Nesting Containers for Faithful Datacenters Emulations

Tommaso Caiazzi*, Mariano Scazzariello†, Samuele Quinzi*,
Lorenzo Ariemma*, Maurizio Patrignani*, and Giuseppe Di Battista*
*Roma Tre University – Rome, Italy †KTH Royal Institute of Technology – Stockholm, Sweden

***Abstract*— Datacenters are a critical part of the Internet infrastructure as they guarantee efficient deployment of a wide range of services. Since a considerable amount of datacenter failures is caused by software bugs and configuration errors, the management and testing of these networks is a crucial task. In this field, emulation-based digital twins have proven their effectiveness. To faithfully emulate the typical three layers hierarchy, composed of physical servers, virtual machines, and containers, the support for nested virtualization is a fundamental requirement. Further, the emulation of hyper-scale datacenters needs to leverage on horizontal scaling over a cluster of nodes. Existing container-based proposals do not meet both requirements. On the contrary, existing VM-based proposals meet such requirements, but they need complex configurations and high resource demands. We propose a container-based framework to faithfully emulate datacenters. This is a fundamental building block for designing datacenter digital twins, that would allow testing of real software implementations in a lightweight, scalable, and easily configurable environment.**

***Index Terms*—Datacenters, Network Testing, Nested Virtualization, Digital Twins**

I. Introduction

Nowadays, due to the exponential growth in the volume of data to be handled, cloud providers were forced to scale their infrastructures to global networks of warehouse-sized datacenters (DCs). This sudden growth also came with several management difficulties. Cloud providers usually host different services belonging to different customers in a complex virtualized environment, implementing a virtual network on top of the physical hosting layer. Furthermore, management operations such as deployment, service migration, and version updating must minimally impact the production environment.

In the scope of DC testing and management, digital twins are an emerging trend to support such operations, and have proven effective for testing both the physical and software environment. A *digital twin* (DT) of a DC can be used to simplify management operations by reproducing, or even anticipating through simulations, events happening in the physical counterpart. However, simulation-based systems cannot account for the real software running in the infrastructure. In fact, even if simulation allows to model all the system's aspects, including physical ones, it assumes a bug-free environment, and can only assess correct execution and expected error handling. Several Over-The-Tops (OTTs) state that a considerable amount of failures in their DCs is caused by software bugs [1]–[3], that usually cause million-dollar losses [4]. To mitigate unexpected outages, faithful emulation is needed. Network emulators allow to run real software such as routing daemons, network functions and services, making it possible to test software behaviour before it is deployed in the production environment. *Nested network emulation* is a crucial paradigm to faithfully emulate a DC. In fact, a server in a DC has typically two layers of virtualization: the first composed of several virtual machines (VMs), while the second comprises applications running in such VMs (usually containers). To faithfully reproduce the real behaviour of this environment, it is important that such layers are kept as-is in the emulated counterpart.

In order to create a faithful DT of a DC, a network emulator should (i) allow the execution of real software (including all nested services) through nested virtualization, (ii) be lightweight and scalable, and (iii) be easy to configure. Nowadays, there are no such solutions available in the literature. In this paper, we propose an open-source container-based framework to faithfully emulate DCs by supporting nested virtualization. The framework can support the creation of emulation-based DTs, satisfying all the above requirements.

II. Motivations and Related Work

In this Section, we expose several motivations and related work supporting the need of fully emulated digital twins.

Manage and test a datacenter is difficult and digital twins are effective. DCs' management and testing is a defiant task. Common characteristics of DCs include a warehouse-size dense network topology, the presence of heterogeneous services and technologies, and a massive exploitation of virtualization. In literature, DTs have been proven effective to accomplish these tasks [1], [5].

Simulation-based systems cannot test real software implementations. Simulation allows to cover all the system's aspects, including physical ones, by creating a model of the system and the operations it can perform. There are several proposals that leverage simulation to create a DT of a DC network. Such systems allow to model the physical environment (*e.g.*, energy costs monitoring or cooling simulations) for planning purposes, or to test network configurations using formal verification. As an example, NetGraph [5] is a system for creating a DC's DT, by parsing real network configuration

files. However simulation, assuming a bug-free environment, is unsuitable for testing real software.

