[1]
- VM hospedadas se ejecutan dentro de un sistema operativo nativo o anfitrión [39], algunos ejemplos son Virtualbox[2] y KVM[3].

En la actualidad existen herramientas que permiten desplegar un vIMS siguiendo la virtualización por *hardware* o con VM. Por ejemplo, OpenIMSCore [41] brinda la posibilidad de desplegar las CSCF y el HSS estandarizados por el 3GPP en una o varias VM, en este último despliegue cada CSCF y el HSS se despliegan en una VM. Otra opción es el Proyecto Clearwater [42] que provee las CSCF en 5 componentes (*i.e.,* Bono, Sprout, Homer, Dime y Vellum). Cada uno desplegado en una VM, que se comunican entre sí utilizando protocolos específicos como HTTP o SIP y que respetan las interfaces definidas por el 3GPP con los elementos externos al núcleo IMS.

2.1.2 Subsistema Multimedia IP Virtualizado con Contenedores

En la virtualización de sistema operativo o con contenedores, en vez de utilizar un hipervisor para aislar los recursos *hardware*, se comparten los recursos del sistema operativo y del *kernel* entre todos los contenedores. Los procesos que corren dentro de un contenedor se ejecutan nativamente en el *kernel*, esto brinda mejoras en el uso de recursos sin el gasto adicional que supone el uso del hipervisor [37, 39]. Algunos virtualizadores basados en contenedores son LxC (*Linux Container*), *Warden Container* y Dockers [43]. Figura 2.1 muestra una comparación entre la virtualización por *hardware* y por *software*.

[1] *ESXI* o *VMware ESXI* es un hipervisor BM que se instala directamente en una máquina física y permite crear múltiples VM [38]

[2] *Virutalbox* es un *software* de fuente abierta que permite virtualizar múltiples VM dentro del sistema operativo nativo o anfitrión [40]

[3] *KVM* es una tecnología de fuente abierta que convierte a linux en un hipervisor permitiendo compartir los recursos *hardware* de una máquina física a múltiples VM

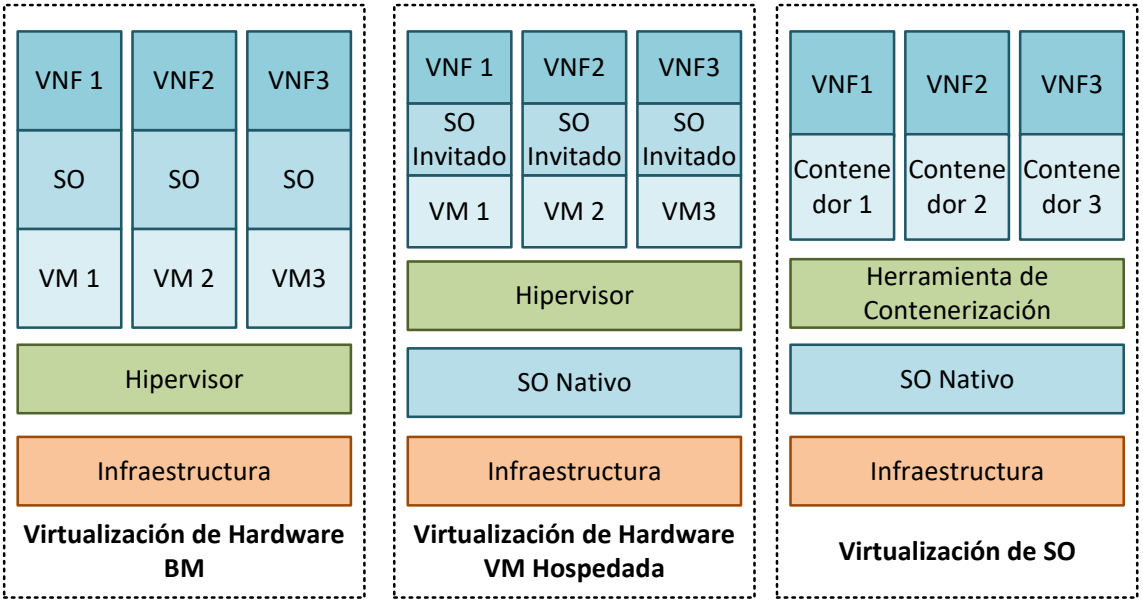

Figura 2.1: Comparación entre virtualización por *hardware* y *software*

La virtualización por *software* brinda una mejora con respecto a la virtualización por *hardware* en tiempo de despliegue y la escalabilidad, ya que permite crear contenedores con la misma función que una VM de forma más rápida [26]. Esto a su vez permite escalar migrar y actualizar la arquitectura en menos tiempo, permitiendo a los CSP adaptar su arquitectura a tráficos altos y ahorrar costos en tráficos bajos.

Es posible desplegar las mismas funciones sobre VM o contenedores. Por lo tanto, soluciones como OpenIMSCore pueden ser desplegadas sobre contenedores aprovechando sus ventajas. Por otro lado, Proyecto Clearwater posee un diseño específico para contenedores Docker el cual separa bases de datos e interfaces que se encontraban en una misma entidad en el despliegue de Clearwater para VM.

2.2 Arquitectura de Microservicios

Microservicios es un modelo arquitectónico donde una aplicación monolítica es dividida en múltiples componentes independientes, cada uno con una función específica [20]. La división funcional permite crear y eliminar microservicios de forma más

rápida y sencilla, lo que permite asignar recursos a ciertos microservicios en particular. Esta asignación específica de recursos conduce a una respuesta más eficiente para el tráfico entrante.

Para el funcionamiento apropiado de los microservicios es necesario un grupo de elementos que en su conjunto conforman la MSA (*MicroService Architecture*). Aunque no existe un estándar o especificación de la MSA, la Tabla 2.1 muestra algunas características y elementos utilizadas en las implementaciones encontradas en la literatura, con las cuales se determinó un modelo básico de la MSA.

Para entender las características de la MSA disponibles en la literatura, estas son clasificadas en dos grupos. Las *Particularidades de microservicios* y los *Elementos MSA*. Las *Particularidades de Microservicios* consisten en las propiedades de los microservicios que dividen una aplicación monolítica. En este grupo se tiene:

1. La descomposición funcional tiene en consideración el modelo de negocio [19]. Esta descomposición indica que cada microservicio debe proveer algo de valor a los usuarios manteniendo un tamaño adecuado. Microservicios que agrupen muchas funciones retornan al diseño monolítico, mientras que microservicios muy pequeños separan funciones fuertemente ligadas, añadiendo complejidad de gestión [44].

2. Cada microservicio es independiente de los otros, por ello la comunicación entre microservicios necesita un protocolo de comunicación común como HTTP [45].

3. Cada microservicio tiene su propia base de datos para lograr un aislamiento de datos [32, 45]. Una base de datos centralizada implica perder la independencia de los microservicios.

El segundo grupo de características MSA son los *Elementos MSA*, los cuales aseguran el adecuado funcionamiento de los microservicios proporcionándoles seguridad, confiabilidad y gestión. Estos elementos son:

- *Balanceador de carga* que distribuye el tráfico de la red cuando un microservicio es escalado en varias instancias para mejorar las capacidades de la arquitectura [32].

Table 2.1: Características MSA

Trabajo	Particularidades de microservicios			Elementos MSA				
	Descomposición funcional considerando modelo de negocio	Protocolo de comunicación común	Microservicios con datos propios	Balanceador de carga	Pasarela API	Servicio de descubrimiento	Corta Circuitos	Orquestador
A VNF-as-a-Service Design Through Micro-Services Disassembling the IMS	✓							
Micro Service Cloud Computing Pattern for Next Generation Networks			✓	✓				
Orchestration of Containerized Microservices for IoT using Docker		✓	✓					✓
Building Microservices			✓		✓	✓	✓	
Microservices						✓		✓
Circuit Breakers, Discovery, and API Gateways in Microservices						✓	✓	✓

- *Pasarela API* es una interfaz encargada de prevenir el establecimiento de una comunicación directa entre usuarios y los microservicios, ocultando la estructura interna de la arquitectura [46].
- *Servicio de descubrimiento* almacena y provee las direcciones de los microservicios con el objetivo de establecer una comunicación efectiva entre los elementos de la arquitectura [47–49].
- *Corta Circuitos* está encargado de responder a las fallas cuando un microservicio no está funcionando de manera adecuada [46, 48, 49].
- *Orquestador* [45, 47–49] gestiona los recursos asignados a los microservicios (*i.e.*, crear, eliminar, replicar y actualizar las instancias de microservicios), esto permite incrementar y liberar recursos.

2.3 Métricas de Rendimiento

Al evaluar el rendimiento de un vIMS diseñado bajo el enfoque de microservicios es necesario considerar las mismas métricas que se hayan utilizado para evaluar un vIMS con un enfoque monolítico, puesto que el vIMS diseñado con microservicios se ejecuta en el mismo entorno que un vIMS sin este enfoque. En la literatura las métricas para está evaluación son: latencia [31], consumo de memoria [19], consumo de CPU [50], y CPS [36].

En concreto, con microservicios se espera aumentar el número de CPS que el sistema puede soportar correctamente sin un aumento excesivo de latencia y manteniendo el consumo de memoria y CPU a niveles comparables al de un despliegue vIMS sin el enfoque de microservicios. Por lo tanto, las métricas de rendimiento que se utilizan para evaluar un vIMS diseñado con microservicios son:

- Consumo de recursos es el consumo de CPU y memoria. Su análisis permite identificar si la arquitectura está asignando recursos de manera eficiente, sin incurrir en uso excesivo de los mismos.

- CPS es el número de llamadas simultaneas que ingresan a la arquitectura. Además, se utiliza junto a la SCR (*Succesful Call Rate*) para determinar el número de llamadas exitosas que logra atender la arquitectura, permitiendo comparar la mejora que los microservicios ofrecen con respecto a un despliegue monolítico.

- Latencia es el tiempo entre el envió del mensaje y su respectiva respuesta [51]. Dado que la latencia es una métrica relacionada con la calidad de servicio percibida por el usuario, debe medirse el aumento de latencia producido al utilizar el enfoque de microservicios y comprobar si esta se encuentra en los límites aceptados, específicamente para la señalización IMS el límite superior de latencia es de 100 ms [52].

Capítulo 3

Trabajos Relacionados

Esta sección presenta una revisión de los trabajos relacionados con el análisis de rendimiento de un IMS basado en NFV considerando microservicios. Estos trabajos fueron clasificados en tres secciones. Sección 3.1 presenta los trabajos que realizan una comparación entre VM y contenedores. Es importante este análisis puesto que en este trabajo se compara un vIMS desplegado con VM con un vIMS diseñado con microservicios desplegado sobre contenedores. Sección 3.2 presenta los trabajos en los que se realiza un análisis de rendimiento de vIMS teniendo en consideración uso de recursos, CPS y escalabilidad. Sección 3.3 presenta los trabajos en los que se analiza un vIMS diseñado con microservicios. Por último, este capítulo termina con unas conclusiones finales.

3.1 Comparación Máquinas Virtuales y Contenedores

En el estudio *"Performance Evaluation of a Virtualized HTTP Proxy in KVM and Docker"* [24] se realizó un análisis de rendimiento entre KVM y Docker en un escenario de pruebas con un servidor *proxy* HTTP como VNF. Para el análisis se consideró el comportamiento de las CDF (*Cumulative Distribution Function*) que

representan la latencia en KVM y Docker. El estudio arrojó como conclusión que Docker tiene un tiempo de procesamiento más rápido que KVM (aproximadamente el doble de rápido) en el ambiente específico del estudio.

En *"Performance Comparison Between Linux Containers and Virtual Machines"* [25] se realizó una comparación cuantitativa entre LxC y VM en términos de rendimiento. Para realizar la comparación se definió un *benchmark* en el que se midieron métricas como latencia y velocidad de procesamiento de un servidor con una aplicación PHP, ejecutándola tanto en VM como en LxC. Los resultados de este trabajo evidenciaron que LxC tiene un mejor rendimiento que las VM, aunque estas pueden ser más adecuadas en casos donde la información transmitida sea confidencial. Debido a que el hipervisor provee aislamiento.

En el artículo *"Container Network Functions: Bringing NFV to the Network Edge"* [26] se realizó un análisis para determinar qué tipo de virtualización es mejor para el borde de la red y se presenta GNF (*Glasgow Network Functions*), una plataforma NFV basada en contenedores. Para seleccionar el tipo de virtualización a utilizar en GNF, se comparó el rendimiento utilizando métricas como latencia, tiempo de inicialización y flexibilidad de despliegue de tres opciones: VM, VM especializadas[1] y contenedores. Esta comparación arrojó como resultado que los contenedores ofrecen un balance entre rendimiento y flexibilidad de despliegue logrando correr en cualquier ambiente linux.

El trabajo *"Performance Considerations of Network Functions Virtualization using Containers"* [27] se centró en explorar el uso de tecnologías de enrutamiento para redes de contenedores y comparar el rendimiento de una red virtualizada utilizando VM y contenedores. Para la comparación de rendimiento se tomaron como métricas *jitter* (*i.e.*, variación de retardo) y latencia, entre VNF ejecutadas en VM y en contenedores. Los resultados evidenciaron que las tecnologías de enrutamiento virtualizadas con contenedores tienen menor latencia y menor *jitter*.

En el trabajo *"An empirical case for container-driven fine-grained VNF resource flexing"* [28] se realizó una comparación cuantitativa del rendimiento de VNF uti-

[1]Una *VM especializada* es una VM que utiliza un ambiente *software* específico para ofrecer alto rendimiento en métricas particulares, por ejemplo, bajo retardo [26]

lizando BM, VM y contenedores en dos escenarios. El primer escenario fue un *Evolved Packet Core*, donde se virtualizó solamente la *Mobility Management Entity* y el HSS para determinar el tiempo utilizado para procesar peticiones de registro de usuarios. El segundo escenario de pruebas fue *Suricata*[2], donde se evaluó la memoria y CPU utilizadas para desplegar el sistema. Como resultado se obtuvo que el rendimiento de las VNF utilizando contenedores es ligeramente inferior si se implementan en BM y superior al de las VM con base en las métricas definidas.

Estos trabajos muestran que los contenedores tienen mejor rendimiento en términos de tiempo de procesamiento, latencia, flexibilidad de despliegue, *jitter*. Ninguno de estos trabajos compara el rendimiento de la arquitectura vIMS desplegada con VM con una desplegada sobre contenedores.

3.2 Rendimiento en el Subsistema Multimedia IP Virtualizado

En el artículo *"Cloudified IP Multimedia Subsystem (IMS) for Network Function Virtualization (NFV)-based architectures"* [29] se presentaron las características de distintas arquitecturas *software* para virtualizar IMS de manera eficiente. Los autores describieron las características de tres vIMS: *"Virtualized IMS"* donde se asigna cada componente (P/S/I-CSCF y HSS) a una VM, *"Split-IMS"* utiliza varias VM trabajando en paralelo para una misma función de red (P/S/I-CSCF) y *"Merge-IMS"* que virtualiza la CSCF y el HSS en una misma VM. Además, se realizó el análisis del rendimiento de *"Merge-IMS"* midiendo la memoria y CPU utilizadas por la CSCF. Debido a las limitaciones de memoria de S-CSCF solo se alcanzaron 50 CPS, los autores concluyeron que es un número elevado de CPS para un número limitado de instancias.

En el artículo *"NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions"* [30] se presentó el desarrollo de "NFV-VITAL", un *framework* para caracterizar VNF basado en las preferencias de los usuarios y los

[2] *Suricata* es un sistema de detección de intrusos IDS (*Intrusion Detection System*) [28]

recursos disponibles. En el artículo se expuso un análisis de rendimiento variando la escalabilidad de un núcleo vIMS, utilizando como métricas el uso de CPU y el número de CPS exitosas. Los autores concluyeron basados en su *framework* que aumentar los recursos de las instancias virtualizadas conlleva a una mejora lineal del rendimiento de vIMS.

En *"Dependability evaluation and benchmarking of Network Function Virtualization Infrastructures"* [53] se expuso una metodología experimental para la evaluación de confiabilidad y *benchmark* de infraestructuras NFV basada en la inyección de fallas. Los autores realizaron un análisis de rendimiento basado en las métricas de latencia y *throughput* en entornos donde se presentan fallas, como estudio de caso se utilizó un vIMS con *Clearwater* [42]. En este artículo se concluyó, que utilizando las métricas seleccionadas para el análisis de rendimiento su *benchmark* fue capaz de identificar los cuellos de botella del vIMS.

En el trabajo *"Load based automatic scaling in virtual IP based multimedia subsystem"* [13], los autores presentaron una comparación entre dos mecanismos de escalado automático para vIMS. El primero se basa en métricas de VM (uso de CPU y RAM). El segundo mecanismo se basa en métricas VNF, por ejemplo, uso de recursos y red, conexiones TCP, número de llamadas perdidas y peticiones SIP (Session Initiation Protocol). Como resultado de la comparación, los autores determinaron que el uso de métricas VNF es la mejor manera de escalar el vIMS.

En *"Quality Audit and Resource Brokering for Network Functions Virtualization (NFV) Orchestration in Hybrid Clouds"* [2], los autores propusieron un *framework* de auditoría de calidad y de intermediación de recursos NFV. Este *framework* permite escalar vIMS en función de la carga del usuario y asignar los recursos en diferentes plataformas de nube. Los autores analizaron su propuesta desplegando un vIMS integrado con su *framework* en un banco de pruebas sobre una nube híbrida privada/pública. Como resultado los autores concluyeron que es posible escalar el vIMS dinámicamente, aprovechando los recursos de las nube.

En el trabajo *"NFV-Bench: A Dependability Benchmark for Network Function Virtualization Systems"* [31] se realizó el diseño de un *benchmark*[3] basado en la mon-

[3]Un *benchmark* es una técnica para medir el rendimiento de un sistema [53]

itorización del rendimiento bajo inyección de fallas llamado *"NFV-Bench"*. Para probar su *benchmark* los autores diseñaron un estudio de caso, utilizando Clearwater para desplegar un vIMS, comparando la respuesta a fallas de la virtualización de *hardware* (utilizando VMware ESXi) con la de la virtualización de *software* (utilizando Docker). En esta comparación se consideraron métricas como tiempo de inactividad y *throughput*. Basados en los resultados obtenidos por las pruebas con su *benchmark*, los autores concluyeron que el vIMS desplegado con contenedores es menos confiable, debido a que los contenedores son más susceptibles a fallas.

Los análisis presentados en estos trabajos permiten identificar las métricas utilizadas para hacer un análisis de rendimiento de vIMS, estas son: uso de recursos, CPS y latencia. Todos los trabajos presentados comparten la misma brecha, las arquitecturas vIMS evaluadas tienen un enfoque monolítico. Por lo tanto, ninguno presentó el análisis de rendimiento de un vIMS diseñado con microservicios, los cuales permiten una escalabilidad fina para manejar el tráfico más eficientemente.

