μvIMS: A Finer-Scalable Architecture Based on Microservices

Juan S. Orduz, Gabriel D. Orozco, Carlos H. Tobar-Arteaga and Oscar Mauricio Caicedo Rendon
Grupo de Ingeniería Telemática, Universidad del Cauca
Email: juanorduz,gabrielorozco20,carlost,omcaicedo@unicauca.edu.co

***Abstract*—The steps toward all over IP have defined to the IP Multimedia Subsystem (IMS) 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 virtualized IMS (vIMS) architectures. However, the current vIMS deployments use monolithic designs that do not allow finer-scalability. In this paper, we present μvIMS, an architecture that uses microservices to provide finer-scalability and more effective resource usage than regular monolithic design. To test our architecture, we evaluate μvIMS prototype regarding CPU usage, RAM usage, Successful Call Rate (SCR), and latency metrics. Our test results reveal that μvIMS achieves a higher SCR, using the available resources effectively with a negligible latency increasing. Thus, we can state that dividing the monolithic vIMS architecture in microservices allows providing finer-scalability.**

I. Introduction

The IP Multimedia Subsystem (IMS) is a telco-architecture intended to provide voice and multimedia IP services regardless of access networks [1]. Some IMS relevant features are openness, interoperability, and support of trending technologies, such as VoLTE, VoWiFi, and WebRTC [2], [3]. Considering these features, the ETSI (European Telecommunications Standards Institute) has decided to continue using IMS in the 5th Generation (5G) networks for provisioning multimedia services [4]. Since in 5G the user traffic will be dynamic, IMS must be adapted to such dynamism [5]. Current IMS deployments have scalability issues because their architectural core components usually are deployed over network appliances or monolithically. These appliances make it difficult to scale the architecture capabilities effectively because Network Functions (NFs) are tied to specific hardware. In turn, a monolithic IMS implicates the assignment of resources to a group of Virtualized Network Functions (VNFs), avoiding to achieve a finer-scalability (*i.e.*, scale a specific function to handle particular network traffic).

Several works use Network Functions Virtualization (NFV) [6], [7] to deal with IMS scalability problem obtaining virtualized IMS (vIMS) and supporting its operation with additional systems [5], [8]. In [5], the authors use a framework that monitors the quality offered by a vIMS, and they scale it regarding network conditions (*e.g.,* service execution times and available bandwidth). In [8], the authors add an autoscaling mechanism that considers several vIMS metrics (*e.g.,* resource usage, number of requests, and latency), allowing to scale depending on user traffic. However, the authors use a monolithic architecture avoiding finer-scalability and failing in using available resources efficiently.

To deal with vIMS monolithic architectures, in the last years, the concept of microservices has been used [9], [10]. The microservices are an architectural model that divides a monolithic application into different components, each one of them with specific functionality [11]. Since these components are smaller than the monolithic application, it is easier to add and delete microservices instances [12], allowing to achieve finer-scalability. In [9], the authors provide IMS registration, authorization, and authentication processes as services using microservices. In [10], the authors present a microservice-based vIMS for cloud computing, in which the main IMS components are inside of a single microservice. Their microservice-based architecture uses an orchestrator and a load balancer for automatic horizontal scaling (*i.e.,* addition and releasing of microservice instances to improve architecture capabilities). In summary, to the best of our knowledge, the microservice-based vIMS solutions proposed in the literature consider a limited number of IMS processes or do not divide the IMS functions efficiently, which unfollow the microservice design, avoiding to achieve finer-scalability.

In this paper, we introduce μvIMS, a microservice-based vIMS architecture that aims to provide finer-scalability in 5G networks. By microservices, our architecture offers finer-scalability, allocating resources just in the necessary IMS functionalities. In μvIMS, we decompose the IMS core in microservices and use elements from the MicroService Architecture (MSA) to ensure security, reliability, and manageability. We implement a μvIMS prototype using Clearwater and Kubernetes. We analyze the performance of μvIMS prototype comparing it with a vIMS. In such comparison, we evaluate the capability of μvIMS to handle a higher number of users with the same amount of resources, measuring the Successful Call Rate (SCR) for an increasing Calls Per Second (CPS) number that simulates unpredictable traffic. The evaluation results reveal that μvIMS reached a higher SCR using a similar amount of resources, and with an acceptable latency increasing, showing the feasibility of providing finer-scalability and allocating resources effectively.

The main contributions presented in this paper are:

- A microservice-based vIMS design for facing IMS scalability issues.
- A microservice-based vIMS prototype that follows the

978-1-7281-2561-9/19/$31.00 ©2019 IEEE

described design.
- The demonstration that our architecture is feasible to provide finer-scalability for IMS and to attend more users with the available resources.

This paper is organized as follows. Section II describes related work. Section III introduces a motivation scenario. Section IV describes the μvIMS architecture. Section V presents the μvIMS prototype and its comparison with vIMS. Finally, Section VI provides conclusions and implications for future works.

