An architecture for on-demand service deployment over a telco CDN

Pantelis A. Frangoudis*, Louiza Yala*, Adlen Ksentini*, and Tarik Taleb§
*IRISA/University of Rennes 1, France
§Aalto University, Finland
Email: *name.surname@irisa.fr, §talebtarik@ieee.org

Abstract—**Internet Service Providers are becoming more involved in the audiovisual content delivery chain. One manifestation of this trend is the emergence of telco CDNs, i.e., content delivery networks operated by telecom service providers. In this work, we make the case for opening the telco CDN infrastructure to content providers by means of network function virtualization (NFV) and cloud technologies. We design and implement a CDN-as-a-Service architecture, where content providers can lease CDN resources on demand at regions where the ISP has presence. Using open northbound RESTful APIs, content providers can express performance requirements and demand specifications, which can be translated to an appropriate service placement on the underlying cloud substrate. To gain insight which can be applied to the design of such service placement mechanisms, we evaluate the capabilities of key enabling virtualization technologies by extensive testbed experiments.**

I. Introduction

Internet traffic is dominated by data distributed over Content Delivery Network (CDN) infrastructures, and the current Internet ecosystem is, to a significant extent, shaped by the interactions of different key actors with often conflicting objectives, including Over-the-Top (OTT) content providers, IP transit providers, CDN infrastructure providers and ISPs [1].

Content distributed over the top creates a bidirectional dependence between CDN and network providers: CDN customers, and, in turn, end users, depend on the underlying network infrastructure and user experience is affected by its conditions. Network operators, on the other hand, are vulnerable to the traffic dynamics caused by time-varying shifts in content demand. At the same time, network providers wish to take advantage of their regional presence and proximity to end users to enter the content delivery market.

This has led to the emergence of telco CDNs, i.e., content delivery infrastructures owned and operated by network providers. In this case, the network operator installs data centers at its points of presence (or other points strategically located in its network) and offers a CDN service advertising high-performance delivery due to user proximity. From the perspective of a content provider, traditional and telco CDNs are not competitive, but, rather, complementary services: A telco has the advantage of proximity to end users, but typically this is limited to specific geographic locations where it has presence. In contrast, traditional CDN players compete offering a global service.

In this work, with cloud orchestration frameworks and network function virtualization (NFV) as our enabling technologies, we design a scheme which offers the flexibility to a telecom operator to lease its CDN infrastructure in a dynamic manner, offering a virtual CDN (vCDN) service which can be deployed on demand over the operator's private cloud infrastructure. This can be considered an evolution towards opening a telco CDN to potentially (but not exclusively) smaller-scale content providers. Our basic design goals are (i) to offer a well-specified, extensible northbound interface to customers, which will allow them to express service demand specifications and performance constraints, allowing them sufficient control at the service level, but abstracting internal network and infrastructure details, and (ii) to be able to combine customer-provided demand dimensioning information (e.g., target number of users per region) with infrastructure/network awareness for optimal resource allocation. We make the following contributions:

- We design and implement an extensible cloud-based vCDN service architecture, tailored to telecom providers. Our design features RESTful APIs which enable content providers to lease virtual CDN resources on demand.
- We carry out extensive experiments to explore the capabilities and limitations of the virtualization technologies we apply. Our results can be used to drive resource allocation/service placement algorithms that aim to optimally satisfy user demand, and to provide insight to the system operator with respect to service pricing.

This paper is structured as follows. In Section II we review recent advances in telco CDNs and NFV. Section III presents our architecture for CDN-as-a-Service (CDNaaS) provision. In Section IV, we present an experimental evaluation of the capacity of candidate virtualization technologies which can support our vision, before we conclude the paper in Section V.

II. Related work

A. Telco CDNs

Given the complementary competitive advantages of telco and traditional CDNs, Lee et al. [2] analyze the strategic interactions between the two types of players. Importantly, they study the conditions that can lead to alliances among

This work was supported in part by the French FUI-18 DVD2C project.

telco CDNs and provide evidence that if a telco CDN properly manages to offer better service quality exploiting its competitive advantages (e.g., joint traffic engineering and content distribution), market benefits are possible.