3.3 Microservicios en el Subsistema Multimedia IP

En *"A VNF-as-a-service design through micro-services disassembling the IMS"* [19], los autores propusieron un enfoque para lograr un diseño VNF óptimo utilizando microservicios. Con este enfoque, ellos presentaron *IMS-as-a-Service* que proporciona registro, autorización y autenticación de IMS como servicio. Los autores probaron *IMS-as-a-Service* comparándolo con un vIMS sin microservicios. Como resultado, *IMS-as-a-Service* alcanzó un mayor número de llamadas exitosas y un mejor rendimiento en la utilización de recursos. Sin embargo, los autores se centraron en los procesos de registro, autorización y autenticación de IMS y no consideraron las funciones completas del núcleo IMS.

El trabajo *"Micro Service Cloud Computing Pattern for Next Generation Networks"* [32] describió una arquitectura para la implementación elástica de vIMS basada en microservicios en la nube. Los autores asignaron la información de HSS en las bases

de datos de microservicios. Además, compararon su implementación con un vIMS sin microservicios considerando el retardo en el establecimiento de llamadas. Concluyeron que su arquitectura mantiene resultados similares a los del vIMS regular. Sin embargo, en este trabajo las CSCF estuvieron implementadas un solo microservicio, impidiendo una escalabilidad fina. Por lo tanto, no fue posible asignar recursos en las funciones específicas que manejan el tráfico.

Como se puede observar, las soluciones existentes no están diseñadas para implementar el completo aprovisionamiento de las funciones de IMS o no cumplen los criterios de diseño de microservicios especificados en la Sección 2.2. Por lo tanto, se concluye la necesidad de un análisis de rendimiento de una arquitectura vIMS diseñada con microservicios considerando las métricas de CPS, uso de recursos y latencia.

3.4 Conclusiones Finales

- De acuerdo al análisis de la literatura solamente un trabajo compara el rendimiento entre VM y contenedores en un vIMS [31]. Sin embargo, en este análisis se consideró la confiabilidad. Lo que indica que no existe un trabajo que compare el rendimiento de distintos tipos de virtualización para el despliegue de un vIMS con respecto a las métricas planteadas en este trabajo. Es importante esta comparación dado que los contenedores son el mecanismo de virtualización que se utiliza en el diseño de arquitecturas con microservicios, por lo que guarda una estrecha relación con el rendimiento de un vIMS diseñado con microservicios.

- El análisis de la literatura permite evidenciar que el rendimiento de un vIMS ha sido ampliamente analizado, considerando uso de recursos, CPS, confiabilidad y escalabilidad. Sin embargo, los trabajos en los que se realiza este análisis consideran un vIMS con una arquitectura monolítica. Este tipo de arquitecturas no proveen escalabilidad fina, lo que conlleva a un uso ineficiente de recursos.

- El concepto de un vIMS diseñado con microservicios ha sido poco analizado, muestra de ello es que en el análisis de la literatura solo se encontraron dos trabajos [32] [19]. Debido a la carencia de estudios sobre un vIMS diseñado con MSA y las ventajas que microservicios puede ofrecer a esta arquitectura en términos de escalabilidad y gestión de recursos, es necesario investigar en este campo y determinar si el rendimiento de vIMS con una MSA considerando las métricas seleccionadas es peor, igual o mejor al de un vIMS sin este modelo arquitectónico.

Capítulo 4

Subsistema Multimedia IP Virtualizado basado en Microservicios

En este capítulo se presenta el diseño del Subsistema Multimedia IP Virtualizado basado en microservicios que de ahora en adelante se le denominará μvIMS. El diseño se presenta en cuatro Secciones. Sección 4.1 presenta dos escenarios de motivación. Sección 4.2 introduce una visión general de μvIMS. Sección 3.3 presenta el *Clúster de microservicios CSCF*. Sección 4.4 introduce los *Elementos MSA*, los cuales fueron adaptados de los elementos de la literatura descritos en la Sección 2.2 tomando en consideración las características de μvIMS.

4.1 Motivación

En esta Sección se plantean dos escenarios de motivación en los que se muestra la importancia de realizar el análisis de rendimiento de un vIMS diseñado con microservicios.

Primero, se considera el escenario de un operador de telecomunicaciones que tiene un

IMS con el que brindó servicios de voz en una región durante varios años. Después de analizar a su competencia, el operador concluye que deben proporcionar servicios 5G (*e.g.,* Telemedicina, *Gamming*, *Smartcities*, Inteligencia Artificial Vehicular e IoT) para seguir siendo competitivo. Para soportar el tráfico que generan estos nuevos servicios, el operador precisa migrar su IMS de una arquitectura vIMS monolítica a una con microservicios (Figura 4.1). Este operador necesita saber si utilizar un enfoque de microservicios para su implementación IMS le va a permitir atender más tráfico del que alcanzaría con un arquitectura vIMS monolítica. Además, dado que la latencia guarda una estrecha relación con la calidad de servicio percibida por el usuario en los nuevos servicios 5G desplegados, es necesario que el operador tenga la certeza de que su implementación no implica un aumento excesivo de latencia.

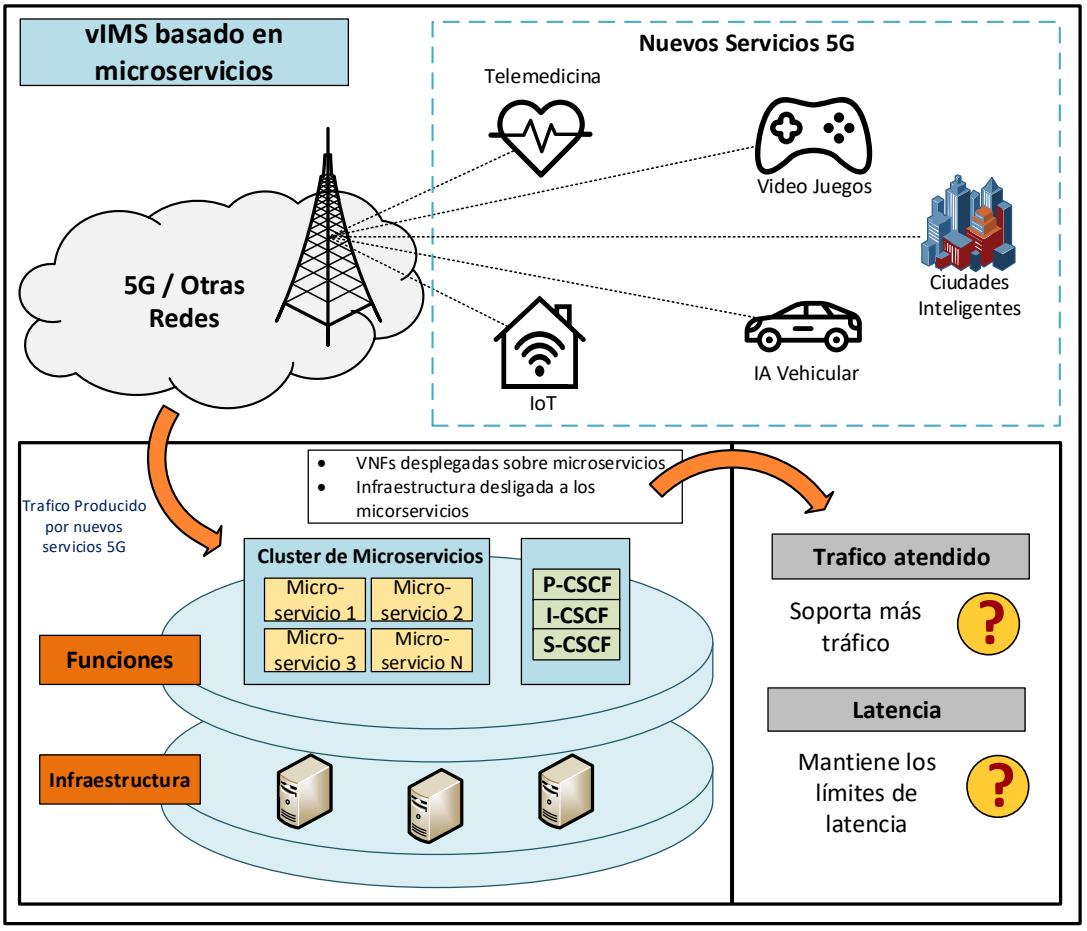

Figura 4.1: Escenario de motivación adición de nuevos servicios 5G

El segundo escenario es el de un CSP que debido a una migración poblacional en la zona donde presta el servicio de voz, necesita aumentar la cantidad de usuarios que soporta su IMS sin invertir en infraestructura, debido a que esto acarrearía un costo adicional. Por esa razón el CSP decide cambiar su arquitectura vIMS monolítica a una con microservicios (Figura 4.2). En este caso, debido a que el número de usuarios atendidos va aumentar, la nueva arquitectura vIMS deberá ser capaz de soportar más llamadas con los recursos disponibles. Por lo tanto, el CSP necesita un estudio que le permita saber si su nueva arquitectura maneja correctamente un número de llamadas más alto que la arquitectura vIMS monolítica con la misma cantidad de recursos disponibles.

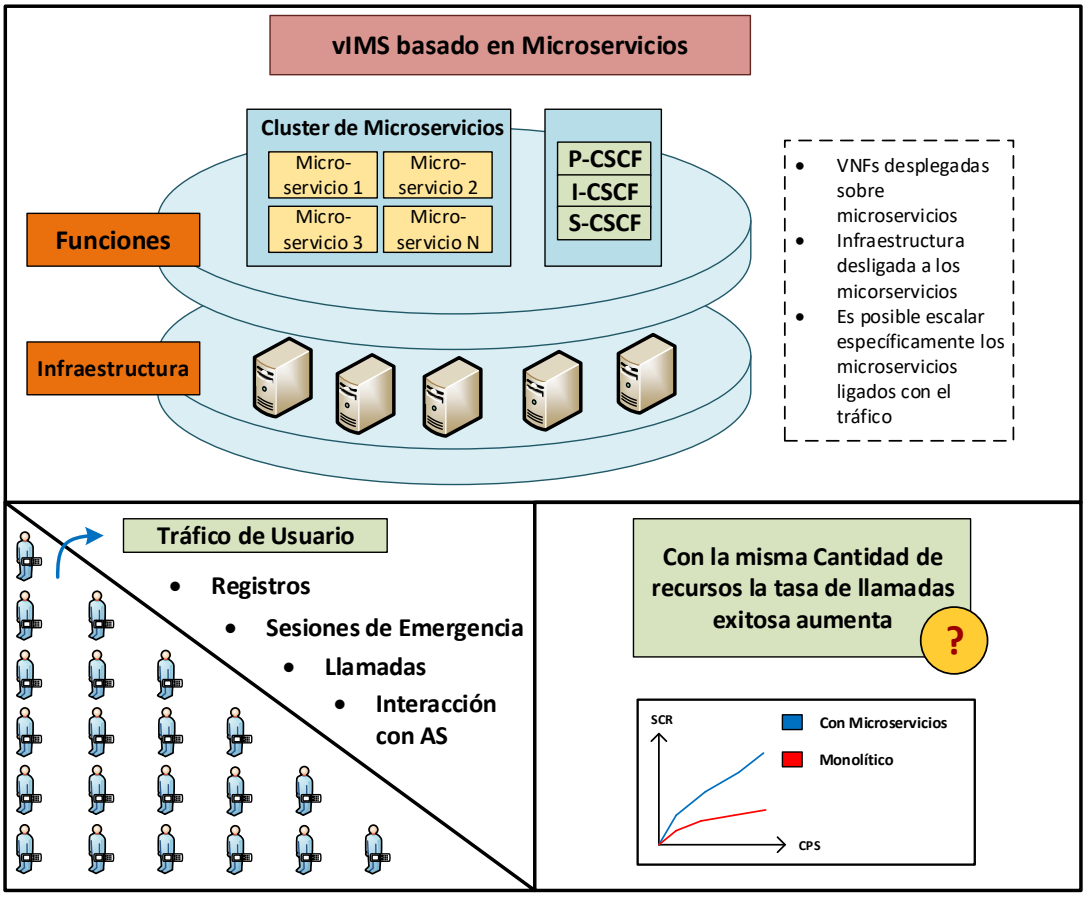

Figura 4.2: Escenario de motivación gestión de recursos disponibles

4.2 Vista General

μvIMS divide las funciones principales de IMS en microservicios para enfrentar el tráfico impredecible de 5G y otras redes. Esta división funcional permite asignar recursos solo en los microservicios necesarios para manejar el tráfico, lo que permite una escalabilidad fina y reduce el uso innecesario de recursos. Para mantener el funcionamiento adecuado de estos microservicios se utilizan los *Elementos MSA* descritos en la Sección 2.2. Los cuales se encargan del registro de microservicios, la gestión de microservicios, y la interoperabilidad con elementos externos a la arquitectura.

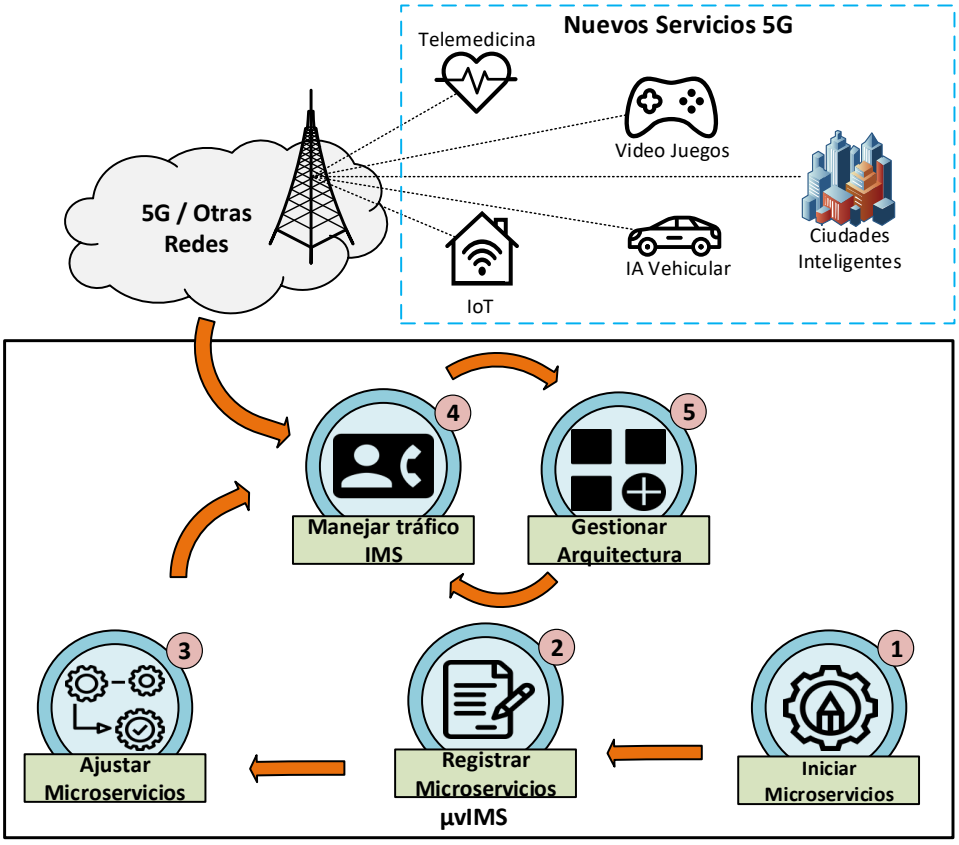

Figura 4.3: Vista General de μvIMS

La Figura 4.3 muestra la operación general de μvIMS:

1. Se inician los microservicios que realizan las funciones principales de IMS.
2. Se registra las direcciones de microservicios para garantizar la comunicación entre ellos y otros elementos arquitectónicos.
3. Se ajusta los microservicios para permitirles manejar el tráfico entrante.
4. Se maneja el tráfico originado por 5G y otras redes.
5. Se gestiona la arquitectura. Esta gestión consiste en manejar las fallas en los microservicios, supervisar el rendimiento de la arquitectura y escalarla. Es importante destacar que la arquitectura es capaz de agregar nuevas instancias de microservicio con el objetivo de aumentar el número de usuarios manejados.

4.3 Clúster de Microservicios de Función de Control de Sesión de Llamada

Este Clúster utiliza microservicios para construir las CSCF responsables del control de sesiones multimedia de usuarios y de los AS. Estos microservicios ofrecen a μvIMS las siguientes características:

1. La división funcional de la CSCF puede mantener un tamaño adecuado. Debido a que microservicios de gran tamaño agrupan muchas funciones, volviendo al diseño monolítico. Microservicios de tamaño pequeño desacoplan funciones muy dependientes entre sí, lo que resulta en una complejidad de gestión adicional [19, 44]
2. La división funcional de la CSCF mantiene la independencia de los datos utilizando una base de datos propia para cada microservicio, una base de datos centralizada implica la pérdida de independencia [32].

La Figura 4.4 muestra la descomposición de CSCF en microservicios. En esta división, se proporciona a cada microservicio una funcionalidad completa e independiente, basándose en las CSCF especificadas por 3GPP para 5G [8].

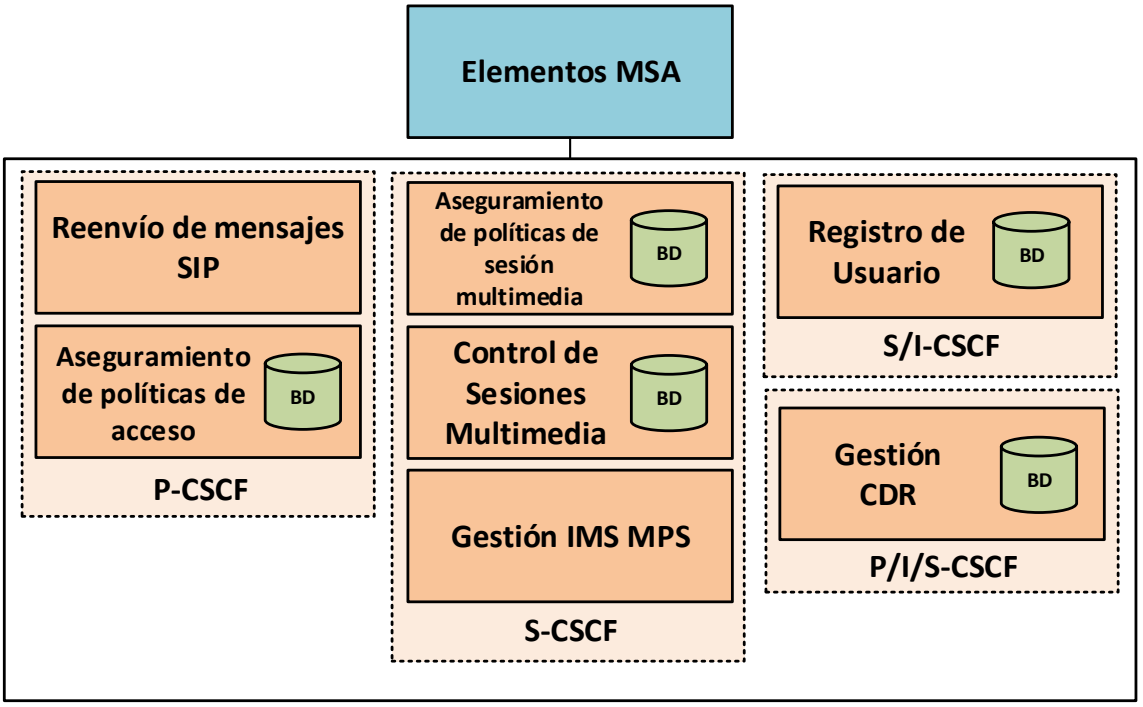

Figura 4.4: Clúster de Microservicios CSCF

Se divide P-CSCF en dos microservicios:

- *Reenvío de mensajes SIP.*
- *Aseguramiento de políticas de acceso.*

Se divide S-CSCF en cuatro microservicios:

- *Registro de usuarios.*
- *Aseguramiento de políticas de sesión multimedia.*
- *Control de Sesiones Multimedia.*
- *Gestión IMS MPS.*

El microservicio *Reenvío de mensajes SIP* es responsable de enrutar los mensajes SIP generados por elementos externos, por ejemplo, teléfonos tradicionales, *softphones*

y WebRTC (*Web Real-Time Communication*). Antes de enrutar los mensajes SIP al microservicio correspondiente, *Reenvío de mensajes SIP* se asegura de que los mensajes tengan el formato correcto, además detecta y enruta solicitudes de sesiones prioritarias.