II. Related Work

Several works have addressed the IMS scalability issues taking into account different technologies and methods. In the work [8], the authors present a comparison between two auto-scale mechanisms for vIMS. The first one is based on VM (Virtual Machine) metrics (CPU and RAM usage). The second mechanism is based on VNF metrics, such as resource and network usage, TCP connections, number of call drops, and Session Initiation Protocol (SIP) requests. As a comparison result, the authors determine that the use of VNF metrics is the best way to scale vIMS. In [5], the authors propose an NFV-compliant quality audit and resource brokering framework. This framework allows scaling IMS according to user traffic and allocating the resources in different cloud platforms. The authors analyze their proposal deploying a vIMS integrated with their framework in a test-bed over a real hybrid private/public cloud. The above proposals share the same shortcoming; they have IMS monolithic architecture designs, which do not allow finer-scalability and implies a less specific resource allocation that leads to waste resources in scaling functions unrelated to traffic.

In the literature, few works use microservices design pattern in IMS architecture. In [9], the authors propose an approach to achieve an optimal VNF design using microservices. With this approach, the authors introduce *IMS-as-a-Service* that provides IMS registration, authorization, and authentication as a service. The authors tested *IMS-as-a-Service* comparing it with a non-microservice vIMS. As a result, *IMS-as-a-Service* reached a higher number of successful calls and better performance in resource utilization. The authors focus on IMS registration, authorization, and authentication processes, and they do not consider complete IMS core functionalities. The work [10] describes an architecture for the elastic implementation of vIMS based on microservices for cloud computing. The authors allocate the Home Subscriber Server (HSS) information in microservices databases. Besides, they compare their implementation with a vIMS without microservices regarding call establishment delay and conclude that their architecture maintains similar results to regular IMS. However, the authors implemented the Call Session Control Functions (CSCFs) into a single microservice, avoiding finer-scalability. Thus, it is not possible to allocate resources in the specific functions that handle the traffic.

In summary, in order to improve resources usage, vIMS architecture needs finer-scalability and specific resource allocation. Our μvIMS addresses this issue distributing the main IMS functionalities into microservices efficiently. This distribution allows scaling just the necessary functions in order to provide IP Multimedia services to a higher number of users with the available resources.

III. Motivation

In 5G networks, IMS provides voice and multimedia IP services with CSCFs [4]. These CSCFs are: Proxy-CSCF (P-CSCF), Serving-CSCF (S-CSCF), Interrogating-CSCF (I-CSCF). P-CSCF is the first contact point between users and IMS, which validates requests and routes them to their destination. S-CSCF manages the multimedia session, registers users, and forwards requests to the correct IMS element. I-CSCF verifies users profile, assigns the user to an S-CSCF, and routes requests to other IMS networks.

Now, let us consider the next scenario: a telco-operator enterprise has an IMS that provided voice service in a region for several years. After analyzing other telco-operators, the enterprise concludes that they must provide 5G services (*e.g.*, telemedicine, and Internet of Things) to remain competitive. The implementation of these services is successful, but after a while, the telco operator customers complain about 5G services continually fail. The enterprise analyzes the problem and discovers that its IMS unsuccessfully tries to handle traffic with unpredictable peaks introduced by these 5G services. Thus, the telco-operator needs to improve its IMS capabilities.

In the previous scenario, the telco-operator has an IMS that cannot handle unpredictable traffic because it does not scale efficiently. A solution for this scenario may be to virtualize its IMS and use an additional system for automatic scalability [5], [8]. However, this solution uses a monolithic IMS architecture that implies a less specific resource allocation when it is scaled. The telco-operator can use microservices to deal with this monolithic architecture, but the current approaches [9], [10] do not implement the IMS functions needed for multimedia session control or do not provide specific resource allocation. In μvIMS, we divide the CSCFs into seven independent microservices that provide the full multimedia session control. This division allows scaling the microservices with the functions that are handling the incoming traffic, using better the available resources to handle more users.

IV. A μvIMS architecture

μvIMS is an architecture based on microservices and designed to improve IMS scalability in 5G. Microservices offer the ability to allocate resources in the functions needed to handle traffic, leading to finer-scalability and reducing unnecessary resource usage. Figure 1 depicts the μvIMS overall view that includes two kinds of components: *CSCF Microservice Cluster* and *Enhancing Elements*. *CSCF Microservice Cluster* performs the CSCF with microservices. *Enhancing Elements* maintain the proper operation of these microservices performing microservice registry, microservice management, and interoperability with outside elements (*e.g.*, 5G Networks, User Entities, Application Server, and HSS).

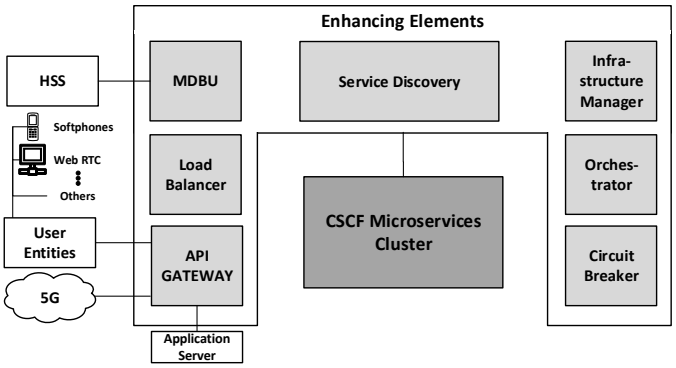

Fig. 1: Overall μvIMS Architecture

In the next subsections, we describe the functional division of the CSCFs into microservices. Afterward, we present the *Enhancing Elements*. We use the *Enhancing Elements* that we found in the literature [13], [14], [15], and we propose new ones to provide interoperability and security.