Kamiyama et al. [3] study some critical decisions that a telco CDN operator needs to take, namely, where to place data centers, and where and how to cache content. For an overview of the challenges, design goals and principles for a telco CDN, the reader is referred to the work of Spagna et al. [4].

Our work focuses on different aspects, i.e., how the CDN infrastructure can be flexibly managed and leased to content providers in a dynamic fashion, attempting to match user demand taking into account insight from performance measurements for service dimensioning.

B. *Network function virtualization*

Network Function Virtualization (NFV) is becoming a key technology for future large-scale service delivery [5]. NFV involves carrying out in software networking tasks that were traditionally performed by costly, special-purpose hardware. NFV is being applied to a diverse set of functions [6], [7]. In our case, the basic components of our architecture (service orchestration, virtual infrastructure management, etc.) and the components of the CDN service (caches, load balancers, name servers, etc.) are implemented as VNFs.

As pointed out by Wood et al. [8], the flexibility offered by NFV comes with a performance cost due to virtualization. They therefore identify the need for a carefully designed NFV platform, coupled with a sophisticated SDN control plane. Our work addresses such challenges, by experimentally quantifying the performance capabilities of core enabling virtualization technologies, and by offering expressive service representations and management and control interfaces, which can enable optimal service instantiation considering customer and operator cost, performance and quality constraints.

An effort worth noticing is T-NOVA [9], [10], an EU-funded project proposing a VNF marketplace, where VNF providers will be making available their functions to be deployed over the infrastructure of a network or cloud service provider, developing the necessary support for VNF brokering, management and service delivery. The NF-as-a-Service approach of T-NOVA shares similarities with our work, but the overall target application and business environment is different. Our architecture is designed and optimized for content delivery (CDNaaS). Resource management and other relevant low-level deployment decisions are taken by the operator, based on customer demands and constraints, and accurate knowledge of the structure and conditions of its infrastructure.

III. A FRAMEWORK FOR CDNAAS PROVISION

A. *Features*

We present an architecture which allows a network operator to virtualize its CDN infrastructure and lease it to content providers on demand. The market particularly targeted by the ISP is that of small-to-medium content providers, but other options are possible. For instance, the virtual CDN

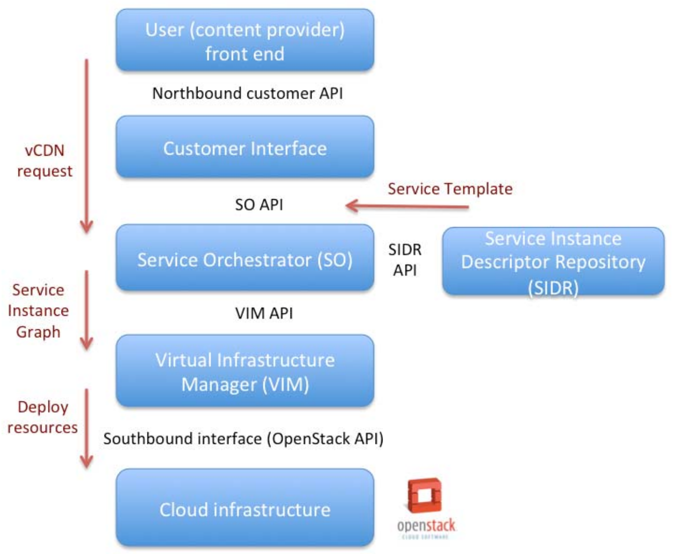
User (content provider) front end
Northbound customer API
vCDN request
Customer Interface
Service Template
SO API
Service Orchestrator (SO)
SIDR API
Service Instance Descriptor Repository (SIDR)
Service Instance Graph
VIM API
Virtual Infrastructure Manager (VIM)
Deploy resources
Southbound interface (OpenStack API)
Cloud infrastructure
openstack

Fig. 1. CDNaaS architectural components.

instantiated by a customer can be coupled with other CDN infrastructures; the flexibility offered by our design allows the customer to, e.g., use the leased infrastructure to respond to predicted traffic surges at specific regions, taking advantage of the network operator's regional presence. From the perspective of the network provider, our design allows for more efficient use of its infrastructure resources, compared to a less dynamic resource reservation model with static allocation of data center resources to clients.