El microservicio *Aseguramiento de políticas de acceso* gestiona las políticas del operador en el acceso a la arquitectura. Este microservicio almacena las políticas del operador en su base de datos y las proporciona al microservicio *Reenvío de mensajes SIP* para garantizar el enrutamiento de los mensajes SIP de acuerdo con dichas políticas.

El microservicio *Registro de Usuarios* se encarga de autorizar el registro de usuarios en la arquitectura, agrupando las funciones de registro I/S-CSCF. Además, traduce las direcciones E.164 [1] necesaria para algunos tipos de registros de usuarios. Este microservicio necesita una base de datos para almacenar la información de los usuarios para el proceso de registro o determinar que el usuario ya está registrado previamente. La información que este microservicio registra se sincroniza con la base de datos de otros microservicios que atienden a los usuarios después que completan el proceso de registro, por ejemplo, el microservicio *Control de Sesiones Multimedia*. Para realizar esta sincronización los microservicios utilizan el *Gestor de Base de Datos* descrito en la Sección 4.4.

El microservicio *Aseguramiento de políticas de sesión multimedia* gestiona las políticas del operador en el control de sesión multimedia. Este microservicio certifica la identidad del suscriptor si se configura a través de las políticas del operador. *Aseguramiento de políticas de sesión multimedia* almacena las políticas del operador en su base de datos y proporciona estas políticas al microservicio *Control de Sesiones Multimedia*.

El microservicio *Control de Sesiones Multimedia* proporciona las funciones principales del S-CSCF relacionadas con el control de sesiones multimedia. Estas funciones incluyen: inicio, mantenimiento y finalización de sesiones multimedia. Se agrupó el control de sesión multimedia entre dos usuarios y entre un usuario con un AS en

[1] el *E.164* es un número único global para cada dispositivo en la Red Telefónica Pública Conmutada

un solo microservicio porque estos tipos de sesión están fuertemente ligados entre sí. Por ejemplo, una llamada entre dos usuarios podría incluir interacción con un AS para redireccionar o almacenar un mensaje en un correo de voz, o bloquear usuarios no deseados. El uso de dos microservicios (que se encarguen del control de sesión del usuario con usuario y usuario con AS) implica tráfico y lógica adicional para comunicarlos y proporcionar el control de sesión. Para que cada microservicio provea una funcionalidad completa, el microservicio *Control de Sesiones Multimedia* maneja los dos tipos de sesión descritos anteriormente. Además, este microservicio tiene su propia base de datos para manejar la información del usuario, la cual se sincroniza con los usuario registrados por el microservicio *Registro de Usuarios*.

El microservicio *Gestión IMS MPS* maneja IMS MPS (*Multimedia Priority Service*) proporcionando un tratamiento de acceso preferencial a los usuarios prioritarios, cuando la congestión bloquea el establecimiento de sesión. Así pues, este microservicio valida si un usuario está autorizado para utilizar servicios prioritarios ofrecidos a través del AS. De ser este el caso, el microservicio *Gestión IMS MPS* incluye el nivel de prioridad en la solicitud del usuario y la reenvía.

El microservicio *Gestión CDR* gestiona los CDR (*Call Detail Records*), agrupando la lógica P/I/S-CSCF asociada a los CDR[2]. Este microservicio recoge el CDR de otros microservicios, por ejemplo, el microservicio *Control de Sesiones Multimedia* puede registrar el tiempo en que inicia y finaliza la sesión entre dos usuario para determinar el tiempo total de la llamada y determinar el costo de la misma. Con esta información el microservicio *Gestión CDR* genera CDR estandarizados que almacena en su base de datos y envía a una entidad de facturación externa.

[2]Los *CDR* son piezas de información de una llamada, se crea al final de una llamada y tiene datos como número que llama, número llamado, tiempo de llamada, etc [54]

4.4 Elementos de la Arquitectura de Microservicios

La Figura 4.5 presenta los *Elementos MSA*. Los cuales proporcionan seguridad, confiabilidad y gestión de microservicios:

1. *Servicio de Descubrimiento*
2. *Orquestador*
3. *Gestor de Infraestructura*
4. *Corta Circuitos*
5. *Pasarela API*
6. *Gestor de Base de Datos*
7. *Balanceador de Carga*

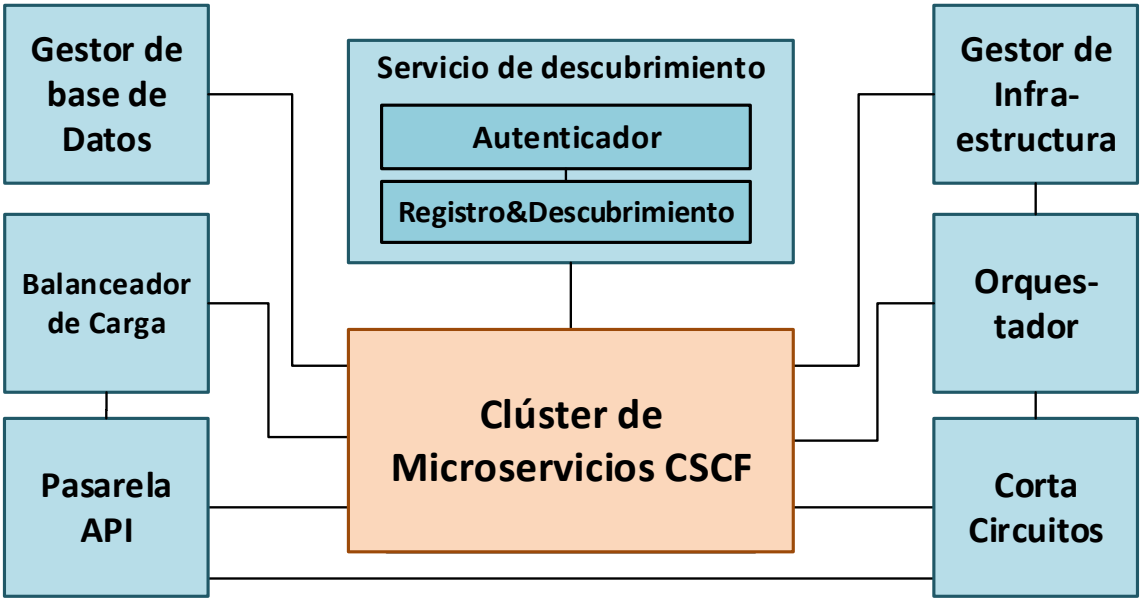

Figura 4.5: Elementos MSA

4.4.1 Servicio de Descubrimiento

El *Servicio de Descubrimiento* es un elemento de registro que soporta la comunicación de microservicios, está compuesto de dos componentes: *Registro&Descubrimiento* y *Autenticador*. El *Registro&Descubrimiento* almacena y proporciona direcciones de microservicio. Estas direcciones son URIs[3] que apuntan a una dirección IP necesaria para el reenvío de tráfico a los microservicios.

Para garantizar la seguridad en la provisión de direcciones de microservicios, se propone un nuevo componente en el *Servicio de Descubrimiento* llamado *Autenticador*. Este nuevo componente verifica que el elemento que solicita una dirección de microservicio es confiable. El *Autenticador* puede ser implementado de varias maneras. Por ejemplo, un sistema básico de verificación de claves y contraseñas.

4.4.2 Orquestador

Debido a que el Orquestador de MSA no está bien definido, se decidió adaptar funcionalidades de NFV-MANO que tiene una estandarización definida por el ETSI [56]. De esta manera, *Orquestador* adapta las funcionalidades de VNFM (*VNF Manager*) y NFVO (*NFV Orchestrator*). En este sentido *Orquestador* gestiona el ciclo de vida de los microservicios replicando, migrando, iniciando, pausando y eliminando instancias de microservicios. Logrando de esta manera una gestión de los recursos disponibles y adaptándose al tráfico entrante.

La labor del *Orquestador* es fundamental en la arquitectura, ya que este asegura una gestión centralizada de los microservicios transparente a la infraestructura. Esto permite crear instancias de microservicios interactuando directamente con el *Orquestador* y no con la máquina donde se ejecutará el microservicio. La no utilización de un *Orquestador* implicaría tener que gestionar manualmente el ciclo de vida de cada instancia de microservicio.

[3]Una *URI* es una secuencia de caracteres que identifican un recurso abstracto o físico [55]

4.4.3 Gestor de Infraestructura

El *Gestor de Infraestructura* adapta las funcionalidades de VIM (*Virtualized Infrastructure Manager*) de NFV-MANO [56]. Así, *Gestor de Infraestructura* interactúa directamente con los recursos de infraestructura (*i.e.,* computo, almacenamiento y *networking*) de cada máquina donde se despliega el *Clúster de Microservicios CSCF*. *Gestor de Infraestructura* recibe órdenes de gestión provenientes del*Orquestador* y las ejecuta en el *Clúster de Microservicios CSCF*. De esta forma, el *Gestor de Infraestructura* decide en qué máquina asignar un microservicio en función de los recursos disponibles.

Otra función del *Gestor de Infraestructura* es asegurar que las instancias de microservicios indicadas por el *Orquestador* se estén ejecutando en alguna de las máquinas. Por lo que si alguna de las máquinas falla o es eliminada por el administrador de red para ahorrar costos, el *Gestor de Infraestructura* debe ejecutar las instancias en alguna otra de las máquinas disponibles. Adicionalmente, otra de las funciones es la de asegurar que se cumplan políticas definidas para el despliegue de instancias de microservicio. Por ejemplo, en caso de que se decida que las instancias de los microservicios *Reenvío de mensajes SIP* y *Control de Sesiones Multimedia* no deben ser ejecutadas dentro de una misma máquina, el *Gestor de Infraestructura* deberá asegurar que se cumpla esa política.

4.4.4 Corta Circuitos

Corta Circuitos proporciona confiabilidad al *Clúster de Microservicios CSCF* manejando las fallas de las instancias de microservicios. Cuando se produce un fallo en el microservicio (*e.g., Timeouts* y respuestas HTTP/SIP 4xx/5xx), el *Corta Circuitos* lo registra. Después de que el número de fallos registrados por *Corta Circuitos* supera un umbral configurado por el administrador de red, *Corta Circuitos* determina que es necesario reiniciar la instancia de microservicio de fallos. Finalmente, *Corta Circuitos* indica al *Orquestador* cuales instancias de microservicios deben ser reiniciadas.

Si después de ejecutar el procedimiento anterior, el fallo persiste, el *Corta Circuitos* llega a la conclusión de que el fallo no se ha podido manejar con un reinicio de la instancia de microservicio. Por lo que le indica a la *Pasarela API* que detenga el tráfico externo e informa al administrador de la red sobre el fallo en la instancia del microservicio.

4.4.5 Pasarela API

La *Pasarela API* es un elemento que se encarga de impedir el acceso no autorizado al *Clúster de Microservicios CSCF* por parte de elementos externos a la arquitectura, y de ocultar la topología de red interna del μvIMS. Para asegurar que el usuario o entidad que está solicitando acceso a μvIMS es confiable la *Pasarela API* utiliza mecanismo de autenticación (*e.g.,* un sistema de clave valor). Después de asegurar que el solicitante es confiable, la *Pasarela API* distribuye el tráfico entre *Clúster de Microservicios CSCF* y el elemento o microservicio que solicita la dirección [57].

Para ocultar la topología de red, *Pasarela API* utiliza diferentes métodos, por ejemplo, proveer una dirección IP y un puerto externos a la arquitectura que el usuario utiliza para acceder al servicio solicitado. De esta manera el usuario no establece una conexión directa con los microservicios.

4.4.6 Balanceador de Carga

El *Balanceador de Carga* es un elemento que mejora las capacidades de la arquitectura. Su función principal es la de distribuir el tráfico destinado a un microservicio entre sus diferentes instancias. Este tráfico puede proveenir de elementos externos a la arquitectura como entidades de usuario o elementos internos como otros microservicios.

Para realizar la distribución del tráfico, el *Balanceador de Carga* puede utilizar varios algoritmos, como *round robin* [4] o *Greedy* [5]. El algoritmo utilizado guarda una

[4] *Round robin* divide el tráfico equitativamente entre servidores [58]
[5] *Greedy* determina el mejor destino posible basándose en unas condiciones específicas [59]

estrecha relación con el rendimiento de la arquitectura y pueden utilizarse algoritmos distintos dependiendo de las necesidades de la arquitectura.

4.4.7 Gestor de Base de Datos

Gestor de Base de Datos es un nuevo *Elemento MSA* propuesto en este trabajo específicamente para el diseño del μvIMS. El *Gestor de Base de Datos* tiene dos funciones principales. La primera es mantener una sincronización entre los datos de los microservicios. Esto es importante ya que algunos microservicios deben compartir información entre sí. Por ejemplo, el microservicio de *Control de Sesiones Multimedia* requiere la información registrada por el *Registro de Usuario*.

La segunda función del *Gestor de Base de Datos* es la de sincronizar la base de datos de los microservicios con el HSS. Es decir que si en el HSS externo se ingresó nueva información, esta debe ser actualizada en los microservicios del μvIMS. Es importante mencionar que el HSS no puede ser dividido en bases de datos de microservicios porque 5G y otras redes lo utilizan. Con este elemento, μvIMS no afecta a otras redes modificando el HSS.

4.5 Características de μvIMS

Las características de μvIMS son las siguientes:

- Está diseñado para proveer las funcionalidades del núcleo vIMS (CSCF).
- Provee las CSCF con 7 microservicios independientes entre sí.
- Permite asignar los recursos específicamente a los microservicios que manejan el tráfico entrante, logrando escalabilidad fina.
- Asegura una autonomía de gestión entre el ciclo de vida de los microservicios y la infraestructura de despliegue, utilizando el *Orquestador* y el *Gestor de Infraestructura*.

- Provee confiabilidad asegurándose que los microservicios estén funcionando correctamente, utilizando el *Corta Circuitos*.
- Se asegura que la dirección de los microservicios se provea solamente a elementos confiables, utilizando el *Autenticador* del *Servicio de Descubrimiento*.
- Oculta la estructura interna de la arquitectura utilizando la *Pasarela API*.
- No afecta el funcionamiento de otras redes de telecomunicaciones que utilizan el HSS, utilizando el *Gestor de Base de Datos* que sincroniza la información del HSS con las bases de datos de los microservicios.

4.6 Conclusiones Finales

Este Capítulo presentó el Subsistema Multimedia IP Virtualizado basado en microservicios llamado μvIMS que está compuesto de dos tipos de componentes: *Clúster de Microservicios CSCF* y *Elementos MSA*. El primero está compuesto de siete microservicios que realizan las CSCF. Los Elementos MSA proveen seguridad, fiabilidad y gestión a los microservicios.

- Las contribuciones son:
  - Diseño de Subsistema Multimedia IP Virtualizado basado en microservicios.
  - División funcional de las CSCF en siete microservicios independientes entre sí.

Capítulo 5

Evaluación del Subsistema Multimedia IP Virtualizado basado en Microservicios

5.1 Implementación de Referencia

Se utilizó Clearwater para implementar el *Clúster de Microservicios CSCF* modificando su arquitectura de acuerdo al diseño de μvIMS. Clearwater [42] es un núcleo IMS de código abierto, desarrollado por Metaswitch, ampliamente utilizado y diseñado para entornos de nube que sigue las interfaces del núcleo IMS estandarizadas por el 3GPP. Se escogió Clearwater como herramienta de implementación porque sus componentes están diseñados para ser fácilmente escalados horizontalmente, esta característica permite que al adaptar los componentes al diseño μvIMS, sea posible añadir instancias de microservicios. Además, Clearwater es la herramienta más utilizada en la literatura en trabajos en los que se utiliza vIMS.

Clearwater tiene dos tipos de despliegue, uno sobre VM y otro sobre contenedores Docker. La Tabla 5.1 muestra los componentes de ambos despliegues y la definición de cada uno. Los componentes Ellis, Bono, Sprout y Homer son asignados a una VM o a un contenedor respectivamente. Mientras que Dime se divide en Ralf, Home-

stead y Homestead-prov. Vellum se divide en Astaire, Cassandra, Chronos y Etcd. Cassandra y Etcd no son elementos propietarios de Clearwater, estos fueron modificados para añadirles la configuración de la arquitectura Clearwater por Metaswitch. Cassandra es una base de datos noSQL distribuida diseñada por *Apache Software Fundation* para soportar grandes volúmenes de datos [60]. Etcd es un sistema clave valor distribuido diseñado por *Cloud Native Computer Fundation* [61].

Table 5.1: División de Clearwater

Clearwater sobre VM	**Clearwater sobre Docker**	**Definicion**
Ellis	Ellis	Portal de provisión web utilizado para registro de usuarios, gestión de contraseñas y control de configuración de servicios.
Bono	Bono	Proxy de frontera SIP que realiza funciones del P-CSCF.
Sprout	Sprout	Proxy de enrutamiento, autorización y registro SIP que realiza funciones del I-CSCF y S-CSCF.
Homer	Homer	Servidor de gestión de documento XML que almacena configuración de servicios de usuario.
Dime	Ralf	Interfaz para facturación que provee una API HTTP para que Bono y Sprout reporten eventos de facturación.
	Homestead	Interfaz que provee información de perfil de usuario a Sprout a través de interfaces web.
	Homestead-prov	Interfaz que provee información de usuario a Ellis a través de interfaces web.
Vellum	Astaire	Servicio de Memcached que almacena estado de registro y sesión.
	Cassandra	Base de datos que almacena y provee información de usuario a Sprout a través de Homestead.
	Chronos	Servicio de temporizadores que permite a Sprout manejar eventos de larga duración.
	Etcd	Servicio coordinador distribuido de clave-valor utilizado para compartir información en un clúster, entre dos o más componentes.

Figura 5.1 muestra la interacción de los elementos de Clearwater desplegado sobre VM, sus protocolos y las interfaces para interactuar con otros elementos externos. Este despliegue de Clearwater sobre VM sigue un enfoque monolítico donde un solo

componente realiza las funciones de varios de los componentes de Clearwater sobre Docker.