A. CSCF Microservice Cluster

This Cluster uses microservices to build up the CSCFs responsible for multimedia session control of users and Application Servers (ASs). These microservices offer to μvIMS the following characteristics. First, the CSCF functional division maintains an adequate size. Note that large-size microservices group many functions, returning to monolithic design. Small-size microservices decouple functions heavy dependent of each other, resulting in additional management complexity [9], [16]. Second, the CSCF functional division maintains data independence because each microservice has its own database, a centralized database implicates losing independence [10].

Figure 2 describes the decomposition of CSCF into microservices. In this division, we provide in each microservice a complete and independent functionality based on CSCF specified by the 3rd Generation Partnership Project (3GPP) [4]. We divided the P-CSCF into two microservices: *Forwarding SIP Messages* and *Ensure Access Policies*. *Forwarding SIP Messages* microservice is responsible for routing SIP messages generated by outside elements (*e.g.*, traditional phone, softphone, and WebRTC), ensuring that the SIP messages have the correct format, as well as detecting and routing emergency sessions requests. The *Ensure Access Policies* microservice manages operator policies in architecture access. This microservice stores operator policies in its database and provides them to *Forwarding SIP Messages* microservice to ensure routing SIP messages according to operator policies.

We divide the S-CSCF into four microservices: *User Registry*, *Ensure Multimedia Session Policies*, *Multimedia Session Control* and *Manage IMS Multimedia Priority Service (MPS)*. The *User Registry* microservice is responsible for authorizing the user registry in the architecture, grouping I/S-CSCF registry functions. Furthermore, it translates the E.164 address (*i.e.*, globally unique number for each device in the Public Switched Telephone Network) required for some type of user registries. This microservice needs a database to store user information. The *Ensure Multimedia Session Policies* microservice manages operator policies in multimedia session control. This microservice attests subscriber identity if it is configured through operator policies. *Ensure Multimedia Session Policies* microservice stores the operator policies into its database and provides these policies to *Multimedia Session Control* microservice.

The *Multimedia Session Control* microservice provides the main functionalities of the S-CSCF related to the multimedia session control. These functionalities include start, maintenance, and end multimedia sessions. We group the multimedia session control of user with user and user with AS into a single microservice because they are heavily dependent on each other. For example, a call between two users could include AS interaction for redirecting, voice mail recording or blocking unwanted users. The usage of two microservices for each sort of multimedia session implies additional network traffic and logic to communicate them and provide session control. With the purpose that each microservice provides complete functionality, the *Multimedia Session Control* microservice handles any type of session. Furthermore, this microservice has its own database to handle user information.

Manage IMS MPS microservice handles IMS MPS by providing preferential access treatment to priority users when congestion on the net is blocking the session establishment. In this sense, this microservice validates if a user is authorized for priority service by AS, includes priority level in the user request and forwards it. The *Manage CDRs* microservice manages Call Detail Records (CDRs), grouping P/I/S-CSCF logic associated with CDRs. This microservice recollects CDR from other microservices and generates standardized CDRs that stores in its database, and sends to an external billing entity.

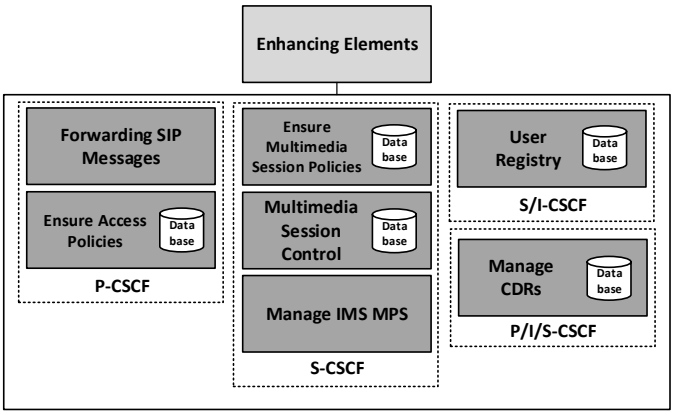

Fig. 2: CSCF Microservice Cluster

B. Enhancing Elements

Figure 3 presents the *Enhancing Elements* that provide security, reliability and microservice management namely *Service Discovery*, *Orchestrator*, *Infrastructure Manager*, *Circuit Breaker*, *API Gateway*, *Microservice DataBase Updater*

(MDBU), and Load Balancer. The *Service Discovery* is a register element that supports microservices communication. It includes the *Register&Discovery* and *Authenticator* components. The *Register&Discovery* component stores and provides microservice addresses. These addresses are URIs that point an IP address necessary for the forwarding of traffic to the microservices. To guarantee security in microservices addresses provision, we propose a new *Service Discovery* component called *Authenticator*. This new component verifies that the element that requests a microservice address is trusty. *Authenticator* could be implemented with several options. For example, a basic key-password verification system.