B. *Architecture*

Our design involves various functional blocks, communicating via well-specified interfaces. This decouples their operation from any physical location, allowing the CDNaaS provider to execute any of these blocks autonomously as virtual functions over its own (or any) cloud infrastructure. One of our core design principles is to expose open APIs to customers, but also among the components of our architecture, and to design with extensibility in mind, so that our scheme can be extended towards a generic Any-as-a-Service model. In this section we provide the main components of our CDNaaS architecture (Fig. 1), their functionality and their interactions.

1) Customer Interface Manager: Our system provides a RESTful northbound API, through which customers can request to deploy a virtual CDN over the telco cloud. This API exposes information on the available services and regional presence, abstracting information about the underlying network and cloud infrastructure, and mediating the communication of customers and the service orchestrator. Using the northbound API, a customer can express its service requirements per region that it wishes to cover, and in particular (i) demand specifications, i.e., how many clients it wishes to server per region, and what is the service lease duration, and (ii) quality specifications, i.e., a target Quality of Experience (QoE) rating, desired response times, target uptime, etc.

2) Service Instance Descriptor Repository (SIDR): Each service supported by our scheme has some inherent requirements and constraints. These are encoded in a service instance template, which provides information such as the minimal number of VNFs that need to be deployed for a service instance, constraints with respect to the processing and memory resources that should be available to each VM, etc. Although we mainly focus on the provision of a virtualized CDN service, our design includes an extensible service template description language which can support further services. The SIDR component stores these service instance templates and provides them to the Service Orchestrator (SO) to drive service deployment decisions.

3) Service Orchestrator: The Service Orchestrator coordinates vCDN service deployment. After receiving a request from the client, it derives an appropriate VNF placement taking into consideration (i) the inherent service requirements, as expressed in the service instance template, (ii) the client demand and quality specifications, included in the service request, and (iii) its operational capacity. The result of the execution of the VNF placement algorithm is a Service Instance Graph (SIG), which maps VNF instances (VMs) to physical nodes. The SIG is then passed on to the Virtual Infrastructure Manager (VIM) for deployment.

4) Virtual Infrastructure Manager (VIM): The VIM component is responsible for the deployment of a service on the underlying cloud infrastructure. It receives the SIG that the SO derives following a customer request, and uses the southbound API of the cloud management software (in our case, OpenStack [11]) to set up and configure VNF instances.

C. vCDN service components and life cycle

At a high level, a CDN service is typically composed of one or more origin servers, where the content provider places content items, and a number of caches distributed across the CDN, which temporarily store content replicas and efficiently serve them to users, without having the latter retrieve content directly from the origin servers. Therefore, a mechanism to direct user requests to the optimal cache (e.g., the one located closest to them) is also necessary. As per our design approach, the main components of a CDN, i.e., origin servers, caches, DNS servers, request load balancers, etc., are implemented as VNFs.

The vCDN service which he have designed and implemented supports a two-level load balancing: A user's DNS request for the URL of a content item is resolved to the appropriate regional data center, based on the user's geographic location, as inferred from the latter's IP address (DNS geolocation). Then, we perform HTTP request load balancing across all cache VNF instances deployed in a regional data center.

Upon receiving a customer request for the creation of a vCDN, the SO runs an algorithm to calculate a VNF instance placement on the appropriate regional data centers in the form of a SIG, and uses the VIM API to request its deployment on the provider's infrastructure. The VIM, which is responsible for translating the abstract service representation (SIG) to an actual deployment, carries out the following steps: (i) It launches the origin server VNF(s), (ii) automatically configures one or more DNS VNF instances for request geolocation, (iii) configures cache VNF instances so that they proxy all user requests towards the content origin server VNF(s), and (iv) configures each region/host's load balancers.