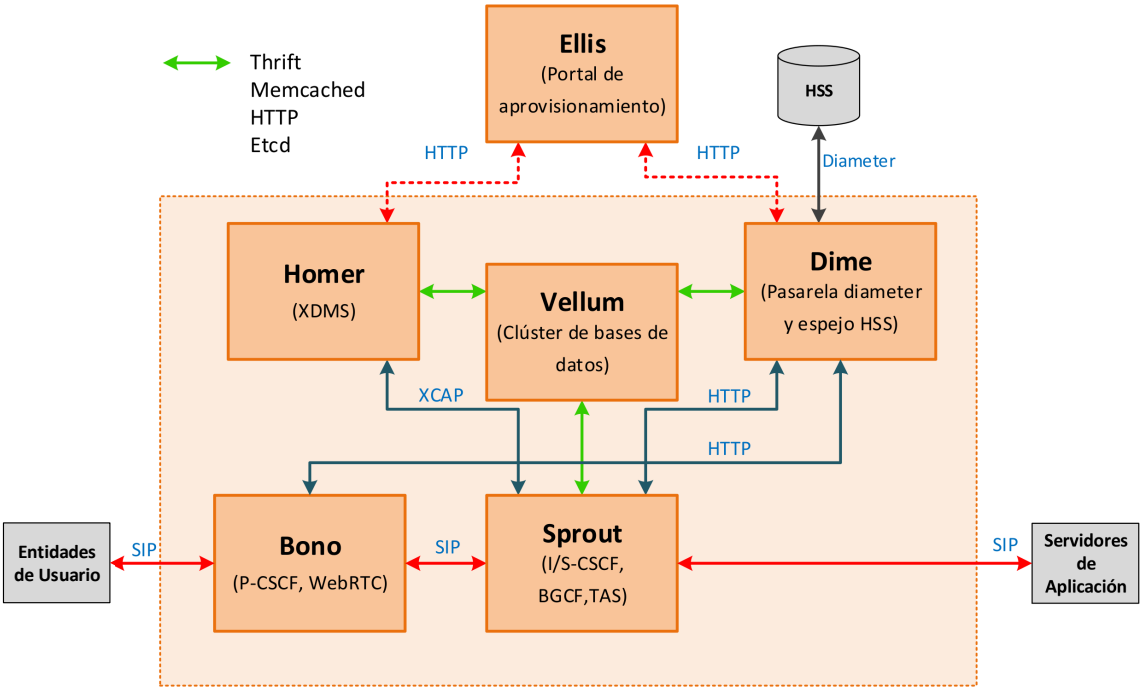

Imagen adaptada de http://www.projectclearwater.org/technical/clearwater-architecture/

Figura 5.1: Arquitectura de Clearwater sobre VM

Figura 5.2 muestra la interacción de los elementos de Clearwater desplegado sobre Docker. En este despliegue cada componente se implementa en un solo contenedor Docker. Además, estos componentes son similares a los microservicios del *Clúster de Microservicios CSCF*. Por lo tanto, en este trabajo de grado se adaptaron estos componentes utilizándolos o dividiéndolos para obtener el prototipo de μvIMS.

Para el control de sesiones multimedia y registro de usuarios, Clearwater sobre Docker utiliza Sprout, Homestead y Cassandra. Para mantener independencia entre los microservicios *Registro de usuarios* y *Multimedia Session Control*, se dividieron los componentes Sprout, Homestead y Cassandra. Sprout se dividió en URSprout y MSCSprout, Homestad se dividió en URHomestead y MSCHomestead, Cassandra se dividió en URCassandra y MSCCassandra. La adaptación de los microservicios del *Clúster de Microservicios CSCF* (Sección 4.3) con los componentes de Clearwater sobre Docker es la siguiente:

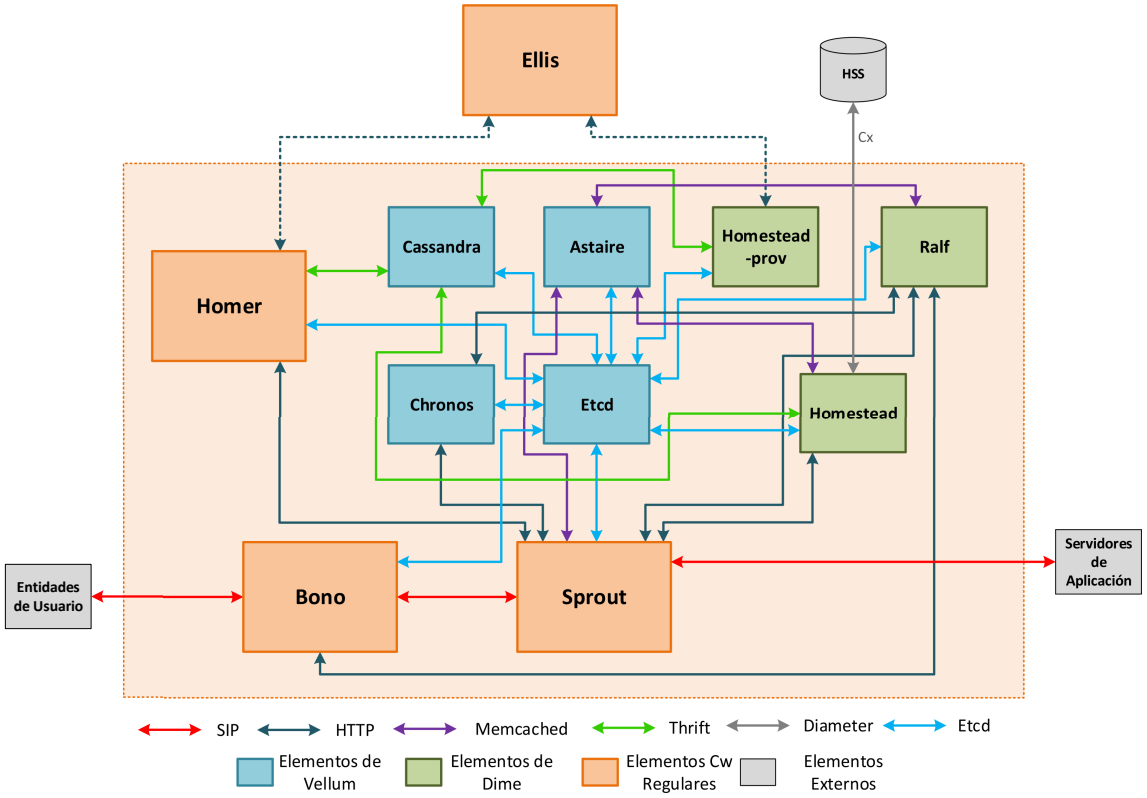

Figura 5.2: Arquitectura de Clearwater sobre Docker

- Para el microservicio *Forwarding SIP Messages* se utiliza Bono directamente.
- Para el microservicio *Manage CDRs* se utiliza Ralf directamente.
- Para el microservicio *Registro de usuarios* se utiliza URSprout, URHomestead, y URCassandra.
- Para el microservicio *Multimedia Session Control* se utiliza MSCSprout, MSC-Homestead y MSCCassandra.

Es importante aclarar que URCassandra y MSCCassandra sincronizan su información porque MSCSprout gestiona sesiones multimedia de usuarios registrados por URSprout en URCassandra. Para abordar esta sincronización, se implementó el *Gestor de Base de Datos* con Etcd para compartir la información de los usuarios, formando un clúster de Cassandras. Esta sincronización es instantánea al momento

que un nuevo dato es añadido en alguno de las instancias de URCassandra y MSC-Cassandra. Los nuevos datos son compartidos a todos miembros del clúster de Cassandra mediante el sistema clave valor distribuido de Etcd.

Figura 5.3 presenta el prototipo que implementa el *Clúster de Microservicios CSCF* y *Elementos MSA*. Para implementar los *Elementos MSA* se utilizó Kubernetes [62] con los cuales se busca proporcionar seguridad, fiabilidad y gestión de microservicios. Kubernetes es un orquestador de contenedores de código abierto ampliamente utilizado que para el caso de μvIMS es usado para gestionar el ciclo de vida de los microservicios del *Clúster de Microservicios CSCF*.

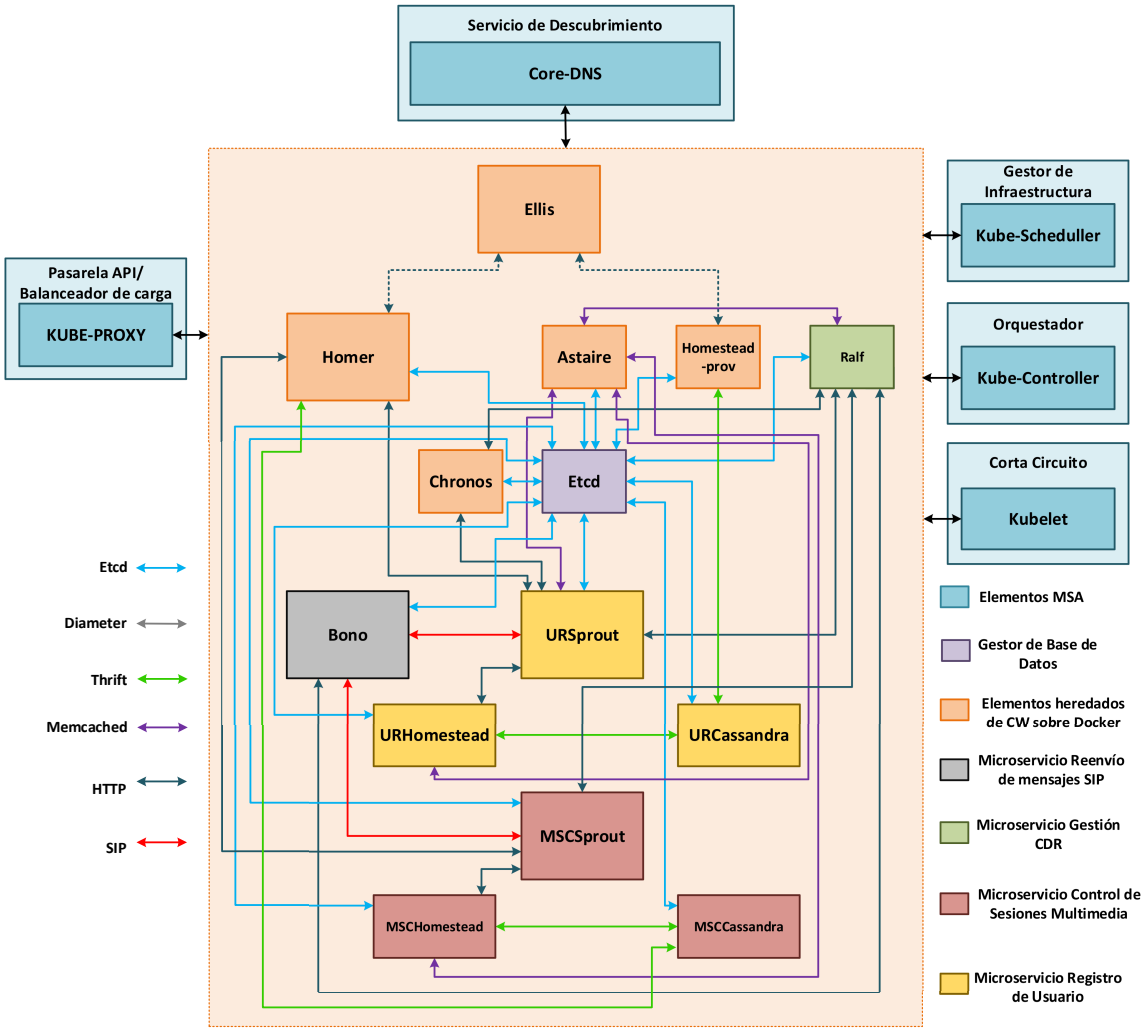

Figura 5.3: Prototipo de μvIMS

Kubernetes trabaja sobre una o más máquinas que forman un clúster Kubernetes. El clúster de Kubernetes está formado por un *Maestro Kubernetes* y varios *Esclavos Kubernetes*, donde el *Maestro Kubernetes* se encarga de la gestión de recursos (CPU y memoria) proporcionados por los *Esclavos Kubernetes* para desplegar sobre pods los componenetes de la adaptación de Clearwater sobre Docker. Un pod es la unidad más pequeña de Kubernetes que puede ser gestionada y manejada, en ella se ejecutan uno o más contenedores que se manejan como una sola entidad [63].

La implementación de los *Elementos MSA* con Kubernetes es la siguiente:

- Para el *Servicio de Descubrimiento* se utilizó el servidor DNS de Kubernetes llamado Core-DNS que realiza la funcionalidad de *Registro&Descubrimiento*.
- Para el *Orquestador* se utilizó el componente de Kubernetes llamado Kube-controller, el cual se asegura que los microservicios estén ejecutándose.
- Para el *Gestor de Infraestructura* se utilizó el Kube-scheduler que rastrea los recursos disponibles de los *Esclavo Kubernetes* y asigna microservicios en ellos.
- Para el *Corta Circuitos* se utilizó Kubelet en cada *Esclavo Kubernetes* que se asegura que los microservicios funcionan correctamente.
- Para el *Pasarela API* y *Balanceador de Carga* se utilizó el Kube-proxy que se ejecuta en cada *Esclavo Kubernetes* proporcionando acceso externo y distribuyendo la carga de trabajo de cada microservicio.

Figura 5.4 presenta el proceso de registro de un usuario que siguen los componentes del prototipo de μvIMS:

1. El usuario envía una petición de registro a Bono utilizando el protocolo SIP.
2. Bono se asegura que la petición tenga el formato correcto y la reenvía a URSprout.
3. URSprout requiere determinar si el usuario existe en la base de datos, para ello utiliza URHomestead con el que envía una petición a través del protocolo HTTP.

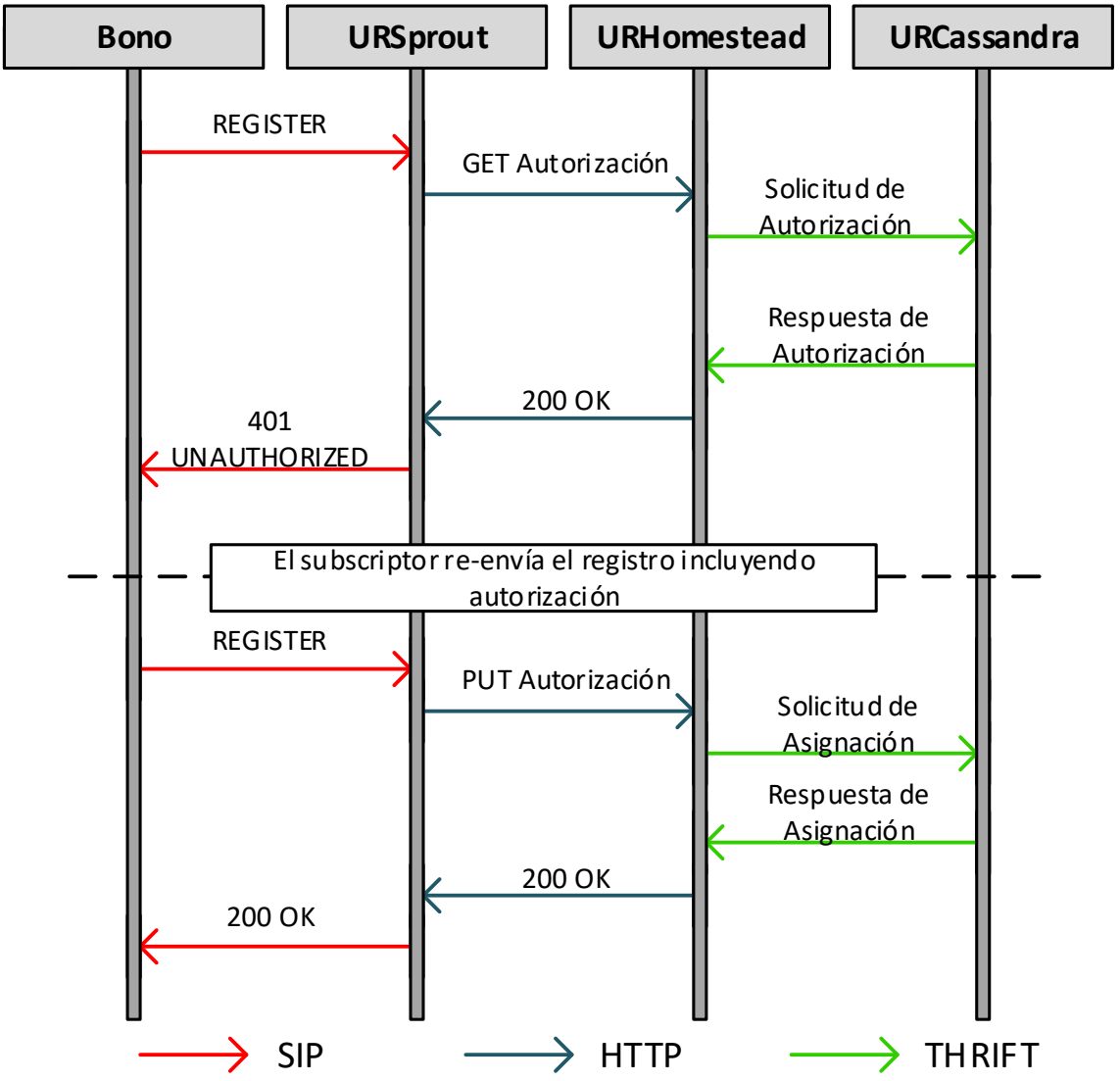

Figura 5.4: Registro de usuarios

4. URHomestead recibe la petición y se comunica directamente con URCassandra utilizando el protocolo Thrift.

5. URCassandra le responde a URHomestead que el usuario existe con Thrift [64].

6. URHomestead le notifica a URSprout que el usuario existe a través de HTTP.

7. URSprout le envía una confirmación a Bono de que el usuario existe con SIP,

y solicita la información de autorización.

8. Bono reenvía la confirmación al usuario, el cual envía su información de autorización.

9. Cuando el usuario manda la información de autorización a Bono esta se la reenvía a URSprout por SIP.

10. URSprout debe comprobar nuevamente en la base de datos que la información de autorización es correcta, para ello le envía una petición HTTP a URHomestead.

11. URHomestead se comunica por Thrift con URCassandra para solicitar la información de autorización del usuario, y se la envía a URSprout por HTTP.

12. URSprout determina si la información de autorización que envía el usuario corresponde a la que se tiene en la base de datos, si la información es correcta el usuario estará registrado en la arquitectura y URSprout le notifica a Bono por SIP que el registro fue exitoso.