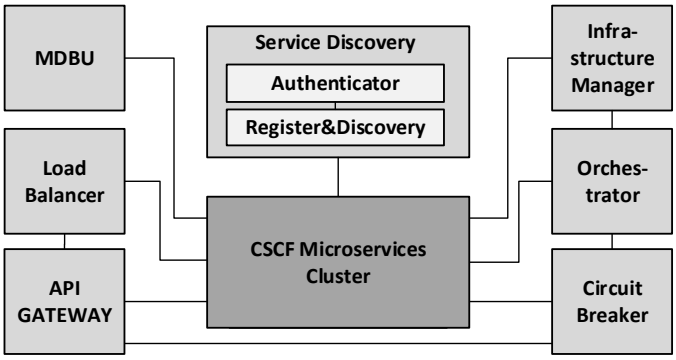

Fig. 3: Enhancing Elements

The management elements are *Orchestrator*, *Infrastructure Manager*, and *Circuit Breaker*. *Orchestrator* and *Infrastructure Manager* are adapted the functionalities of three entities of NFV-MANO [17]: Virtualized Infrastructure Manager (VIM), VNF Manager (VNFM) and NFV Orchestrator (NFVO). *Orchestrator* adapts the functionalities of VNFM and NFVO. In this sense *Orchestrator* manages the microservices life cycle independently of infrastructure by replicating, migrating, initiating, pausing and removing microservices instances. In this sense, the network administrator can easily managing microservices using Orchestrator without manual infrastructure interaction.

The *Infrastructure Manager* adapts the VIM functionalities. Thus, it interacts directly with the infrastructure resources (*i.e.,* computing, storage, and networking) of each machine where we deploy the *CSCF Microservice Cluster*. *Infrastructure Manager* receives management orders from *Orchestrator* and performs them in the *CSCF Microservice Cluster*. Thus, the *Infrastructure Manager* decides where to allocate a microservices according to the available resources. Additionally, when a machine fails, *Infrastructure Manager* deploys microservices that ran on it within another available machine.

Circuit Breaker provides reliability to the *CSCF Microservice Cluster* by handling the microservices failures. When a microservice failure occurs the *Circuit Breaker* records it. After the number of failures recorded by *Circuit Breaker* exceeds a threshold configured by the network administrator, *Circuit Breaker* determines that it is necessary to reset the failure microservice instance. Then, the *Circuit Breaker* indicates to the *Orchestrator* which microservices instances reset. If the failure persists, the *Circuit Breaker* concludes that the failure could not be managed just by resetting the microservice instance. Therefore, this component indicates to the *API Gateway* to stop outside traffic and informs the network administrator about the microservice failure.

API Gateway, *Load Balacer* and *MDBU* are interoperability elements that allow secure microservice communication with outside elements such as User Entities, 5G, and Others Networks. The *API Gateway* is an element that prevents unauthorized access to the *CSCF Microservice Cluster*. When the *API Gateway* ensures reliable access with an authentication mechanism, it distributes the traffic between *CSCF Microservice Cluster* and outside architecture [18]. *Load Balancer* is an element that improves architecture capabilities by distributing traffic between microservice instances. *Load Balancer* uses an algorithm to distribute traffic, such as randomized, round-robin or greedy. The distributed traffic comes from external elements as User Entities or internal elements as other microservices.

MDBU is a new *Enhancing Element* that shares information between microservices databases and actualizes them with the HSS (*i.e.*, the principal repository to store an retrieve information of users). HSS cannot be divided into microservices databases because 5G and other networks use it. With this element, our architecture does not affect other networks by modifying the HSS. The *MDBU* performs HSS synchronization by two steps: (i) it monitors the HSS looking for changes, (ii) it actualizes the microservices databases when the HSS is updated.

V. Evaluation

To evaluate our architecture, first, we implemented a μvIMS prototype. Second, we built a test environment to compare our architecture with vIMS without microservices. Third, we performed tests regarding SCR, resource usage, and latency.

A. Prototype

We use Clearwater to implement our *CSCF Microservice Cluster* modifying its architecture according to our design (modifications are explained in the next paragraph). Clearwater [19] is a widely used open-source IMS core designed for cloud environments that follows IMS core standardized interfaces. Clearwater has two kinds of deployment, one over VMs and other over Docker containers. Clearwater-over-VM deployment follows a monolithic approach, where a single component allocates several Clearwater-over-Docker components. Table I shows the components of both deployments and their functionalities. In Clearwater-over-Docker deployment, each component is implemented over a single container. Also, these components are similar to microservices of *CSCF Microservice Cluster*. Thus, we adapt Clearwater-over-Docker components by using or dividing them to obtain μvIMS prototype.

We implement *Forwarding SIP Messages* microservice using Bono. In turn, we use Ralf to implement *Manage CDRs* microservice. Clearwater-over-Docker uses Sprout, Cassandra