Eventually, the customer entry point, with connection information and the necessary credentials for the origin server, is included in the northbound customer API response.

The vCDN service is terminated and the respective resources are released either automatically, when the lease specified in the customer request expires, or when the customer explicitly requests its termination using the northbound API.

D. Implementation

As a proof of concept, we have implemented our CDNaaS architecture on top of OpenStack [11]. VNF instances (origin servers, caches, DNS servers) are executed as Debian Linux virtual machines on kvm [12] hypervisors (compute nodes, in the OpenStack terminology). We are using *nginx* [13] to implement HTTP server/caching functionality, a choice motivated by its wide adoption.

The components of our scheme (customer interface, SIDR, SO, VIM) have been implemented in Python and communicate over HTTP. Our RESTful northbound API has also made it straightforward to create a web-based customer front end.

For VNF placement, we have designed a simple greedy algorithm which aims at covering all regions requested by the client, evenly distributing cache VNF instances across them, and satisfying the constraints specified in the vCDN service template. Our ongoing work focuses on devising measurement-driven algorithms for optimal VNF placement, using the results we present in Section IV. Note that our software architecture allows for the integration of VNF placement algorithms in a pluggable manner.

IV. Testbed experiments

A. Performance metrics

For service dimensioning purposes, and given a customer's specific performance demands, it is necessary for the CDNaaS provider to be aware of the associated service delays and request throughput, given a specific configuration and conditions. Service performance has many factors affecting it: Network capacity and conditions, resources (CPU, memory, storage) allocated to the service, current demand (concurrent number of users accessing the service), the specific virtualization technologies and their configuration, the software used to implement a service, the cloud platform, etc.

In this experimental study, we focus on performance mainly as a function of the available processing resources. Our processing unit (or, otherwise put, our unit of scaling) is a virtual CPU (vCPU), which in our case is a single CPU core. Under various experimental configurations, we measure the following quantities: (i) the time it takes to instantiate a service, (ii) the HTTP/caching server's response times, and (iii) the request throughput, i.e., the number of requests per second that a server can process.

B. Virtualization technologies and testbed configuration

We perform a comparative study of two candidate technologies to implement VNFs, namely virtualization and containerization. In the first case, we use virtual machines on top of the kvm [12] hypervisor, and, in the second, docker containers [14]. Note that both technologies are supported by OpenStack, our cloud computing software of choice. For a comparison of the features and the internal workings of the two technologies the reader is referred to the work of Dua et al. [15]. Felter et al. [16], on the other hand, present a thorough experimental evaluation of their associated overhead for specific applications.

We execute our tests on an HP Z800 workstation with a 16-core Intel Xeon processor and 16 GB of RAM, running Ubuntu 14.10. We are benchmarking the performance of the popular *nginx* HTTP server, which is the technology we are using in our proof-of-concept CDNaaS implementation. We have carefully tuned nginx and the operating system (both host and guest) for high performance and in order to alleviate network and I/O bottlenecks. Nginx works by spawning a number of worker processes, which handle user requests. The optimal strategy is to launch one worker per CPU core available. To deal with large numbers of concurrent users, we increased the maximum number of allowed concurrent connections per worker (and the respective operating system limits on the number of open file descriptors), and set the `tcp_tw_reuse` operating system option, to allow for reusing sockets that are in the TIME_WAIT state.

One of the aspects that we wish to quantify is the scalability of the service as a function of the CPU resources available. Intuitively, this should scale linearly with the number of available virtual CPUs (cores); our intuition is experimentally verified. To achieve the appropriate level of isolation, for each of our experiments, we pin the server VM/container and the load generation tool to separate core sets using the *taskset* utility, and give the highest priority to the respective processes using the *nice* command.

To benchmark our server, we use the *weighttp* tool [17]. Being multi-threaded, it takes advantage of the availability of multiple CPU cores, ensuring that the HTTP traffic generator does not become the performance bottleneck.