13. Finalmente Bono le notifica al usuario que el registro se completó satisfactoriamente.

Figura 5.5 muestra el proceso para el establecimiento de sesión entre dos usuario que sigue el prototipo de μvIMS:

1. El usuario1 envía una petición de inicio de sesión con el usuario2 a Bono utilizando el protocolo SIP.

2. Bono se asegura que la petición tenga el formato correcto y la reenvía a MSCSprout.

3. MSCSprout le indica al usuario que está procesando la información.

4. MSCSprout le solicita a MSCHomestead la identidad pública del usuario1 por HTTP.

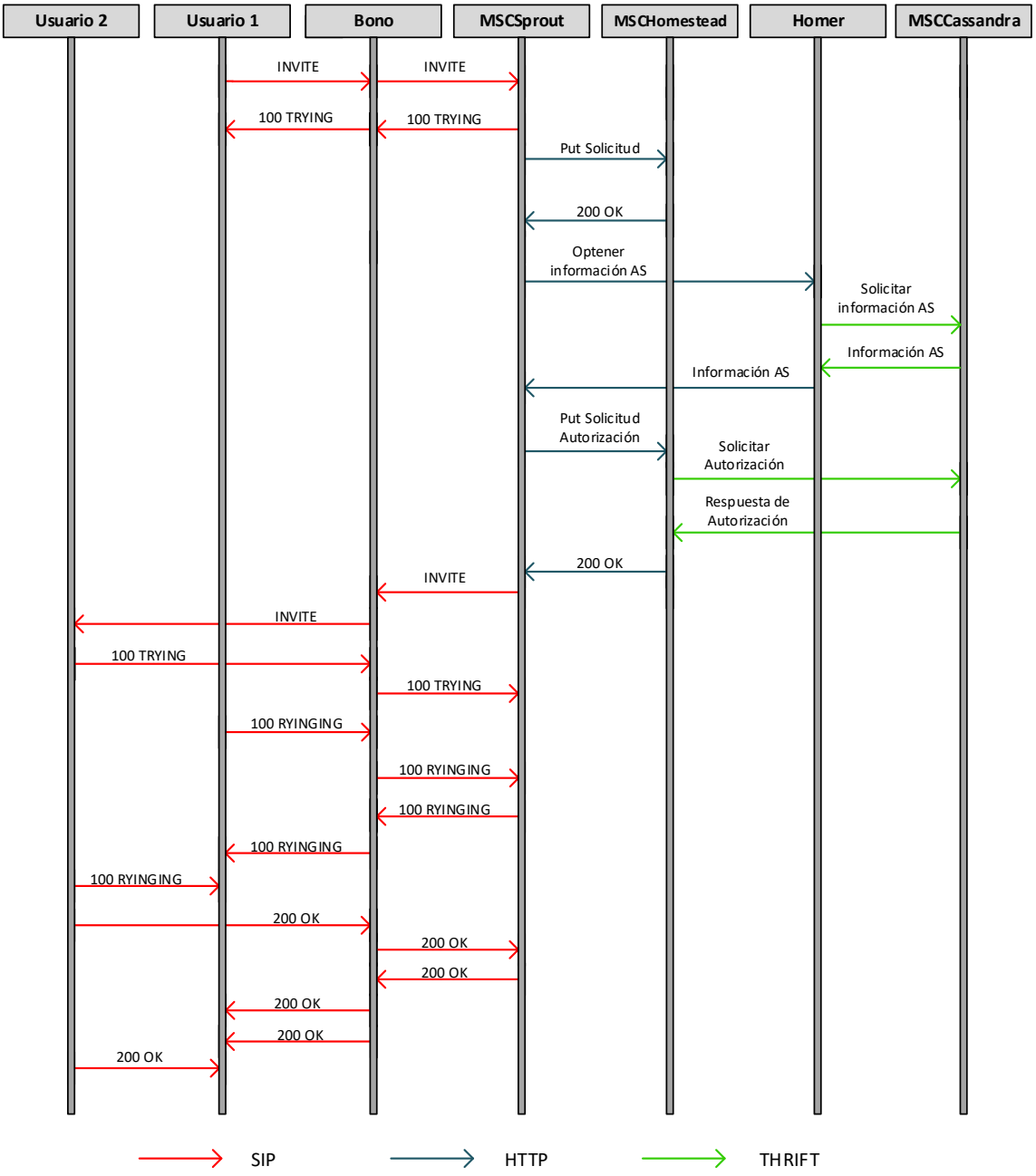

Figura 5.5: Inicio de sesión

5. MSCSprout le solicita a Homer información de los AS que están configurados para el usuario1 a través de HTTP.

6. Homer solicita la información de los AS configurados al usuario1 a MSCCas-

sandra utilizando Thrift.

7. MSCSprout solicita la identidad pública del usuario2 a MSCHomestad. En caso que no se tenga ningún AS configurado al usuario1 y la sesión consista solamente en la comunicación entre dos usuario.

8. MSCHomestead consulta la identidad pública del usuario2 a MSCCassandra a través de Thrift y se la reenvía a MSCSprout por HTTP.

9. Conociendo la identidad pública del usuario2, MSCSprout envía una solicitud de inicio de sesión a Bono el cual se la reenvía al usuario 2 por SIP.

10. Mientras el usuario2 responde, se le envía una notificación al usuario1 en la cual se le informa que se está tratando de contactar con el usuario2 a través de Bono y MSCSprout utilizando SIP.

11. Cuando el usuario2 responde, MSCSprout y Bono reenvían la confirmación al usuario1 y la sesión entre los dos usuarios inicia.

5.1.1 Implementación de Servidor de Aplicaciones

A continuación presenta la interacción de la arquitectura μvIMS con un AS. IMS cuenta con la interfaz ISC Para realizar interacciones con AS. La manera como se interactúa con los AS es la misma que con una entidad de usuario, utilizando el protocolo SIP. Adicionalmente, para que la arquitectura sepa que AS debe ser activado se utiliza el IFC (*Initial Filter Criteria*) cuya configuración está alojada junto a la información de cada usuario. Los IFC se incluyen en formato XML como está descrito en la TS 29.28 del 3GPP [65]. Con el IFC se configura la interacción con el AS dependiendo de varios factores, por ejemplo, el tipo de mensaje SIP o la extensión a la que se está llamando. Tabla 5.2 presenta herramientas con las que se pueden implementar AS [66].

Para el prototipo de μvIMS se decidió utilizar Asterisk como herramienta para implementar AS. Debido a que es la herramienta con documentación más detallada que

Table 5.2: Herramientas para implementar AS

Servidor de Aplicacion	Sitio Web	Descripción
Asterisk	https://www.asterisk.org/	Framework libre y abierto para construir aplicaciones de telecomunicaciones.
Mobicents	http://mobicents.sourceforge.net	AS orientado a eventos altamente escalable.
SaRP	http://sarp.sourceforge.net/	Proxy RTP y SIP diseñado para manejar los problemas heredados de NAT y SIP.
Yate	http://www.yate.ro/	Servidor de telefonía utilizado para VoIP y telefonía fija.
Wildfly	https://wildfly.org/	AS desarrollado en java flexible y ligero que ayuda a construir aplicaciones.
Kamailio	https://www.kamailio.org/w/	Servidor SIP de fuente abierta capaz de manejar miles de establecimientos de llamadas por segundo.
Payara	https://www.payara.fish/	AS de fuente abierta y amigable para el desarrollo de aplicaciones java.

permitió desplegar servicios de manera ágil. Utilizando Asterisk se configuró el servicio de *VoiceMail*, utilizando el IFC mostrado en el Anexo C. *Voicemail* permite que si una llamada no es contestada o la línea está ocupada se ponga en funcionamiento una contestadora, que permitirá al usuario que está generando la llamada dejar un mensaje de voz. El usuario al que se llamó puede acceder a todos los mensajes de voz mediante la marcación a número de buzón configurado.

Para la configuración del AS desde el μvIMS se utiliza Ellis mediante línea de comandos y su interfaz gráfica. El proceso de configuración del AS y establecimiento de sesión con dicho elemento desde el prototipo es el siguiente:

1. Se añade el IFC del AS dentro de Ellis utilizando la línea de comandos para

que el servicio aparezca disponible a todos los usuarios.

2. Con la interfaz gráfica de Ellis se activa el servicio para el usuario. De manera que el usuario ya lo pueda utilizar.

3. Se establece una sesión multimedia desde o hacía el usuario.

4. Por medio de Homer MSCSprout revisa si los usuarios tienen un AS activo, de ser así se le hace saber al MSCSprout.

5. En caso de que la sesión cumpla con el IFC configurado, se establece comunicación con el AS. Con lo cual MSCSprout reenvía el último mensaje recibido desde el usuario al AS correspondiente para ejecutar el servicio.

5.2 Ambiente de Pruebas

En esta Sección, se presenta el entorno de prueba utilizado para evaluar el prototipo μvIMS. El ambiente de pruebas fue desplegado sobre la plataforma Telco 2.0. Esta plataforma es una red de telecomunicaciones convergentes, la cual está diseñada para proveer los siguientes servicios:

1. Investigación y desarrollo. La red está diseñada para permitir el desarrollo de redes y servicios telemáticos.

2. Servicios Multimedia Combinados. La red está diseñada para ofrecer capacidades multimedia sobre redes convergentes fijo–móvil.

3. Independencia del tipo de acceso. La red está diseñada para que su acceso sea independiente de la red o terminal.

4. Aplicaciones WEB y servicios telemáticos. La red está diseñada para permitir la integración entre tecnologías de la información y las telecomunicaciones, redes definidas por software y NFV.

Table 5.3: Recursos del ambiente de pruebas

<table>
<tr><th>Máquina</th><th>Recursos</th><th>vIMS</th><th>μvIMS</th></tr>
<tr><td>Máquina1</td><td rowspan="6">4 Intel(R) Xeon(R) CPU E5-2670 2.30 GHz and 16 GB memoria</td><td>Ellis</td><td>Maestro Kubernetes</td></tr>
<tr><td>Máquina 2</td><td>Bono</td><td rowspan="6">Esclavo Kubernetes (Clúster Microservicios CSCF)</td></tr>
<tr><td>Máquina 3</td><td>Sprout</td></tr>
<tr><td>Máquina 4</td><td>Homer</td></tr>
<tr><td>Máquina 5</td><td>Dime</td></tr>
<tr><td>Máquina 6</td><td>Vellum</td></tr>
<tr><td>Máquina 7</td><td>4 Intel(R) Xeon(R) CPU E5-2670 2.30 GHz and 6 GB memoria</td><td>Bind9</td></tr>
<tr><td>Máquina 8</td><td>4 Intel(R) Xeon(R) CPU E5-2670 2.30 GHz and 16 GB memoria</td><td colspan="2">SIPp</td></tr>
</table>

Para determinar el rendimiento de μvIMS, se asigno el mismo número de máquinas con la misma cantidad de recursos de la plataforma Telco 2.0 a dos despliegues: el prototipo μvIMS y un vIMS desplegado con Clearwater sobre VM. La Tabla 5.3 muestra la asignación de recursos de ambos despliegues.

La Figura 5.6 presenta el despliegue de μvIMS que consiste en siete máquinas que conforman un clúster de Kubernetes. Este clúster está compuesto por un Maestro Kubernetes y seis Esclavos Kubernetes. Además, se tiene una octava máquina para generar tráfico. Sobre los seis Esclavos Kubernetes, se despliega el *Clúster de Microservicios CSCF*. En el Maestro Kubernetes se despliega el *Servicio de Descubrimiento*, el *Orquestador* y el *Gestor de Infraestructura*. De esta manera Maestro Kubernetes decide en qué Esclavo Kubernetes asigna cada microservicio.

La Figura 5.7 presenta la implementación de vIMS que incluye seis máquinas. Estas máquinas despliegan cada uno de los componentes de Clearwater sobre VM: Ellis, Bono, Sprout, Dime, Homer y Vellum. Estos componentes necesitan un DNS para comunicarse entre sí. Así pues, se implementó en una séptima máquina usando Bind9 [67]. Finalmente, vIMS tiene una octava máquina para generar tráfico. En ambas implementaciones, la octava máquina utiliza SIPp [68] como herramienta de generación de tráfico. Se seleccionó SIPp como herramienta dado que es el generador

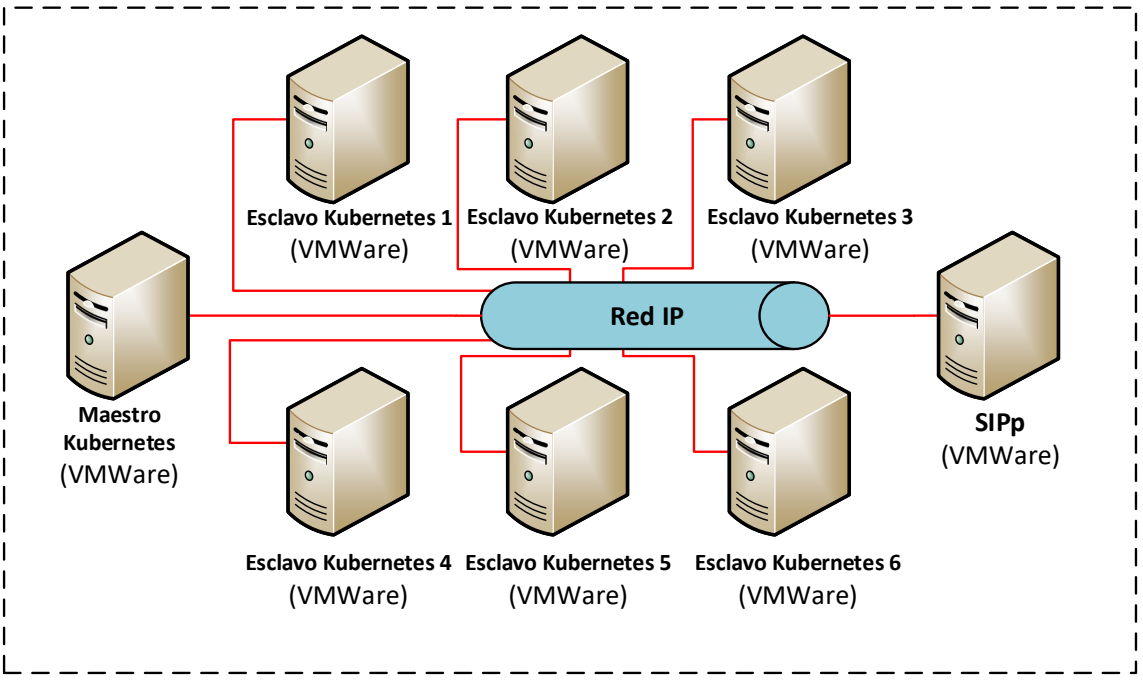

Figura 5.6: Despliegue de μvIMS

de tráfico SIP más utilizado en la literatura para realizar evaluaciones de IMS.

Figura 5.8 presenta el escenario de prueba utilizado para comparar ambas implementaciones, este escenario utiliza SIPp con tres fases: registro de dos usuarios, establecimiento de llamada y fin de llamada entre ellos. Los dos usuario se simulan con SIPp, y acceden al prototipo a través de Bono. La interacción entre los dos usuarios configurada para el escenario es la siguiente:

1. Se registra el usuario1 siguiendo el proceso normal, donde inicialmente se comprueba que el usuario1 está en la base de datos y después se le solicita la información de autorización.

2. Se registra el usuario2 siguiendo el proceso normal, donde inicialmente se comprueba que el usuario2 está en la base de datos y después se le solicita la información de autorización.

3. Se tiene un retardo que simula el tiempo entre el registro de los usuarios e inicio de la solicitud de llamada por parte del usuario1 al usuario2.

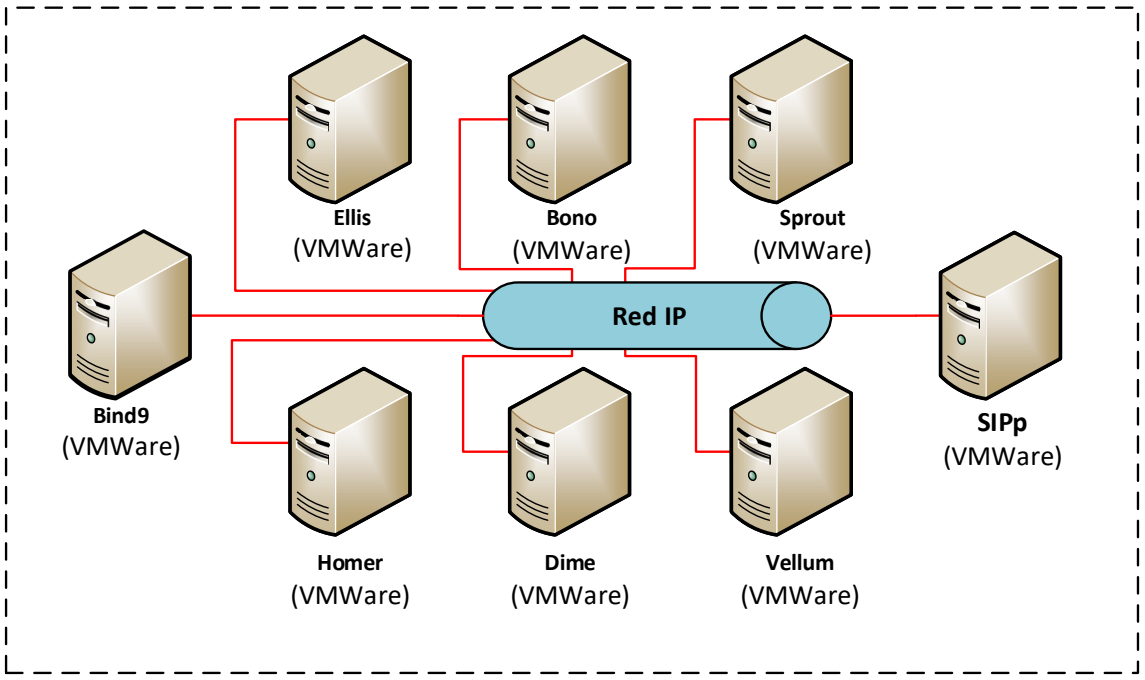

Figura 5.7: Despliegue de vIMS (Clearwater sobre VM)

4. El usuario1 envía una petición de inicio de sesión a μvIMS el cual la procesa y la reenvía al usuario2.

5. Puede ocurrir que primero llegue al usuario1 la notificación de que su solicitud se está procesando y después llegue la petición de inicio de sesión al usuario2 (opción 1), o viceversa (opción 2). El escenario de prueba considera ambas opciones.

6. El usuario2 responde con un ringing, el cual se reenvía al usuario1. Además, se simula una pausa correspondiente al tiempo que se demora el usuario2 en responder.

7. Cuando el usuario2 responde se envía una confirmación al usuario1.

8. Se intercambia parámetros de la sesión multimedia (*e.g.*,, tipo de flujo y codecs) entre los usuarios y la sesión inicia.