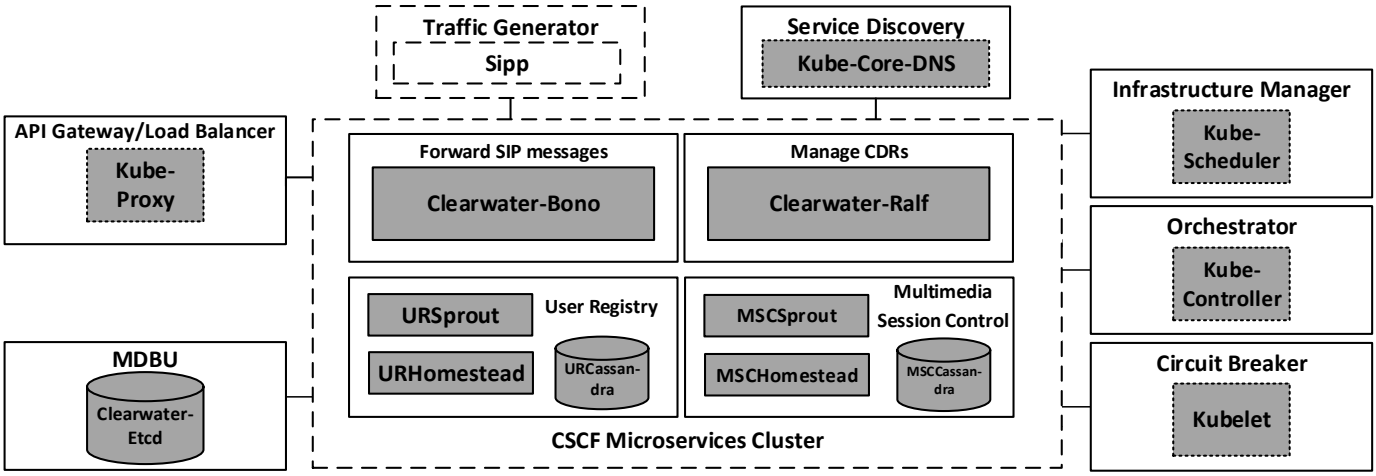

Fig. 4: μvIMS prototype

TABLE I: Clearwater division

Clearwater-over-VM	Clearwater-over-Docker	Functionality
Ellis	Ellis	Web-based provision portal for user sign-up, password management, and control of service settings.
Bono	Bono	SIP edge proxy that performs P-CSCF functionalities.
Sprout	Sprout	SIP register and authorization routing proxy which perform I-CSCF and S-CSCF functionalities.
Homer	Homer	XML Document Management Server (XDMS) that stores user services settings.
Dime	Ralf	HTTP API to recollect billable events from Bono and Sprout.
	Homestead	Interface to provide communication between Sprout and Cassandra.
	Homestead-prov	Interface to provide communication between Ellis and Cassandra.
Vellum	Astaire	Memcached service to store user registry/session states.
	Cassandra	Database to store and provide user information to Sprout through Homestead.
	Chronos	Temporizer service to allow Sprout large-duration events management.
	Etcd	Distributed Key-value service to share information between two or more cluster elements.

database, and Homestead for multimedia session control and registry users. To maintain our independence between *User Registry* and *Multimedia Session Control* microservices, we divide the Sprout, Homestead, and Cassandra components. In particular, we implement *User Registry* microservice with URSprout, URHomestead, and URCassandra. Also, we use MSCSprout, MSCHomestead, and MSCCassandra to implement *Multimedia Session Control* microservice. It is essential to highlight that, URCassandra and MSCCassandra synchronize their information because MSCSprout manages multimedia session of users registered by URSprout in URCassandra. To address this synchronization, we implement *MDBU* with Etcd to share users information, forming a cluster of Cassandras.

We use Kubernetes [20] to implement the *Enhancing Elements* intended to provide security, reliability, and microservice management (Figure 4). Kubernetes is a widely used open-source containers orchestrator that manages our *CSCF Microservice Cluster*. Moreover, Kubernetes works over one or more machines that shape a Kubernetes cluster. The Kubernetes cluster consists of one master and several workers, where the master is in charge of resource management (CPU and RAM) provided by workers to deploy microservices. For the *Service Discovery* implementation, we use a DNS server named Core-DNS that performs *Register&Discovery* functionality. We implement *Orchestrator* functionalities with Kube-controller component, which ensures microservices are running over a worker. Afterward, we use Kube-scheduler component to implement *Infrastructure Manager* functionalities, this element tracks available workers resources and allocates microservices in them. We implemented *Circuit Breaker* deploying Kubelet over each worker. It ensures microservices health by checking their state. Finally, we use Kube-proxy as *API Gateway* and *Load Balancer*, Kube-proxy is in each worker providing external access and distributing the microservice workload.

B. Test Environment

In this section, we present the test environment used to evaluate the μvIMS prototype. We constructed it over the University of Cauca datacenter called Telco 2.0. We Assigned the same number of machines deployed over VMWare 6.0 with the same amount of resources to two deployments: the μvIMS prototype, and the vIMS. Table II shows both deployments resources assignation and their roles in each deployment. Figure 5 presents the μvIMS deployment that consisted of seven Virtual Machines (VMs) that shape a Kubernetes cluster. This cluster is composed of one Kubernetes master and six Kubernetes workers, and an eighth VM to generate traffic. Over the six Kubernetes workers, we deployed the *CSCF Microservice Cluster*. The Kubernetes master decides in which

TABLE II: Test environment resources

Machine	Resources	vIMS	μvIMS
Machine1	4 Intel(R) Xeon(R) CPU E5-2670 2.30 GHz and 16 GB RAM	Ellis	Kubernetes Master
Machine 2		Bono	Kubernetes Workers (CSCF Microservice Cluster)
Machine 3		Sprout	
Machine 4		Homer	
Machine 5		Dime	
Machine 6		Vellum	
Machine 7	4 Intel(R) Xeon(R) CPU E5-2670 2.30 GHz and 6 GB RAM	Bind9	
Machine 8	4 Intel(R) Xeon(R) CPU E5-2670 2.30 GHz and 16 GB RAM	SIPp	

Kubernetes worker allocates each microservice.