Emulation-based systems are suitable for testing real software implementations. Emulation aims at accurately reproduce the behaviour of a system, running its real software implementation. In computer networks, emulation environments are often based on virtualization. This allows to obtain an isolated environment that can be exploited to test software implementations. Generally, it is possible to distinguish two types of virtualization: full virtualization, that leverages VMs and comprises hardware emulation, and OS-level virtualization, that is based on containers and only virtualizes code, runtimes, system tools, system libraries and settings.

Emulators need nested virtualization to faithfully reproduce the datacenter infrastructure. Real DC networks typically have at least two layers of virtualization: the first layer is composed of VMs while the second layer usually consists in containers running microservices and applications. Common DCs' software stacks (*e.g.*, orchestrators) assume to operate in this layered architecture. Hence, in order to test and support these applications, it is important to reproduce the infrastructure as-is. Thereby, a minimum of three virtualized layers are needed: the first one for physical hosts, and two nested layers for the virtualized DC structure emulation (*i.e.*, VMs and containers). Nested emulation is then crucial to faithfully emulate datacenters.

VM-based digital twins are computationally heavy and difficult to configure. VMs are a virtualization paradigm that can be exploited to create DTs. Despite they are able to fully emulate a device, including its hardware, deploying thousands of VMs involves such a memory overhead that makes it costly to scale up [6]. Moreover, VMs are usually more difficult to configure compared to containers. Many use cases do not require such complexities (*e.g.*, network and software configuration testing). CrystalNet [1] is a VM-based network emulator developed by Microsoft for creating cloud-scaled DC's DTs. It leverages the Azure cloud infrastructure to deploy a sufficient number of VMs to reproduce the physical network. Since CrystalNet uses the Azure APIs, it is bounded to this specific execution environment. Moreover, being based on VMs, it requires a great amount of resources.

Existing container-based emulators are not suitable to create a hyper-scale datacenter digital twin. Container-based emulators are clearly lighter than VM-based ones. Deploying a container brings faster startup times compared to VMs. In the literature, several examples of container-based emulators are present [7]–[10]. Although some of them can scale up to relatively large network topologies, most of them are limited by not supporting nested virtualization [7]–[9]. NestedNet [10], is the only emulator designed to create nested virtual networks. However, it cannot go further the first nested layer. Also, it is not suitable for the emulation of large networks since it is does not run on a distributed environment.

In the remaining Sections, we address the following question: *"Can we design a container-based framework that can faithfully emulate the three-layers structure of hyper-scale DCs to support DTs' deployment?"*

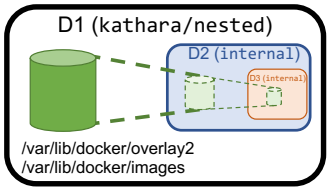


Fig. 1. Bind-mounting the local Docker repository in nested containers.

III. Achieving Design Goals

To obtain a framework suitable for faithfully testing DC infrastructures, our implementation aims to achieve the following goals: (a) **Scalability**; (b) **Nested virtualization**. We develop such a framework using open-source, state-of-the-art technologies. We choose Kathará [7] since it is able to scale up to thousands of virtual devices, thanks to the Megalos [11] extension. Hence, we only need to extend the Kathará network emulator to support nested virtualization in order to meet all the required goals. As Kathará devices are emulated through Docker, it is possible to leverage the Docker-in-Docker (DinD) [12] image, that allows to run the Docker Engine into a Docker container. Since DinD was originally designed to help with the development of Docker itself, the Docker daemon inside a DinD container is agnostic with respect to its execution environment. Each Docker container uses a specific Docker image, that is a read-only template with instructions for creating a Docker container, that can include several elements (*e.g.*, configuration files and libraries). When a container is created and its associated image is not locally found, the Docker daemon downloads it from a remote repository, and stores it locally. By default, each container that runs the DinD image has its own filesystem that is not shared with any other container or with the host. This means that each Docker daemon inside a container created with DinD tries to download from the remote repository images not found locally, storing them inside its filesystem. This procedure has two drawbacks: (i) the startup time heavily depends on the Internet connection speed; (ii) the same image is re-downloaded several times.