9. Se tiene un retardo que simula la duración de la llamada.

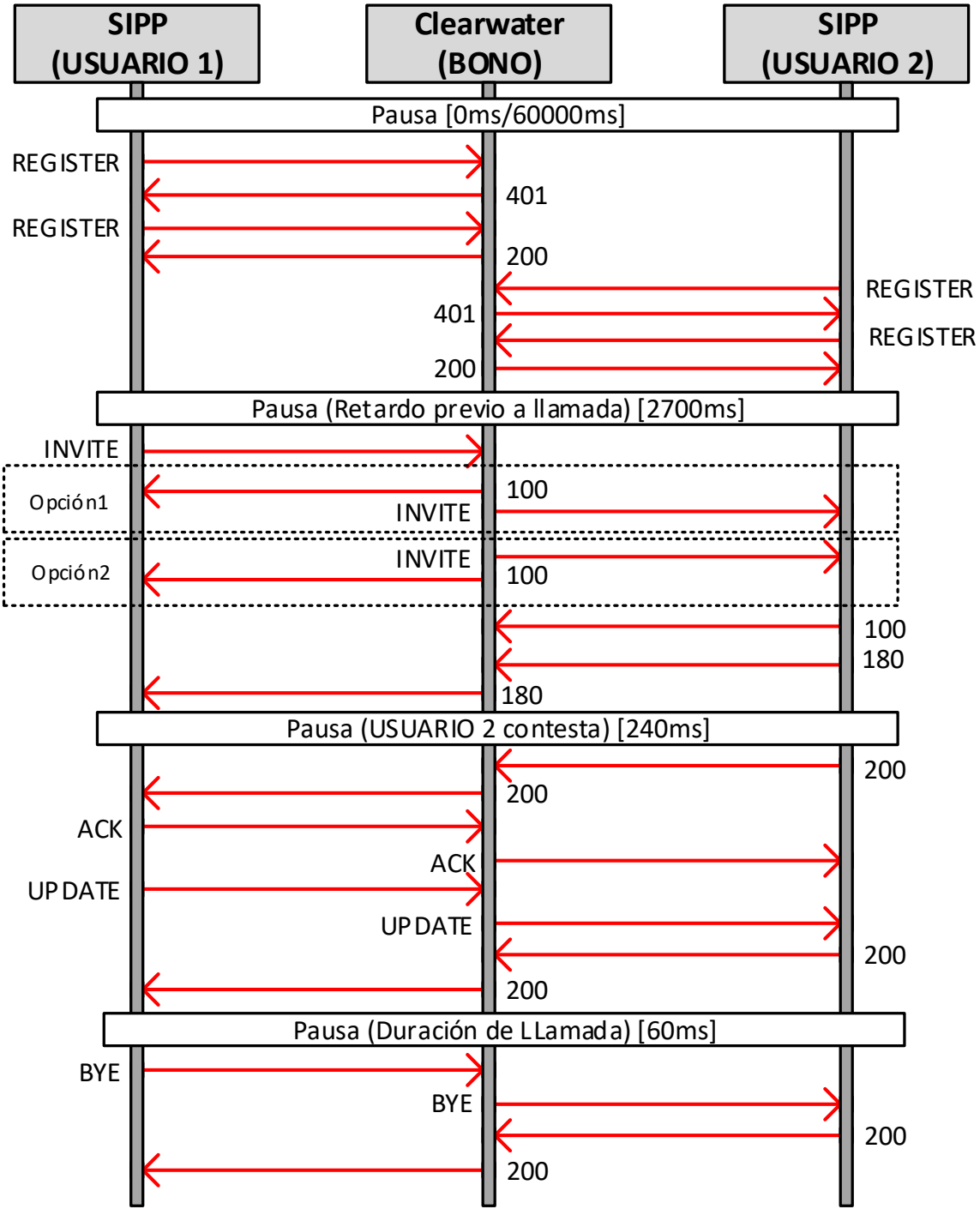

Figura 5.8: Escenario de prueba

10. El usuario1 envía una solicitud para finalizar la llamada a μvIMS, esta se reenvía al usuario2 el cual la confirma y la llamada finaliza.

11. Se tiene un retardo que simula el tiempo después de la llamada.

El escenario de prueba anteriormente descrito se repite para varios pares de usuarios al mismo tiempo, dependiendo de las CPS definidas para probar los despliegues, debido a que más usuarios implican más CPS. Se aumentó las CPS de 25 a 200 en intervalos de 25 CPS, porque estos intervalos ofrecen un balance entra la duración total de la prueba y la precisión para medir SCR. Si los intervalos son muy pequeños la prueba total tendrá una duración de varios días modificando el cronograma. Si los intervalos son muy grandes la diferencia de SCR entre dos intervalos consecutivos sería muy grande, y la aproximación de cuantos CPS soportan los despliegues sería inexacta.

Las parejas de usuarios definidas para un CPS particular se distribuyen a lo largo de 60 segundos utilizando una distribución uniforme para simular una carga constante. El escenario de pruebas se repitió 32 veces en igualdad de condiciones para promediar los resultados. Se emuló este proceso en ambos despliegues, y las métricas que se utilizaron para compararlos son SCR, uso de recursos (CPU y memoria de las siete máquinas que conformaron las implementaciones), y latencia.

Figura 5.9 presenta como se mide el consumo de recursos de ambos despliegues. Con el propósito de una evaluación equitativa, se midió el consumo de total de CPU y memoria de las siete máquinas que componen cada despliegue, excluyendo la octava máquina que genera las pruebas. La monitorización se puede realizar desde cualquiera de las máquinas de la arquitectura, a la máquina donde se ejecuta la monitorización se le denomina máquina de prueba. Para vIMS es la máquina 8 mientras que para el μvIMS es la máquina 1. El proceso es el siguiente:

1. Se envía una orden a las máquinas para que creen un repositorio local donde se almacena la información del consumo de CPU y memoria localmente para unas CPS, y número de prueba específica. Esta orden se envía a las máquinas utilizando SSH.

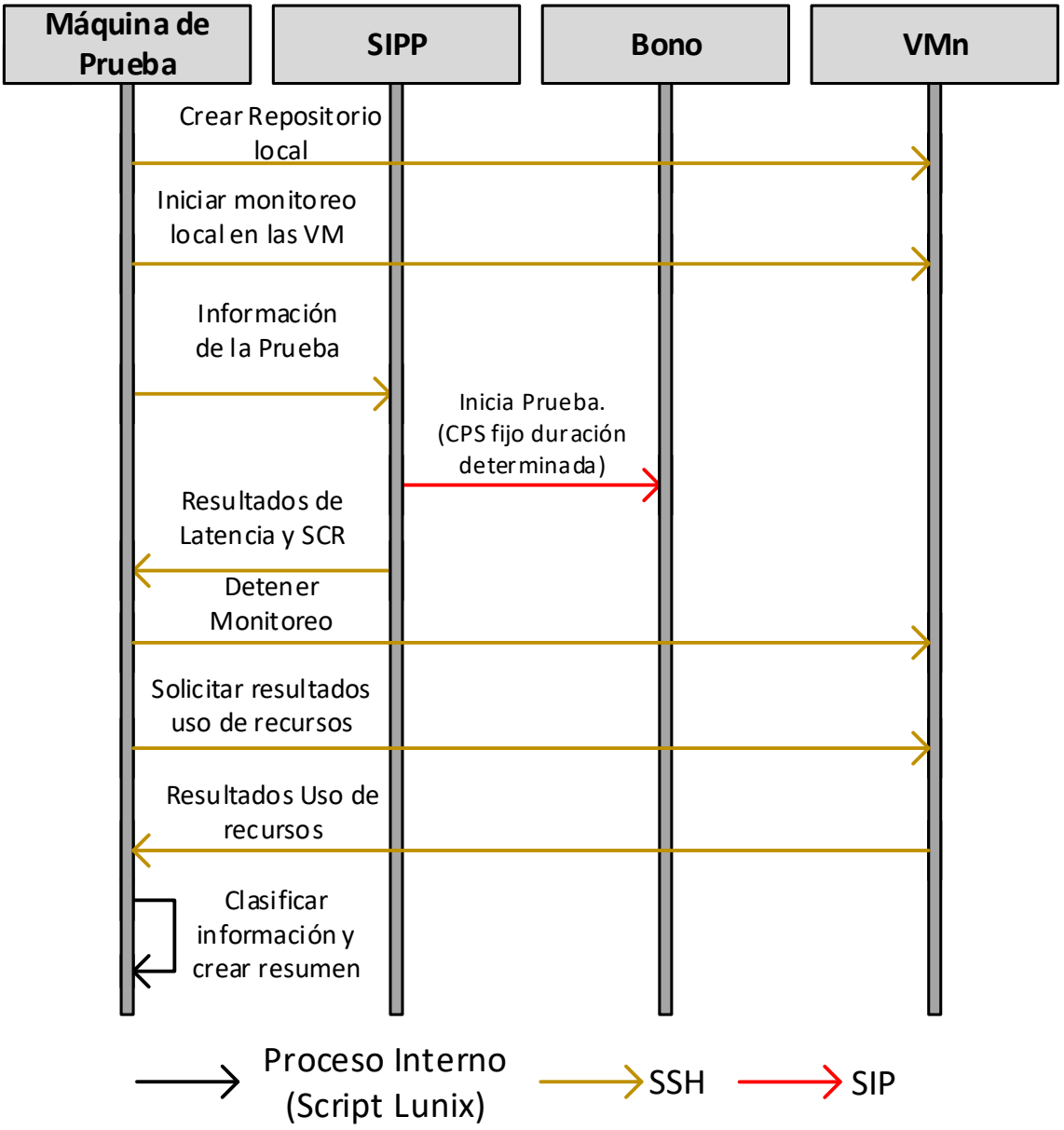

Figura 5.9: Flujo de monitorización de la arquitectura

2. Se envía una orden a las máquinas para que inicien la monitorización de recursos. Esta orden se envía a las máquinas utilizando SSH.

3. Se envía la información de la prueba (CPS) a la octava máquina que tiene SIPp para que inicie el escenario de prueba.

4. SIPp envía el tráfico de la prueba por SIP a Bono, el cual lo reenviará a los

demás microservicios siguiendo los diagramas para registro e inicio de sesión descritos en la Figura 5.4 y Figura 5.5 respectivamente.

5. Cuando la prueba finaliza, la octava máquina envía la información de SCR y latencia a la máquina de prueba.

6. La máquina de prueba le envía una orden por SSH a las máquinas de los despliegue para que detengan la monitorización de recursos.

7. Las máquinas le envían a la máquina de prueba los datos del consumo de recursos.

8. Por último la máquina de prueba clasifica la información que recibió y realiza un resumen del consumo general de recursos de la implementación que se esté probando.

5.3 Resultados y Análisis

5.3.1 Caracterización

Para caracterizar μvIMS, se analizó el impacto de dividir Sprout, comparando un despliegue de Sprout dividido y de Sprout sin dividir utilizando el escenario de prueba descrito en la Sección 5.2. El despliegue de Sprout dividido se refiere al prototipo de μvIMS, donde se divide al Sprout en URSprout y MSCSprout, cada uno con su propio Homestead y Cassandra (Sección 5.1). El despliegue de Sprout sin dividir se refiere a las entidades que define Clearwater sobre Docker. Es importante aclarar que en el despliegue de Sprout sin dividir también se utilizó Kubernetes para que ambas arquitecturas tuvieran el mismo ambiente de despliegue y las mismas condiciones.

Figura 5.10 presenta los resultados de la comparación de cada despliegue considerando la SCR alcanzada por cada uno. Los resultados indican que cuando se divide Sprout se alcanza una mayor SCR. Esto ocurre porque el Maestro Kubernetes puede asignar los recursos de un Esclavo Kubernetes a URSprout y los de otro a

MSCSprout. Estas entidades se encargan de realizar el registro y control de sesión, por ello necesitan una mayor cantidad de recursos.

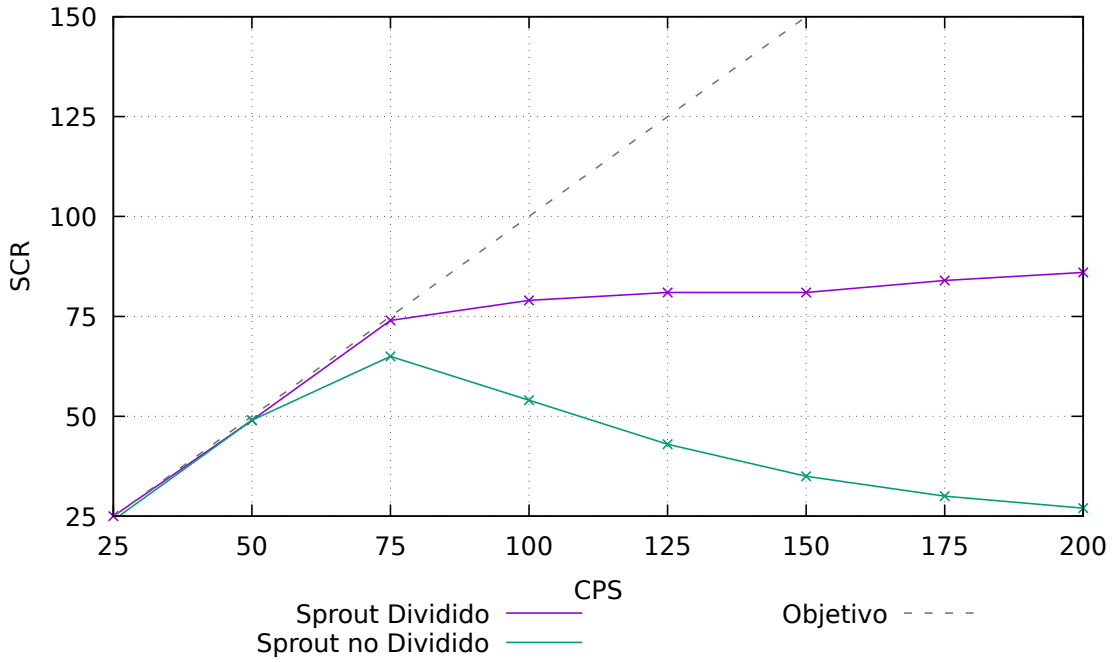

Figura 5.10: Resultados: SCR de Sprout Dividido y Sin Dividir

Al despliegue de Sprout sin dividir, debido a su naturaleza monolítica, solamente se le puede asignar a Sprout los recursos de un Esclavo Kuberentes, este componente se encarga tanto del registro como del control de sesión. Por lo tanto, inicialmente el SCR es similar para ambos despliegues, pero cuando la carga (CPS) empieza a aumentar el despliegue con Sprout sin dividir se sobrecarga y el rendimiento empeora rápidamente.

Después de determinar que dividir Sprout implica una mejora al momento de asignar recursos para atender más usuarios, se analizó qué componente específicamente necesitan más recursos. Para identificarlos, se ejecutó nuevamente el escenario de prueba en el prototipo de μvIMS, en esta ocasión no se midió los recursos de todas las máquinas, sino el consumo de CPU y de memoria de los pods.

Figura 5.11 presenta los cinco componentes que consumen más CPU de μvIMS. Se observa que el componente que más consume CPU es Bono encargado de enrutar las

peticiones SIP sin importar si son de registro o control de sesión. Le siguen URSprout y MSCSprout encargados de realizar el registro y control de sesión respectivamente.

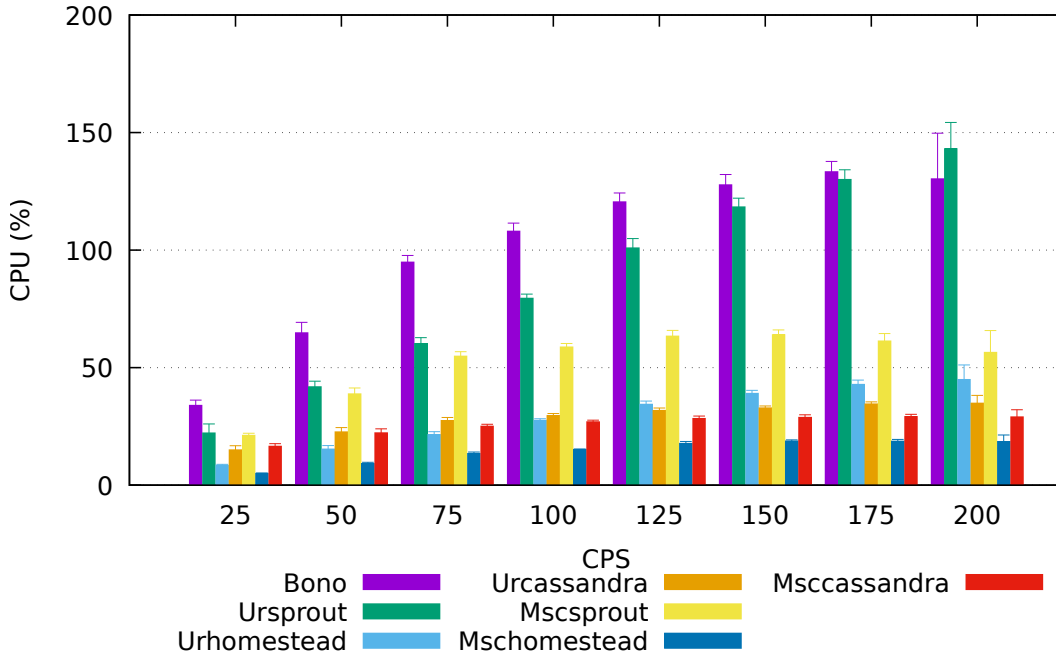

Figura 5.11: Resultados: Consumo de CPU de componentes del prototipo μvIMS

Figura 5.12 presenta los cinco componentes que consumen más memoria de μvIMS. Se observa que el consumo de memoria general se mantiene sin importar que aumenta la carga del sistema. Además, el componente que más consume memoria es URCassandra debido a que guarda las identidades de los usuarios para el control de sesión. Considerando que cada máquina tiene entre 14 Gb y 16 Gb de memoria, el consumo que tienen los componente es despreciable, y por consiguiente se considera que el CPU tiene un mayor impacto en el rendimiento que la memoria.

Considerando los resultados de la caracterización, se determina que los componentes que más recursos consumen (priorizando el CPU) son Bono, URSprout y MSCSprout. Por lo tanto, para probar la escalabilidad del sistema se probaron distintas combinaciones de estos componentes escalados horizontalmente. La Tabla 5.4 muestra el número de instancias de cada componente desplegadas en cada combinación.

- La Combinación1 también denominada despliegue base tiene una instancia de

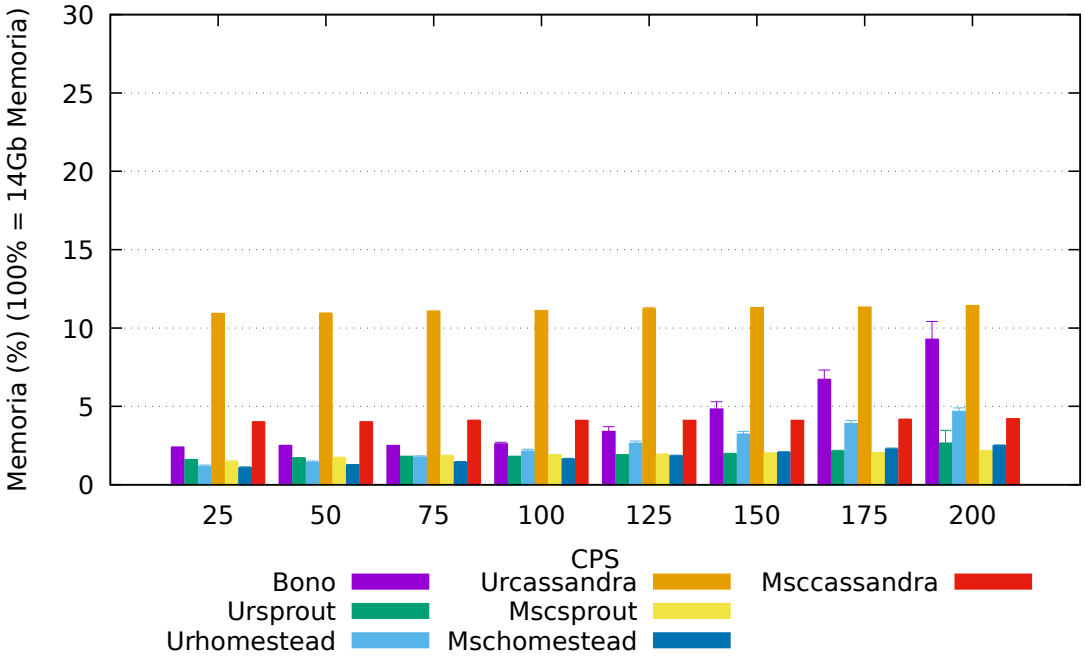

Figura 5.12: Resultados: Consumo de memoria de componentes del prototipo μvIMS

Table 5.4: Combinaciones prototipo μvIMS

Despliegue	Bono (Instancias)	URSprout (Instancias)	MSCSprout (Instancias)
Combinación 1	1	1	1
Combinación 2	2	1	1
Combinación 3	1	2	1
Combinación 4	1	1	2
Combinación 5	3	1	1
Combinación 6	2	2	1
Combinación 7	2	1	2

todos los microservicios.