Figure 6 presents the vIMS deployment that includes six VMs. These VMs deploy each one of Clearwater-over-VMs components: Ellis, Bono, Sprout, Dime, Homer, and Vellum. These components need a DNS to communicate between them. Thus, we implemented it in a seventh VM using Bind9 [21]. Finally, vIMS had an eight VM to generate traffic. In both deployments, the eight VM used SIPp [22] traffic generator as a testing tool.

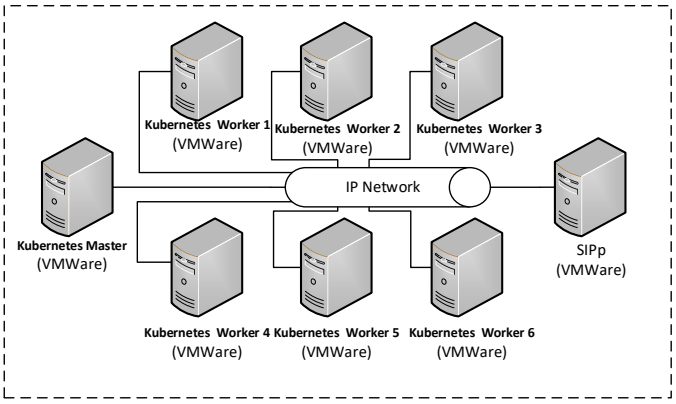

Fig. 5: Modified Clearwater over Kubernetes (μvIMS)

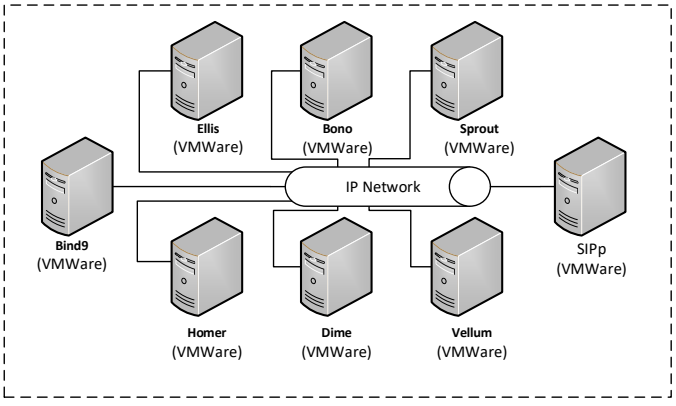

Fig. 6: Clearwater over VM (vIMS)

To compare both deployments, we defined a test scenario using SIPp with three phases: registry of two users, call establishment and call ended between them. This scenario is repeated for several pairs of users at the same time, depending on the Calls Per Second (CPS) defined to test the deployments. We increased the CPS from 25 to 200 in steps of 25 CPS. Each test lasted 60 seconds, and we repeated it 32 times to average the results. We emulated this process in both deployments, and the metrics that we used to compare them are SCR, resource usage (*i.e.*, CPU and Memory of the seven VMs that shaped the deployments), and latency.

C. Results and Analysis

We tested the architecture finer-scalability comparing μvIMS with vIMS. We started from a μvIMS deployment with a single instance for each component. Then, we scaled the most resource-consuming components (*i.e.*, Bono, URSprout, MSCSprout) to determine the component in which allocate resources to achieve better performance. Then, to determine the next combinations, we added new instances to the combination that achieved the best performance. Others combinations were not included in this analysis because performance does not improve. Table III shows the number of component instances deployed in each combination. We ran the test scenario for each combination and vIMS deployment, and we measured SCR. After, we measured the μvIMS deployment resource usage (CPU and RAM) of each combination and comparing it with vIMS. Also we performed a Latency evaluation of μvIMS deployment combinations and vIMS, considering the latency threshold for IMS signaling (*i.e.*, 100 ms) [23]. We aimed to find a combination that achieved a better SCR without an excessive increase in resource consumption and latency, corroborating the feasibility of μvIMS finner-scalability.

TABLE III: μvIMS prototype combinations

Deployment	Bono (Instances)	URSprout (Instances)	MSCSprout (Instances)
Combination 1	1	1	1
Combination 2	2	1	1
Combination 3	1	2	1
Combination 4	1	1	2
Combination 5	3	1	1
Combination 6	2	2	1
Combination 7	2	1	2

Figure 7 shows the SCR evaluation results of vIMS and the seven μvIMS combinations. These results reveal different facts. First, without scaling components (Combination 1), μvIMS presents a slightly better SCR than vIMS. Second, with another Bono instance (Combination 2), the SCR results are better than Combination 1. As well as, with Combination 2, μvIMS reaches a higher number of SCR than vIMS. Third, adding another URSprout instance (Combination 3) or another MSCSprout instance (Combination 4), the results are worse than vIMS. According to those results, Combination 2 has better SCR. Thus, in the next combinations, we add other component instances to Combination 2. Fourth, with three Bono instances (Combination 5), the results are worse than Combination 2, but they are better than vIMS. Fifth, With

two Bono and Two URSprout instances (Combination 6), the results are better than vIMS, and similar to Combination 2. Finally, with two Bono and Two MSCSprout instances (Combination 7), the results get worse than Combination 2 and 6, and in higher CPS, they are similar to vIMS. We conclude that Combination 2 and 6 have higher SCR than vIMS and the other μvIMS combinations. This corroborates the feasibility of μvIMS to attend more calls successfully with the available resources (*i.e.,* the same number of machines with the same resources), using them effectively.