Before the experiments, we also verified that network I/O is not the bottleneck. For the experiments using kvm, we activated the *vhost-net* module, which enables handling packets between the guest and the host in-kernel, thus reducing context switching and packet copying overheads. By using two virtual Ethernet interfaces on the guest and connecting each one of them to a tap device on the host, we measured (using *iperf*) a 30 Gbps aggregate host-to-guest TCP throughput, enough to saturate our HTTP server. For docker, to avoid the overhead associated with Network Address Translation (NAT) in host-to-guest communication, we launched containers using *host*-mode networking. Thus, containers have direct access to the host networking stack.

C. Startup times

One of the advantages of container technologies is that they are more lightweight compared to VMs. To start an HTTP server/cache VNF instance hosted in a VM typically requires booting a full operating system. By startup time we define the time interval from launching a VM/container until it is capable of responding to HTTP. To measure it, we start the VM or the container and simultaneously scan TCP port 80 (where our web server listens to) using *nmap* [18] from the host, until the port is detected open (i.e., the HTTP server is ready). We found that the average startup time for kvm was 11.489 s, while for docker it was 1.129 s (mean values of 50 experiments).

It should be noted that in a cloud environment, booting a VNF instance may also involve transferring the VM image or container from the image store to the actual host(s) where it will be executed.

D. Request throughput

An important aspect of HTTP server performance is the rate at which they can serve user requests. Being able to estimate the request throughput of a VNF instance with specific processing characteristics allows the CDNaaS provider to calculate the number of instances to deploy to cater for the demand of a specific customer, and appropriately respond to demand dynamics by up/down-scaling the deployment. Request throughput is a function of the processing capabilities of the server, its software architecture, the number of parallel users accessing the server, the size of the files to serve, and its disk and network I/O capacity.

We carried out a set of experiments to measure how our nginx-based HTTP server/cache VNF scales with the CPU processors available and the number of concurrent users. In our first test, to study these two properties in isolation, we minimized the effect of disk and network I/O by having HTTP clients request a special URI which corresponds to a single-pixel transparent GIF (the response body amounts to 43 bytes); this is a resource embedded in the HTTP server and causes zero disk I/O. We then emulated parallel users by varying the number of concurrent HTTP sessions on the client side (weighttp). Each emulated user constantly performed HTTP requests and we measured the HTTP request throughput when 1 or 2 CPU cores were allocated to the server. The results of this test are shown in Fig. 2. We notice that request throughput scales roughly linearly with the number of CPU cores.[1] Interestingly, we notice that docker achieves approximately 10k requests/s more than kvm for large numbers of parallel users. Each point in the figure represents a request throughput value calculated from the execution of 100 million requests.

Since one of the target applications of our CDNaaS scheme is video delivery, we further experimented with HTTP object sizes which correspond to a video service. To select realistic

[1]Although the figures show only the case for 1 vs. 2 vCPUs, our experiments with higher numbers of vCPUs demonstrate the same linear scaling.

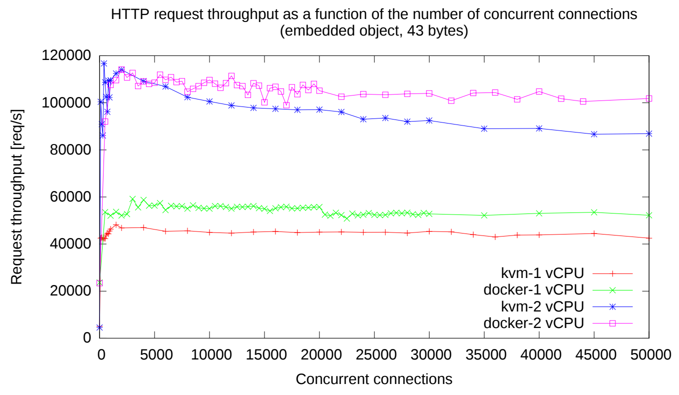

Fig. 2. HTTP request throughput for increasing numbers of parallel connections. We compare the use of docker and kvm and the impact of utilizing more CPU resources.