- La Combinación2 tiene dos instancias de Bono y una instancia de los demás microservicios.
- La Combinación3 tiene dos instancias de URSprout y una instancia de los demás microservicios.

- La Combinación4 tiene dos instancias de MSCSprout y una instancia de los demás microservicios.
- La Combinación5 tiene tres instancias de Bono y una instancia de los demás microservicios.
- La Combinación6 tiene dos instancias de Bono y URSprout, y una instancia de los demás microservicios.
- La Combinación7 tiene dos instancias de Bono y MSCSprout, y una instancia de los demás microservicios.

El escenario de prueba se ejecutó para cada combinación de μvIMS y para la implementación de vIMS. Mientras la prueba estaba en ejecución, se midió el SCR alcanzado por cada combinación para cada CPS, así como el uso de recursos de todas las máquinas que ejecutan el clúster (CPU y memoria). Finalmente, se realizó una evaluación de Latencia de las combinaciones de despliegue μvIMS y vIMS. Para la evaluación de latencia se consideró el tiempo total desde que se envía un mensaje hasta que se recibe su respuesta [51], para ello se utilizaron temporizadores en el escenario definido con SIPp. Después de recolectar la información, se evaluó la latencia agrupando los resultados de todos los CPS en una CDF [52]. Se consideró la latencia de vIMS y cada combinación de μvIMS, determinando si no se supera el límite de latencia para la señalización IMS de 100 ms.

5.3.2 Tasa de Llamadas Exitosas

La Figura 5.13 muestra los resultados de la evaluación del SCR de vIMS y las siete combinaciones μvIMS. A continuación se hace un análisis de los resultados para cada combinación:

1. Combinación 1: con esta combinación μvIMS presenta un SCR similar al vIMS, siendo el SCR de μvIMS ligeramente más alto.

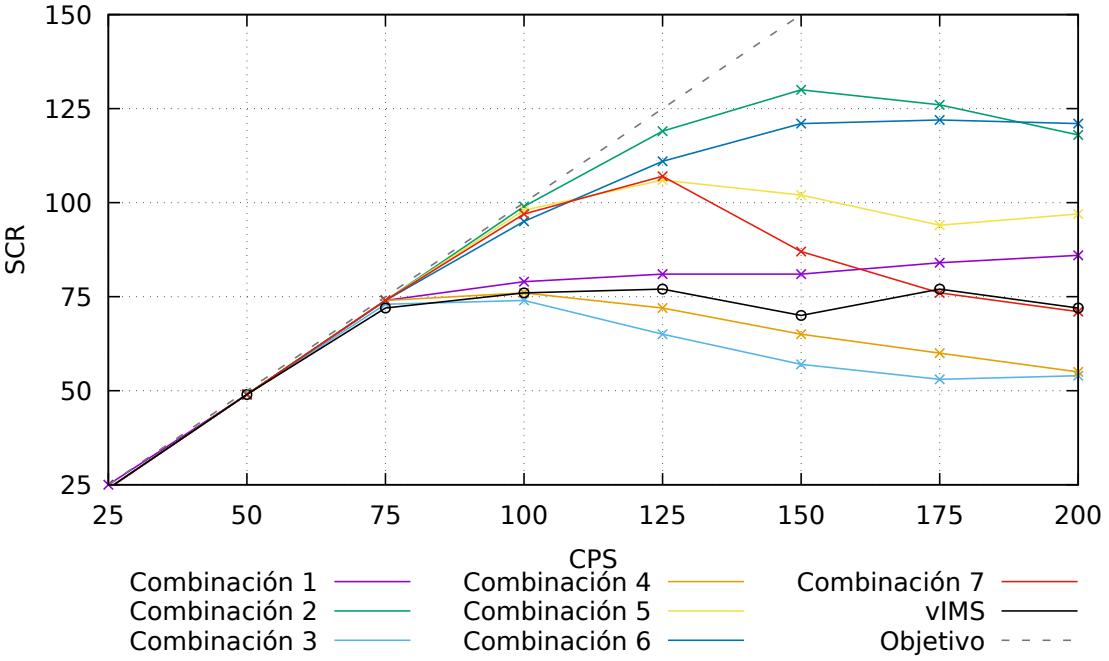

Figura 5.13: Resultados: SCR

2. Combinación 2: con esta combinación los resultados del SCR son mejores que con la Combinación 1. Además, estos son mejores que los resultados de SCR obtenidos por vIMS.

3. Combinación 3 y Combinación 4: los resultados obtenidos con estas combinaciones son peores que los de vIMS. Esto contrastado con el resultado de la Combinación 2, indica que inicialmente es pertinente escalar Bono antes que MSCSprout o URSprout. Lo cual se debe a que Bono, al ser el punto de entrada de la arquitectura, debe atender más tráfico que URSprout y MSCSprout.

4. Combinación 5: los resultados con esta combinación se mantienen por encima de los de vIMS, pero a su vez conllevan una degradación en el resultado comparado con la Combinación 2. Esto implica que a este punto del escalado de la arquitectura no es pertinente añadir una instancia extra de Bono.

5. Combinación 6: los resultados de esta combinación son mejores que los obtenidos con la Combinación 5 y similares a los que se obtuvieron con Combinación 2, siendo los de la Combinación 2 mayor para todos los CPS excepto para 200 en

donde el resultado es ligeramente mejor.

6. Combinación 7: los resultados son peores que los de la Combinación 2 y 6. Además, para CPS superiores a 175 los resultados son similares a los de vIMS.

Se concluye que las combinaciones 2 y 6 tienen un SCR mayor que el vIMS. El resultado más alto en toda la evaluación fue el de la Combinación 2 manteniendo aproximadamente 125 CPS atendidas correctamente, aproximadamente 66% más alto que el de vIMS que mantiene un valor promedio de 75 CPS. Demostrando la factibilidad de μvIMS para atender más llamadas con éxito con los recursos disponibles, utilizándolos de manera efectiva. Es importante mencionar que los resultados indican que las combinaciones 2 y 6 tienen SCR mayor para el ambiente de pruebas específico con el que se probó μvIMS (Sección 5.2). Por lo tanto, en ambientes con un número de máquinas y recursos asignados diferentes, las combinaciones más óptimas pueden variar. Sin embargo, utilizando microservicios se puede encontrar fácilmente una combinación óptima para mejorar el rendimiento de la arquitectura independientemente del escenario donde se despliegue.

5.3.3 Consumo de Unidad de Procesamiento Central

La Figura 5.14 presenta los resultados de la evaluación del uso de CPU de las siete combinaciones μvIMS y de vIMS. Estos resultados revelan lo siguientes:

- La diferencia entre las combinaciones de μvIMS para CPS bajos (*i.e.,* 25, 50 y 75 CPS) es despreciable, para estos CPS aproximadamente todas las llamadas fueron exitosas.
- El uso de CPU de vIMS para 25, 50, 75, 100 y 125 CPS es inferior a las combinaciones de μvIMS. Sin embargo, la evaluación muestra que para 150. 175 y 200 CPS, el uso de CPU de vIMS aumenta a un nivel similar al de las combinaciones de μvIMS, siendo la diferencia entre los resultados despreciable.
- Los resultados de SCR y CPU muestran que un mayor número de CPS conlleva un mayor uso de CPU, incluso si el SCR no mejora. Esto se debe a que las

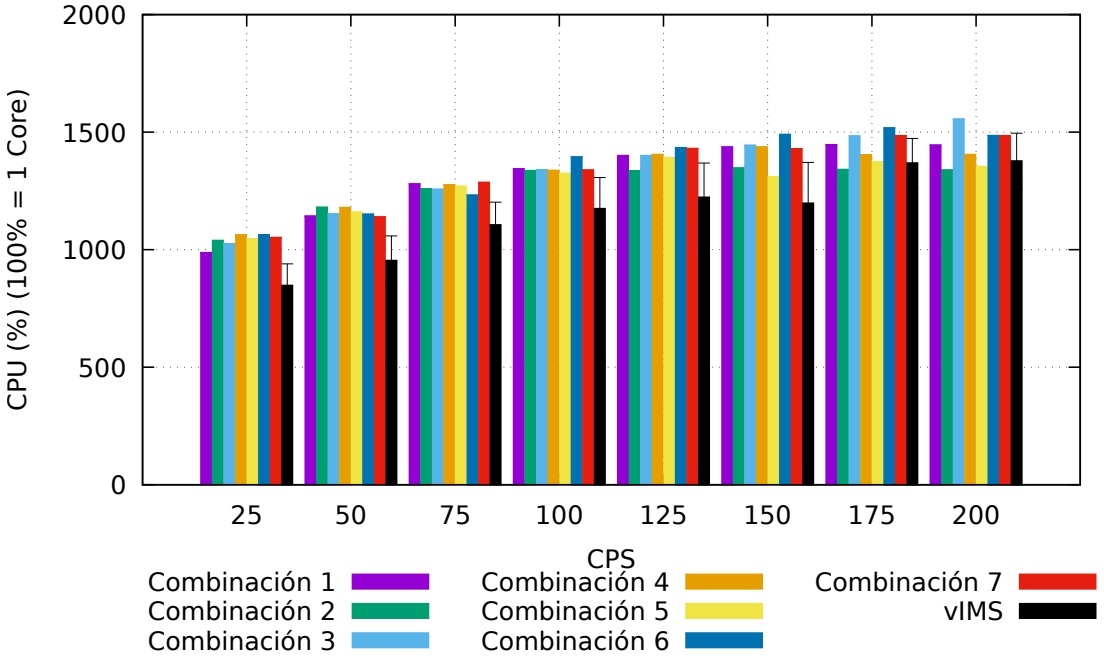

Figura 5.14: Resultados: Consumo de CPU

implementaciones intentan procesar más solicitudes pero no pueden hacerlo con los recursos asignados y la solicitud alcanza un *Time Out*.

5.3.4 Consumo de Memoria

La Figura 5.15 muestra los resultados de la evaluación de memoria. Estos resultados revelan lo siguiente:

- La adición de una instancia de un componente implica un crecimiento en el uso de memoria. Esto se debe a que cada instancia necesita memoria adicional para el rendimiento general.
- El uso de memoria no aumenta significativamente con el número de CPS en la misma combinación, lo que significa que la memoria no es relevante para atender a más usuarios al mismo tiempo.

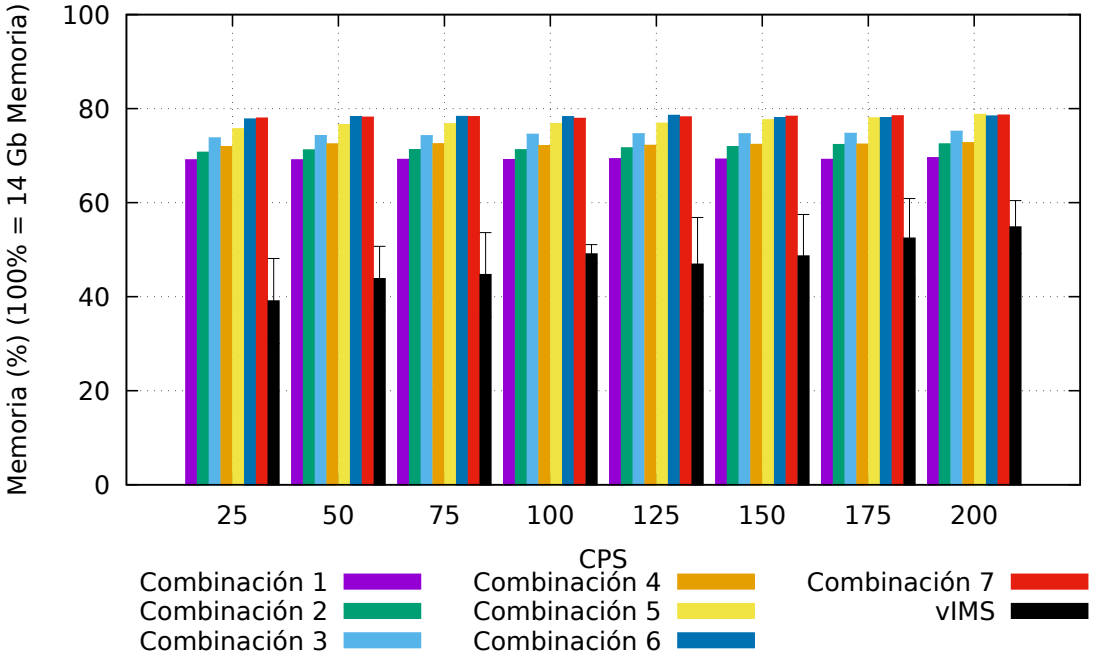

Figura 5.15: Resultados: Consumo de Memoria

- La implementación de vIMS utiliza menos memoria que μvIMS, siendo los resultados de μvIMS entre 75% y 100% más altos. Esto se debe a que los *Elementos MSA* de μvIMS requieren memoria para funcionar, mientras que vIMS no tiene estos elementos.

Los resultados generales del uso de recursos revelan que los elementos MSA necesitan mucha memoria pero poca CPU para funcionar, esto se observa en la diferencia entre vIMS que no tiene *Elementos MSA* y cualquier combinación de μvIMS que tiene los *Elementos MSA*. Finalmente, en el caso de la arquitectura escalada, la adición de nuevas instancias aumenta notablemente el uso de memoria pero no el uso de CPU.

5.3.5 Latencia

Figura 5.16 presenta los resultados de la evaluación de latencia usando una CDF que agrupa la latencia obtenida de 25 CPS a 200 CPS para vIMS y cada combinación de μvIMS. Los resultados son los siguientes:

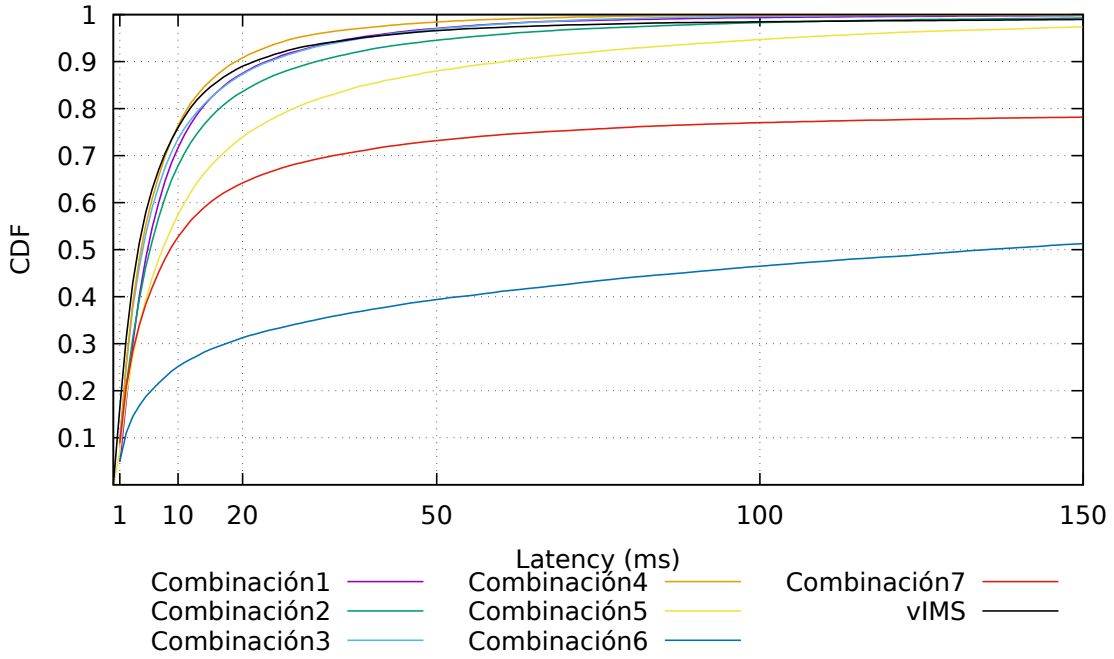

Figura 5.16: Resultados: CDF de Latencia

- Las combinaciones 5, 6 y 7 presentan una notable degradación de la latencia, y superan el límite superior de latencia de 100 ms, por lo que no son óptimas para prestar los servicios de IMS.
- Las combinaciones 1, 2, 3, 4 y vIMS tienen una diferencia de latencia insignificante, y se mantienen por debajo del límite de latencia.

Considerando los resultados de latencia y SCR, se concluye que la Combinación 2 presenta una mejora de SCR con los recursos disponibles sin afectar la latencia como sí lo hace la Combinación 6. Consecuentemente, se concluye que μvIMS alcanza un SCR más alto que vIMS sin sobrepasar el límite de latencia para IMS.

5.3.6 Análisis Cualitativo

Desde un punto de vista cualitativo, se pueden hacer varias afirmaciones:

- μvIMS utiliza microservicios, diferentes de vIMS que posee un diseño monolítico donde un solo componentes realiza varias de las funciones de los microservicios de μvIMS.
- μvIMS utiliza contenedores Docker, lo que permite asignar varias instancias de microservicios a una misma máquina. Esto no se puede lograr en vIMS debido a que cada uno de sus componentes de despliega sobre una máquina.
- μvIMS permite asignar recursos en los microservicios relacionados con el tráfico, logrando una mejora en la escalabilidad. Es importante tener en cuenta que los recursos disponibles son limitados, por ello cuando se los asigna a los microservicios que los necesitan se puede atender más tráfico. Esta asignación no ocurre en vIMS donde se desperdician recursos, debido a que se los asignan a componentes que no son necesarios para manejar el tráfico.
- μvIMS con los recursos disponibles alcanza un mayor SCR que vIMS, lo que demuestra la eficiencia en el uso de los recursos.
- μvIMS utiliza el recurso eficientemente sin sobrepasar el límite de latencia para la señalización IMS (100 ms).

5.3.7 Conclusiones finales

Este Capítulo presentó la evaluación de μvIMS comparándolo con un vIMS sin microservicios. Las métricas utilizadas para su comparación son SCR, consumo de CPU, consumo de memoria y latencia. Como resultado, μvIMS alcanzó un mayor SCR sin pasar el límite de latencia para la señalización IMS de 100 ms, con un ligero incremento del consumo de CPU y memoria.

- Las contribuciones son:
  - Prototipo del Subsistema Multimedia IP basado en microservicios.
  - Escenario de pruebas para el Subsistema Multimedia IP basado en microservicios.