It is important to highlight that the architecture combination shows a stable behavior before reaching a maximum SCR. After a combination reaches its maximum, the curves become saturated, since the capacities of μvIMS for this combination are exceeded. Besides, the SCR results show that each combination performance is related to the advantages of a new instance and the disadvantages of managing this new instance. For example, the results of Combination 3, and 4 show insights that adding MSCSprout and URSprout to Combination 1 does not improve the performance because control traffic is added and the architecture does not need more instances of those components but another Bono instance. For that reason, those combinations present lower SCR than vIMS.

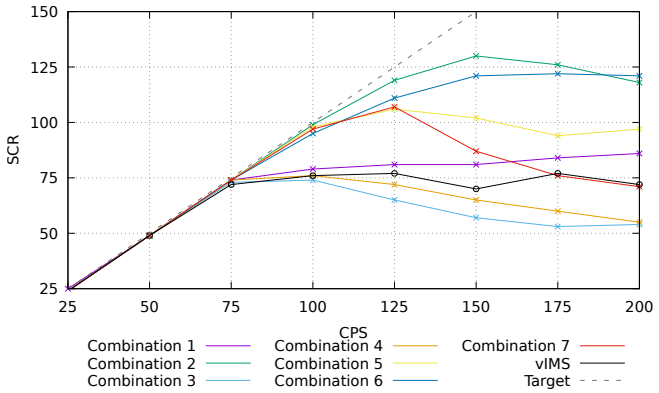

Fig. 7: CPS vs SCR

Figure 8 presents the CPU usage evaluation results of the seven μvIMS combinations and vIMS. These results reveal that the difference between scaled combinations of our μvIMS deployment is negligible. The CPU usage of vIMS for a small number of CPS is lower than μvIMS. However, the evaluation shows that for a high number of CPS, the CPU usage of vIMS increase to a similar level than μvIMS. Furthermore, the SCR and CPU results show that a higher number of CPS implies more CPU usage, even if the SCR does not improve. This is because the deployments try to process more requests but are not able to do it with the available resources. Consequently, the request reaches a time out. Regarding the results of SCR and CPU usage, Combination 2 and 6 achieve a higher SCR without excessive CPU usage.

Figure 9 shows the RAM results. These results reveal that the addition of a new component instance implies a growth in the use of RAM. For example, combinations 2, 3, and

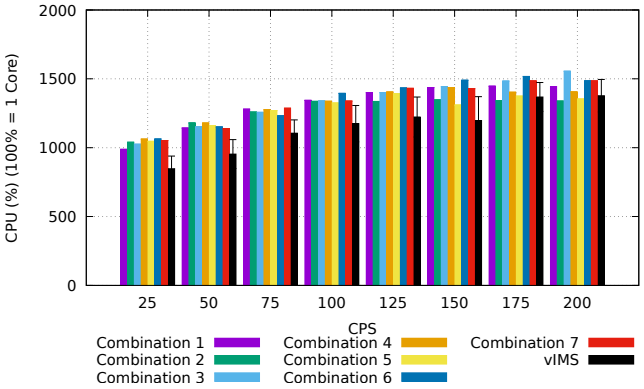

Fig. 8: CPS vs CPU

4 that have one additional instance, use more RAM than Combination 1. This is because each replica needs additional RAM for the overall performing. Another important aspect is that RAM usage does not increase significantly with the number of CPS in the same combination, meaning that RAM is not important to attend more users at the same time. In addition, the regular vIMS implementation uses lower RAM than μvIMS; this is because of the *Enhancing Elements*. The overall results of resource usage reveal that the *Enhancing Elements* need a lot of RAM but few CPU to operate. Finally, in the case of scaled architecture, the addition of new instances increases notably the RAM usage but not the CPU usage.

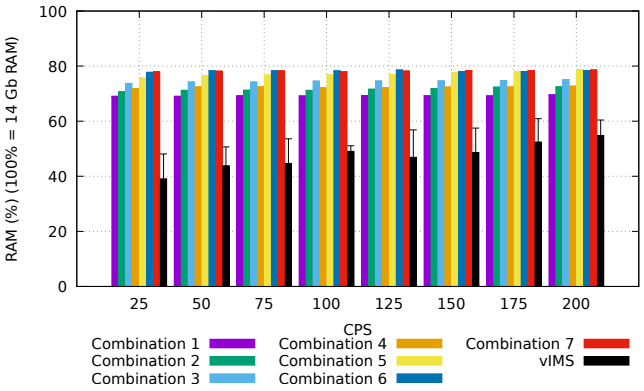

Fig. 9: CPS vs RAM

Finally, Figure 10 presents the results of latency evaluation using a Cumulative Distribution Function (CDF) that groups the latency obtained from 25 CPS to 200 CPS for vIMS and each μvIMS combination. The Combination 5, 6, and 7 present a sightly latency degradation, and they overpass the 100 ms latency threshold. The Combinations 1, 2, 3, 4, and vIMS have a negligible latency difference, and they maintain under the latency threshold. Thus, regarding the overall results, we conclude that Combination 2 present a sightly SCR improvement with the available resources without affecting latency as Combination 6. Consequently, we conclude that μvIMS reaches a higher SCR than vIMS without overpass the latency

threshold for IMS.