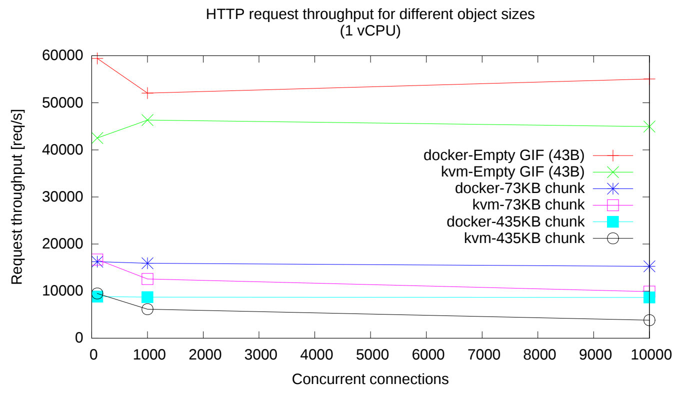

Fig. 3. HTTP request throughput for various object sizes and for increasing numbers of concurrent connections.

such sizes, we encoded a 720p HD video using H.264/AVC, and prepared it for HTTP adaptive streaming delivery[2] using GPAC [19]. Average chunk sizes for our test video were 73 KB and 435 KB for the low and high quality representations respectively (average bitrates of 250 Kbps and 1.7 Mbps). We therefore varied the number of concurrent connections across 100, 1000, and 10000, and measured request throughput when clients were requesting a 73 KB and a 435 KB file.

Fig. 3 presents the achieved HTTP request throughput when all clients are requesting (i) the minimal embedded object (empty GIF, 43 bytes), (ii) 73 KB video chunks (low quality), and (iii) 435 KB video chunks (high quality). We notice the expected response rate decrease as the size of the requested objects increases. One vCPU unit is capable of sustaining a high-quality video request throughput of $\sim$8600 requests/s when the HTTP server is containerized, compared to $\sim$3800 requests/s when it is run in a kvm VM for 10000 parallel HTTP connections. We witnessed a proportional performance increase when the HTTP server was using 2 cores; we have omitted these data from the figure for reasons of clarity.

E. Response times

Our third performance metric is related with user experience. We measured HTTP response times as a function of the virtualization technology used, the number of parallel connections, and CPU resource availability. We launched 90% of the concurrent HTTP connections using *weighttp* and we recorded response times for the remaining 10% using *ab* [20], due to its latency measurement capabilities. All figures of this section present empirical Cumulative Distribution Functions (CDF) of response times; each point represents the percentage of requests which were served in a time less than or equal to a specific value on the x-axis.

[2]With Dynamic Adaptive Streaming over HTTP (DASH) technologies, multiple quality/bitrate representations of the same video are stored on a plain HTTP server. A video is segmented in chunks, and a Media Presentation Description (MPD) file describes chunk information. The client receives the MPD file and proceeds by retrieving the video chunk-by-chunk, potentially switching among available qualities.