– Análisis cuantitativo y cualitativo del Subsistema Multimedia IP basado en microservicios.

Capítulo 6

Conclusiones y Trabajos Futuros

6.1 Conclusiones

En este trabajo responde la pregunta: **¿Cuál es el rendimiento de un núcleo vIMS diseñado con microservicios considerando las métricas de CPS, latencia, consumo de CPU y consumo de memoria?**.

Para responder dicha pregunta fue necesario diseñar un núcleo vIMS con el enfoque arquitectónico de microservicios, al cuál se le denominó μvIMS. Además, se implementó un prototipo utilizando los *softwares* de fuente abierta Clearwater y Kubernetes. Una vez diseñado e implementado μvIMS, se procedió a realizar el análisis de rendimiento con base a las métricas de CPS, latencia, consumo de CPU y consumo de memoria. Los resultados mostraron lo siguiente:

- CPS: Se alcanzó un SCR 66% más alto que el obtenido con un vIMS de enfoque monolítico.
- Consumo de CPU: La diferencia de uso de CPU con respecto a un vIMS de enfoque monolítico es despreciable debido a que el aumento de CPU es mínimo, a pesar de que se alcanzo un SCR más alto que vIMS. Demostrando que el vIMS utiliza mejor los recursos de CPU disponibles.

- Consumo de memoria: El uso de memoria aumentó entre un 75% y un 100%. Sin embargo, dado que la memoria no es importante para atender más usuarios, la diferencia de consumo de memoria se considera despreciable.
- Latencia: La diferencia de latencia de la arquitectura comparándola con vIMS es despreciable, y fue posible mantener el límite de latencia establecido de 100 ms para señalización IMS.

El rendimiento de un vIMS diseñado con microservicios, comparándolo con un vIMS sin microservicios, es mejor dado que es posible alcanzar más CPS atendidas correctamente con la misma cantidad de CPU y de memoria disponible, y manteniendo una latencia dentro de los límites establecidos para señalización IMS.

6.2 Trabajos futuros

Con base al trabajo realizado, se proponen las siguientes ideas para trabajos futuros:

- Analizar el rendimiento de μvIMS con un número variable de Esclavos Kubernetes.
- Implementar un mecanismo de auto-escalabilidad que permita a μvIMS adaptarse al tráfico del usuario y analizar el rendimiento, la latencia y el consumo de recursos.
- Diseñar otras arquitecturas de telecomunicaciones utilizando el enfoque de microservicios, para identificar si este enfoque arquitectónico permite mejorar el rendimiento de arquitecturas distintas a vIMS.

Bibliografía

[1] K. Dandin, I. Hokelek, and G. K. Kurt, "Dynamic load management for ims networks using network function virtualization," in *Network Operations and Management Symposium (NOMS)*. IEEE, 2016, pp. 1011–1012.

[2] G. Carella, L. Foschini, A. Pernafini, P. Bellavista, A. Corradi, M. Corici, F. Schreiner, and T. Magedanz, "Quality audit and resource brokering for network functions virtualization (nfv) orchestration in hybrid clouds," in *Global Communications Conference*. IEEE, 2015, pp. 1–6.

[3] P. Bellavista, L. Foschini, R. Venanzi, and G. Carella, "Extensible orchestration of elastic ip multimedia subsystem as a service using open baton," in *International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2017, pp. 88–95.

[4] GSMA, "Volte service description and implementation guidelines," *FCM.01*, 2014.

[5] ——, "Ims profile for voice, video and sms over wi-fi," *IR.51-v3.0*, 2016.

[6] A. Cuevas, W. Nicoll, and K. Schroder, "The challenges of ims deployment at telefonica germany," *IEEE Communications Magazine*, pp. 120–127, 2012.

[7] INTEL, "vims for communications service providers," *Intel Builders*, 2016.

[8] ETSI, "Digital cellular telecommunications system (phase 2+) (gsm); universal mobile telecommunications system (umts); lte; ip multimedia subsystem (ims); stage 2," *ETSI TS 123 228 V15.2.0*, 2018.

[9] C. F. Lai and M. Chen, "Playback-rate based streaming services for maximum network capacity in ip multimedia subsystem," *IEEE Systems Journal (SJ)*, pp. 555–563, 2011.

[10] J. Garcia-Reinoso, I. Vidal, P. Bellavista, I. Soto, and P. A. A. Gutirrez, "Transparent reallocation of control functions in ims deployments," *IEEE Communications Magazine*, pp. 106–113, 2016.

[11] A. Sánchez-Esguevillas, B. Carro, G. Camarillo, Y. B. Lin, M. A. García-Martín, and L. Hanzo, "Ims: The new generation of internet-protocol-based multimedia services," *Proceedings of the IEEE*, pp. 1860–1881, 2013.

[12] P. Bellavista, A. Corradi, and L. Foschini, "Enhancing intradomain scalability of ims-based services," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, pp. 2386–2395, 2013.

[13] A. B. Alvi, T. Masood, and U. Mehboob, "Load based automatic scaling in virtual ip based multimedia subsystem," in *Consumer Communications Networking Conference*. IEEE, 2017, pp. 665–670.

[14] IETF, "Middleboxes: Taxonomy and issues," 2002.

[15] C. Wang, O. Spatscheck, V. Gopalakrishnan, Y. Xu, and D. Applegate, "Toward high-performance and scalable network functions virtualization," *IEEE Internet Computing*, pp. 10–20, 2016.

[16] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, pp. 90–97, 2015.

[17] Y. Nakajima, H. Masutani, and H. Takahashi, "High-performance vnic framework for hypervisor - based nfv with userspace vswitch," in *European Workshop on Software Defined Networks*. IEEE, 2015, pp. 43–48.

[18] H. Nemati, A. Singhvi, N. Kara, and M. E. Barachi, "Adaptive sla-based elasticity management algorithms for a virtualized ip multimedia subsystem," in *Globecom Workshops (GC Wkshps)*. IEEE, 2014, pp. 7–11.

[19] A. Boubendir, E. Bertin, and N. Simoni, "A vnf-as-a-service design through micro-services disassembling the ims," in *Conference on Innovations in Clouds, Internet and Networks*. IEEE, 2017, pp. 203–210.

[20] J. Thönes, "Microservices," *IEEE Software*, pp. 116–116, 2015.

[21] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design," *IEEE Software*, pp. 91–98, 2017.

[22] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon*. IEEE, 2016, pp. 1–5.

[23] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, "Savi-iot: A self-managing containerized iot platform," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2017.

[24] R. S. V. Eiras, R. S. Couto, and M. G. Rubinstein, "Performance evaluation of a virtualized http proxy in kvm and docker," in *International Conference on the Network of the Future (NOF)*. IEEE, 2016, pp. 1–5.

[25] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *International Conference on Advances in Computer Engineering and Applications (ICACEA)*. IEEE, 2015, pp. 342–346.

[26] R. Cziva and D. P. Pezaros, "Container network functions: Bringing nfv to the network edge," *IEEE Communications Magazine*, pp. 24–31, 2017.

[27] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *International Conference on Computing, Networking and Communications*. IEEE, 2016, pp. 1–7.

[28] A. Sheoran, X. Bu, L. Cao, P. Sharma, and S. Fahmy, "An empirical case for container-driven fine-grained vnf resource flexing," in *Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2016, pp. 121–127.

[29] G. Carella, M. Corici, P. Crosta, P. Comi, T. M. Bohnert, A. A. Corici, D. Vingarzan, and T. Magedanz, "Cloudified ip multimedia subsystem (ims) for network function virtualization (nfv)-based architectures," in *IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2014, pp. 1–6.

[30] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "Nfv-vital: A framework for characterizing the performance of virtual network functions," in *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 2015, pp. 93–99.

[31] D. Cotroneo, L. D. Simone, and R. Natella, "Nfv-bench: A dependability benchmark for network function virtualization systems," *IEEE Transactions on Network and Service Management*, pp. 934–948, 2017.

[32] P. Potvin, M. Nabaee, F. Labeau, K. K. Nguyen, and M. Cheriet, "Micro service cloud computing pattern for next generation networks," *Computing Research Repository*, 2015.

[33] R. C. Tausworthe, "The work breakdown structure in software project management," *Journal of Systems and Software (JSS)*, pp. 181–186, 1979.

[34] D. T. Nguyen, K. K. Nguyen, S. Khazri, and M. Cheriet, "Real-time optimized nfv architecture for internetworking webrtc and ims," in *International Telecommunications Network Strategy and Planning Symposium (Networks)*. IEEE, 2016, pp. 81–88.

[35] J. M. Been, W. S. Yang, J. H. Kim, and J. O. Lee, "Management of iot traffic using a virtualized ims platform," in *Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2015, pp. 456–459.

[36] M. Tamura, T. Namura, T. Yamasaki, and Y. Moritani, "A study to achieve high reliability and availability on core networks with network virtualization," *Docomo Technical Journal (NTT)*, pp. 42–50, 2013.

[37] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *International Conference on Cloud Engineering (IC2E)*. IEEE, 2014, pp. 610–614.

[38] "Vmware," https://www.vmware.com/.

[39] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in *Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*. IEEE, 2015, pp. 1–8.

[40] "Virtualbox," https://www.virtualbox.org/.

[41] "Open ims core," http://www.openimscore.com/.

[42] "Clearwater," http://www.projectclearwater.org/.

[43] "Docker," https://www.docker.com/.

[44] S. Klock, J. M. E. M. V. D. Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *International Conference on Software Architecture*. IEEE, 2017.

[45] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang, "Orchestration of containerized microservices for iot using docker," in *International Conference on Industrial Technology (ICIT)*. IEEE, 2017, pp. 1532–1536.

[46] S. Newman, *Building microservices*. O'Reilly Media, 2015.

[47] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, May 2018.

[48] F. Montesi and J. Weber, "Circuit breakers, discovery, and API gateways in microservices," *CoRR*, 2016.

[49] F. Gutierrez, *Spring Boot Messaging*. Apress, 2017.

[50] D. Cotroneo, R. Natella, and S. Rosiello, "Nfv-throttle: An overload control framework for network function virtualization," *IEEE Transactions on Network and Service Management (TNSM)*, pp. 949–963, 2017.

[51] E. S. O. Association, "Latency in communications networks," 2015.

[52] M. Taqi Raza, S. Lu, M. Gerla, and X. Li, "Refactoring network functions modules to reduce latencies and improve fault tolerance in nfv," *IEEE Journal on Selected Areas in Communications*, pp. 2275–2287, 2018.

[53] D. Cotroneo, L. D. Simone, A. K. Iannillo, A. Lanzaro, and R. Natella, "Dependability evaluation and benchmarking of network function virtualization infrastructures," in *Proceedings of the Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–9.

[54] L. Manunza, S. Marseglia, and S. Romano, "Kerberos: A real-time fraud detection system for ims-enabled voip networks," *Journal of Network and Computer Applications (JNCA)*, pp. 22 – 34, 2017.

[55] IETF, "Uri design and ownership," 2014.

[56] ETSI, "Network functions virtualisation (nfv);management and orchestration," *ETSI GS NFV-MAN 001 V1.1.1*, 2014.

[57] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[58] P. Samal and P. Mishra, "Analysis of variants in round robin algorithms for load balancing in cloud computing," *International Journal of computer science and Information Technologies*, pp. 416–419, 2013.

[59] S. Chowdhury, "The greedy load sharing algorithm," *Journal of Parallel and Distributed Computing*, pp. 93–99.

[60] "Cassandra," http://cassandra.apache.org/.

[61] "Etcd," https://etcd.io/.

[62] "Kubernetes," https://kubernetes.io/.

[63] T. N. Stack, *The New Stack The State of the Kubernetes Ecosystem*. The New Stack, 2017.

[64] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, 2007.

[65] 3GPP, "Ip multimedia (im) subsystem cx and dx interfaces; signalling flows and message contents," *ETSI TS 129 228 V15.1.0 (2018-09)*, 2019.

[66] V. Kumlin, "Open source sip application servers for ims applications: A survey," 2007.

[67] "Bind9," https://www.isc.org/downloads/bind/.

[68] "Sipp," http://sipp.sourceforge.net/.

Análisis del rendimiento del Subsistema Multimedia IP basado en NFV considerando microservicios

ANEXOS

Trabajo de grado

Juan Sebastian Orduz Vidal

Gabriel David Orozco Urrutia

Director: PhD. Oscar Mauricio Caicedo Rendón
Codirector: Msc. Carlos Hernan Tobar Arteaga

Departamento de Telemática
Facultad de Ingeniería Electrónica y Telecomunicaciones
Universidad del Cauca
Popayán, Cauca, 2017

Capítulo 7

ANEXO A

El Anexo A presenta la estructura del repositorio de GIT HUB en el cual está alojado el código de la implementación y evaluación del μvIMS. Las Tablas 7.1, 7.2 y 7.3 presentan la estructura del repositorio de GitHub, cada directorio tiene un README el cual indica como utilizar los diferentes *scripts*. La versión final del repositorio esta en la rama *kubernetes13* disponible en:

https://github.com/juansorduz/clearwater-docker/tree/kubernetes13

Table 7.1: Estructura Repositorio GitHub Parte 1

Directorio	Archivos	Descripción
astaire	Dockerfile, 4 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Astaire en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
base	Dockerfile, 6 archivos con extensión .conf y 3 archivos adicionales	Este directorio tiene los archivos necesarios para crear la imagen Docker en la cual se basan las demás imágenes en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
bono	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Bono en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
cassandra	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Cassandra en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
chronos	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Chronos en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
ellis	Dockerfile, 5 archivos con extensión .conf y 1 archivo adicional	Este directorio tiene los archivos necesarios para crear la imagen Docker de Ellis en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
homer	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Homer en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
homestead	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Homestead en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
homestead-prov	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Homestead-prov en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
ralf	Dockerfile, 3 archivos con extensión .conf y 1 directorio adicional	Este directorio tiene los archivos necesarios para crear la imagen Docker de Ralf en el despliegue de Clearwater sobre Docker (Sprout sin dividir).
sipptest	Dockerfile	Este directorio tiene los archivos necesarios para crear la imagen del generador de tráfico SIPp configurado para probar el despliegue de Clearwater sobre Docker (Sprout sin dividir).
sprout	Dockerfile, 2 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen Docker de Sprout en el despliegue de Clearwater sobre Docker (Sprout sin dividir).

Table 7.2: Estructura Repositorio GitHub Parte 2

Directorio	Archivos	Descripción
astaire2	Dockerfile, 4 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen de Astaire en el despliegue de μvIMS.
base2	Dockerfile, 6 archivos con extensión .conf y 3 archivos adicionales	Este directorio tiene los archivos necesarios para crear la imagen base para crear las imágenes de los otros pods en el despliegue de μvIMS.
bono2	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen de bono en el despliegue de μvIMS.
chronos2	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen de Chronos en el despliegue de μvIMS.
ellis2	Dockerfile, 5 archivos con extensión .conf y 1 archivo adicional	Este directorio tiene los archivos necesarios para crear la imagen de Ellis en el despliegue de μvIMS.
homer2	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen de Homer en el despliegue de μvIMS.
homestead-prov2	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen de Homestead-prov en el despliegue de μvIMS.
msccassandra	Dockerfile, 3 archivos con extensión .conf y 1 archivo adicional	Este directorio tiene los archivos necesarios para crear la imagen de MSCCassandra en el despliegue de μvIMS.
mschomestead	Dockerfile, 3 archivos con extensión .conf y 1 archivo adicional	Este directorio tiene los archivos necesarios para crear la imagen de MSCHomestead en el despliegue de μvIMS.
mscsprout	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen de MSCSprout en el despliegue de μvIMS.
ralf2	Dockerfile, 3 archivos con extensión .conf y 1 directorio adicional	Este directorio tiene los archivos necesarios para crear la imagen de Ralf en el despliegue de μvIMS.
sipptest2	Dockerfile	Este directorio tiene los archivos necesarios para crear la imagen de sipptest que genera el tráfico en el despliegue de μvIMS.
urcassandra	Dockerfile, 3 archivos con extensión .conf y 1 archivo adicional	Este directorio tiene los archivos necesarios para crear la imagen de URCassandra en el despliegue de μvIMS.
urhomestead	Dockerfile, 3 archivos con extensión .conf y 1 archivo adicional	Este directorio tiene los archivos necesarios para crear la imagen de URHomestead en el despliegue de μvIMS.
ursprout	Dockerfile, 3 archivos con extensión .conf	Este directorio tiene los archivos necesarios para crear la imagen de URSprout en el despliegue de μvIMS.

Table 7.3: Estructura Repositorio GitHub Parte 3

Directorio	Archivos	Descripción
kubernetes	24 archivos de extensión .yaml, 3 archivos adicionales y 2 directorios	Este directorio tiene los archivos necesarios para desplegar Clearwater sobre Docker (Sprout sin dividir).
kuberenetes2	24 archivos de extensión .yaml, 3 archivos adicionales y 2 directorios	Este directorio tiene los archivos necesarios para desplegar una implementación transitoria entre Clearwater sobre Docker y el μvIMS.
kubernetes3	31 archivos de extensión .yaml, 3 archivos adicionales y 2 directorios	Este directorio tiene los archivos necesarios para desplegar el prototipo de μvIMS.
ScriptDat2	32 archivos de extensión .sh y 8 directorios	Este directorio tiene los archivos necesarios para: iniciar el despliegue de μvIMS, iniciar las pruebas, recolectar datos, procesarlos y graficarlos.
ScriptDat3	23 archivos de extensión .sh y 6 directorios	Este directorio tiene los archivos necesarios para: iniciar las pruebas en el despliegue de vIMS(Clearwater sobre VM), recolectar datos, procesarlos y graficarlos.
Scriptinstaller	2 archivos de extensión.sh y un archivo adicional	Este directorio tiene los archivos necesarios para instalar el vIMS (Clearwater sobre VM) de manera automática.
utils	4 archivos .sh	Este directorio contiene scripts que permiten ver las direcciones IP de los contenedores y el estado del clúster.

Capítulo 8

ANEXO B

El Anexo B presenta el artículo enviado a la comunidad científica con el propósito de que sean publicados.

- **Juan S. Orduz, Gabriel D Orozco**, Carlos H. Tobar-Arteaga and Oscar Mauricio Caicedo Rendon. **μvIMS: A Finer-Scalable Architecture Based on Microservices** IEEE Local Computer Networks (LCN 2019), Octubre 14-17
  - Estado: Enviado.
  - Índice H: 45 Scimago
  - Rango: A Core

Capítulo 9

ANEXO C

El Anexo C presenta la configuración del IFC para el AS que presta el servicio de Voicemail.

```
"Voicemail" : "
<InitialFilterCriteria>
        <Priority>1</Priority>
        <TriggerPoint>
                <ConditionTypeCNF></ConditionTypeCNF>
                <SPT>
                        <ConditionNegated>0</ConditionNegated>
                        <Group>0</Group>
                        <Method>INVITE</Method>
                        <Extension></Extension>
                </SPT>
        </TriggerPoint>
        <ApplicationServer>
                <ServerName>sip:192.168.190.41</ServerName>
                <DefaultHandling>0</DefaultHandling>
        </ApplicationServer>
</InitialFilterCriteria>"
```