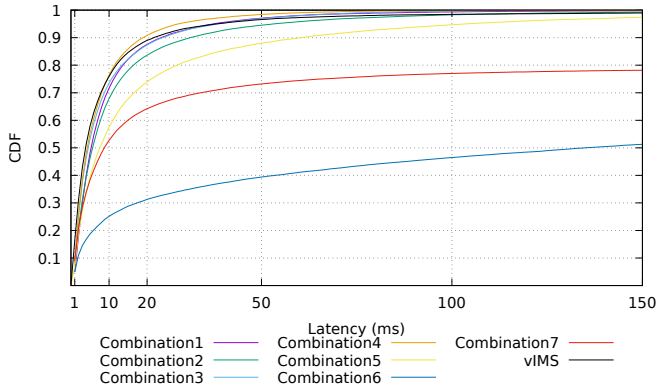

Fig. 10: Cumulative Distribution Function of Latency

Finally, making a qualitative comparison, we can state the following asseverations. First, our architecture design uses microservices, different than vIMS, whose architecture is monolithic. Second, μvIMS allows allocating resources in the microservices related to the traffic, achieving finer-scalability. Third, with the available resources, μvIMS achieve a higher SCR than vIMS, demonstrating resource usage efficiency. Finally, our architecture uses resource efficiently without overpassing the latency threshold for IMS signaling (*i.e.*, 100 ms).

VI. Conclusions and Future Work

In this paper, we present μvIMS, an architecture aimed to provide IMS finer-scalability in 5G networks that suffer dynamic traffic variations. In order to handle this dynamism, μvIMS decomposes the CSCF into microservices and uses *Enhancing Elements*, improving the available resource usage and achieving finer-scalability. The main contributions are: a microservice-based IMS design called μvIMS, a μvIMS prototype for the provision of CSCF, and performance comparison between μvIMS and vIMS using the same number of VMs with the same amount of resources. This comparison reveals that our μvIMS reached a higher SCR than vIMS, with similar resource usage, and without excessive latency increasing. We corroborated that using microservices μvIMS provides finer-scalability because the resources are allocated in the specific IMS function necessary to handle the traffic.

As future works, we are interested in analyzing the performing of μvIMS with a variable number of Kubernetes workers, and the impact of additional traffic introduced by Enhancing Elements. Furthermore, we plan to add an auto-scalability mechanism that allows μvIMS to adapt to user traffic and analyze the performing, latency, and resource usage.

Acknowledgment

The authors would like to thank the working team of the University of Cauca Telco 2.0 for their essential support for this investigation. The implementations presented in this paper were all deployed over the Telco 2.0 datacenter.

References

[1] Kang Wang, Guiqing Gao, Yuanli Qin, and Xiangyong He. Building of communication system for nuclear accident emergency disposal based on ip multimedia subsystem. In *AIP Conference Proceedings*. AIP, 2018.
[2] INTEL. vims for communications service providers. *Intel Builders*, 2016.
[3] V. G. Ozianyi and N. Ventura. Design and implementation of scalable ims charging systems. In *Conference on Local Computer Networks*. IEEE, 2009.
[4] 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.
[5] 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*, pages 1–6. IEEE, 2015.
[6] J. Lai and Q. Fu. Man-in-the-middle anycast (mima): Cdn user-server assignment becomes flexible. In *Conference on Local Computer Networks*. IEEE, 2016.
[7] C. H. T. Arteaga, F. Rissoi, and O. M. C. Rendon. An adaptive scaling mechanism for managing performance variations in network functions virtualization: A case study in an nfv-based epc. In *International Conference on Network and Service Management*, pages 1–7, 2017.
[8] A. B. Alvi, T. Masood, and U. Mehboob. Load based automatic scaling in virtual ip based multimedia subsystem. In *Consumer Communications Networking Conference*, pages 665–670. IEEE, 2017.
[9] 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*, pages 203–210. IEEE, 2017.
[10] Pascal Potvin, Mahdy Nabaee, Fabrice Labeau, Kim Khoa Nguyen, and Mohamed Cheriet. Micro service cloud computing pattern for next generation networks. *Computing Research Repository*, 2015.
[11] D. Gallipeau and S. Kudrle. Microservices: Building blocks to new workflows and virtualization. *SMPTE Motion Imaging Journal*, 127(4):21–31, May 2018.
[12] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu. Conmon: An automated container based network performance monitoring system. In *Symposium on Integrated Network and Service Management*, pages 54–62. IFIP/IEEE, 2017.
[13] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert. Microservices. *IEEE Software*, 35(3):96–100, May 2018.
[14] H. Knoche and W. Hasselbring. Using microservices for legacy software modernization. *IEEE Software*, 35(3):44–49, 2018.
[15] Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and API gateways in microservices. *CoRR*, 2016.
[16] 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.
[17] ETSI. Network functions virtualisation (nfv);management and orchestration. *ETSI GS NFV-MAN 001 V1.1.1*, 2014.
[18] D. Taibi and V. Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62, 2018.
[19] Clearwater. http://www.projectclearwater.org/.
[20] Kubernetes. https://kubernetes.io/.
[21] Bind9. https://www.isc.org/downloads/bind/.
[22] Sipp. http://sipp.sourceforge.net/.
[23] 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*, pages 2275–2287, 2018.