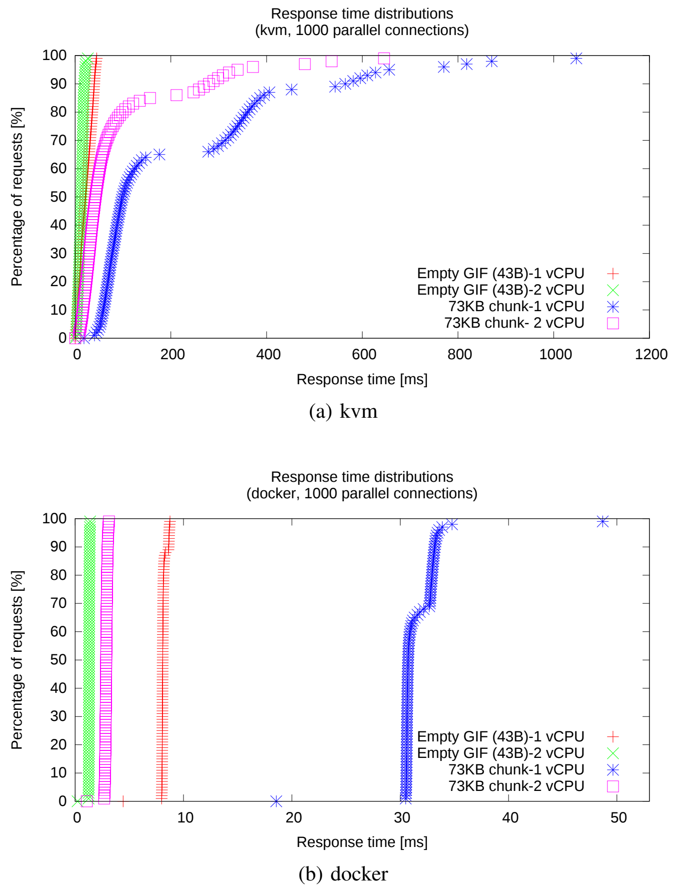

Fig. 4. Response time distributions for different sizes of the requested objects.

Fig. 4 compares a virtualized (kvm) to a containerized (docker) HTTP server. Docker achieves lower response times, since it incurs less overhead for interacting with the operating system and for network I/O (especially given our host-mode configuration, which gives it native access to the host's networking stack). When clients request larger objects, response times increase. This increase is noticeable in both cases. For kvm (Fig. 4a), it reaches more than 200 ms for 35% of the requests for a 73K file when the server VM runs on a single

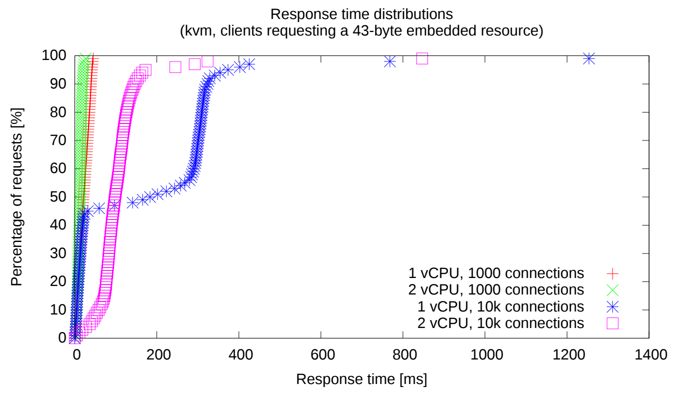

Fig. 5. Response time distributions for a server hosted in a kvm VM as the number of available vCPUs scales, for different numbers of parallel HTTP connections. Clients request a 43-byte embedded object.

CPU core and there are 1000 concurrent connections (blue curve, star points). Docker (Fig. 4b) performs better: In the same settings, 98% of the requests are completed in less than 35 ms (blue curve, star points). Scaling up processing capacity by adding a second vCPUs significantly reduces latency in both cases (square points).

This performance improvement in terms of latency when we scale up CPU resources is more evident as server load grows from 1000 to 10000 parallel HTTP connections. As shown in Fig. 5, an increase in the number of parallel connections results in higher latencies, as a result of the connection management overhead on the server.

F. Uses of our results

Our experiments offer insight on the capabilities of the enabling technologies of our CDNaaS vision. Our findings indicate that virtualization comes with a larger overhead compared to using containers. This observation is in line with other works in the literature [16]. However, it should be noted that containerization technologies for cloud computing have only recently gained popularity, while virtualization can be considered more mature in this context.

There are two main usages of our quantitative results: (i) Optimally deciding on the amount of resources to deploy and their configuration to match customer demand, and (ii) deriving pricing strategies for the offered vCDN service.

In the first case, using the CDNaaS API, a customer can specify the target number of users accessing its service per region, as well as a target QoE level which can be expressed in terms of desirable response times, and information on the content to be served (e.g., high definition video with specific video segment size statistics). With this information in place, and with measurement data on the performance capabilities of the underlying server and virtualization technologies, the CDNaaS provider can decide how many vCPUs hosting VNF instances should be deployed per region, to cover the customer's request with a specific level of confidence.

In the second case, the CDNaaS provider can use such measurements to estimate the amount of physical and virtual resources customer requests translate to. This information can then be used to derive service pricing policies, but also to construct appropriate Service Level Agreements (SLAs).

V. Conclusion

We presented the design and implementation of an architecture for on-demand virtual CDN service instantiation on top of a telco cloud. Our system offers a rich northbound customer API, and allows for the application of sophisticated service placement algorithms, given the inherent awareness of the provider of its current network conditions and capacity, and its presence close to end users. To get a better understanding of the potential of virtualization and containerization as key enabling technologies to support our CDNaaS vision, we have carried out extensive testbed experiments. Such a quantitative evaluation is critical for our ongoing research efforts on service dimensioning and measurement-driven resource allocation algorithm design, and on deriving CDNaaS pricing strategies.

References

[1] R. T. Ma, J. C. Lui, and V. Misra, "On the evolution of the Internet economic ecosystem," in *Proc. WWW*, 2013.

[2] H. Lee, D. Lee, and Y. Yi, "On the economic impact of telco cdns and their alliance on the cdn market," in *Proc. IEEE ICC*, 2014.

[3] N. Kamiyama, T. Mori, R. Kawahara, S. Harada, and H. Hasegawa, "ISP-operated CDN," in *Proc. IEEE INFOCOM Workshops*, 2009.

[4] S. Spagna, M. Liebsch, R. Baldessari, S. Niccolini, S. Schmid, R. Garroppo, K. Ozawa, and J. Awano, "Design principles of an operator-owned highly distributed content delivery network," *IEEE Communications Magazine*, vol. 51, no. 4, pp. 132–140, 2013.

[5] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[6] F. Z. Yousaf, J. Lessmann, P. Loureiro, and S. Schmid, "SoftEPC - dynamic instantiation of mobile core network entities for efficient resource utilization," in *Proc. IEEE ICC*, 2013.

[7] T. Taleb, M. Corici, C. Parada, A. Jamakovic, S. Ruffino, G. Karagiannis, and T. Magedanz, "EASE: EPC as a service to ease mobile core network deployment over cloud," *IEEE Network*, vol. 29, no. 2, pp. 78–88, 2015.

[8] T. Wood, K. Ramakrishnan, J. Hwang, G. Liu, and W. Zhang, "Toward a software-based network: integrating software defined networking and network function virtualization," *IEEE Network*, vol. 29, no. 3, pp. 36–41, 2015.

[9] FP7 T-NOVA. [Online]. Available: http://www.t-nova.eu

[10] G. Xilouris, E. Trouva, F. Lobillo, J. Soares, J. Carapinha, M. McGrath, G. Gardikis, P. Paglierani, E. Pallis, L. Zuccaro, Y. Rebahi, and A. Kourtis, "T-NOVA: A marketplace for virtualized network functions," in *Proc. EuCNC*, 2014.

[11] OpenStack - open source cloud computing software. [Online]. Available: https://www.openstack.org/

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proc. Linux Symposium*, 2007.

[13] nginx. [Online]. Available: http://nginx.org

[14] Docker. [Online]. Available: https://www.docker.com/

[15] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *Proc. IEEE IC2E*, 2014.

[16] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," IBM Research, Tech. Rep. RC25482, July 2014. [Online]. Available: http://goo.gl/ytEvt9

[17] weighttp. [Online]. Available: http://redmine.lighttpd.net/projects/weighttp/wiki

[18] Nmap - the network mapper. [Online]. Available: https://nmap.org/

[19] GPAC multimedia open source project. [Online]. Available: https://gpac.wp.mines-telecom.fr

[20] ab - Apache HTTP server benchmarking tool. [Online]. Available: https://httpd.apache.org/docs/2.2/programs/ab.html