To overcome these limitations, we modify the DinD image to automatically load the Docker images from the host local repository into each device of the hierarchy. In this way, when a device is started, the Docker image is already found locally and it is not re-downloaded. In particular, we build a Docker image, called `internal`, which contains the Docker daemon, the Docker images to pre-load (in this case, `kathara/frr`), and the Kathará network emulator. Then, we craft a wrapper Docker image (`kathara/nested`) that, during the building phase, copies the layers of `internal` into the local Docker image repository folder. The result is that when a container runs the `kathara/nested` image, it already contains the software stack and the pre-loaded Docker images installed in

`internal`, and the `internal` image itself. This solution allows a single level nesting similar to DinD, with the addition of the pre-loaded images that avoids to re-download them. In order to support multiple nesting levels without the same overheads described above, we modified Kathará as follows. When Kathará deploys a device $\mathbf{D}_2$ inside $\mathbf{D}_1$ (with $\mathbf{D}_i$ being a device at depth i) running the `kathara/nested` image, $\mathbf{D}_2$ bind-mounts the Docker image repository folder of $\mathbf{D}_1$. This means that the Docker daemon inside $\mathbf{D}_2$ can directly use the Docker images of $\mathbf{D}_1$, including the `internal` image itself. Hence, each device $\mathbf{D}_n$ (with $n > 2$) that deploys a container with nested virtualization enabled, uses the `internal` image and bind-mounts the image repository folder of $\mathbf{D}_{n-1}$. We illustrate an example of this procedure in Fig. 1.

Notice that bind-mounting the parent Docker image repository presents security issues. For example, if a container in the hierarchy makes changes on one of the pre-loaded images, all the devices $\mathbf{D}_n$ (with $n >= 2$) are affected by such changes. However, the goal of this system is to only provide an isolated environment to faithfully reproduce a DC infrastructure for testing purposes, so there are no malicious accesses to this virtual network. Moreover, the possible changes are not reflected on the host filesystem, since such changes can only affect the volatile Docker image repository folder of the device $\mathbf{D}_1$ running the `kathara/nested` image.

For more details on the procedure for creating the `kathara/nested` image and on how customize it, see [13].

IV. Evaluation

In this Section, we show and discuss the performance of our framework. Source code and scripts to perform the experiments are available at [14].

Our evaluation is based on two experiments. The first aims at measuring the performance overhead introduced by the nested virtualization with different depths. The second experiment shows the startup times needed for deploying different-sized fat-tree topologies, which are commonly used in DCs [15]. To configure the fat-trees, we modified VFT-Gen [16], a tool for deploying virtual fat-tree topologies leveraging on Kathará. We add the support to the nested virtualization, including the possibility of deploying VMs inside servers, and containers inside VMs.

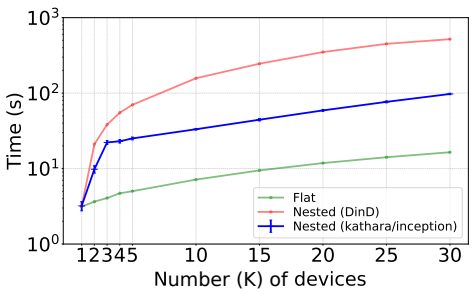
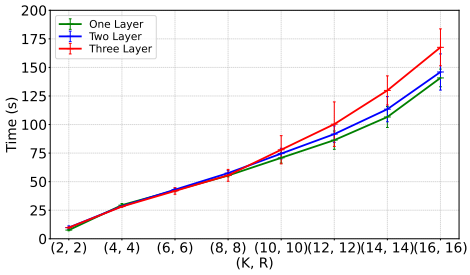
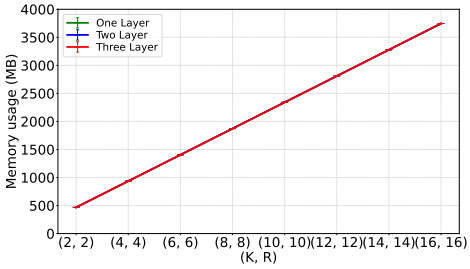


Fig. 2. Startup time comparison among: a flat network, the `kathara/nested` image, and the official DinD image.



(a)



(b) The lines overlap since memory requirements is less than the minimum memory assigned by Kubernetes.

Fig. 3. Startup time (a) and startup memory usage (b) needed for deploying different fat-tree topologies. The three layers $(16, 16)$ has overall 80 routers, 256 servers, 2560 VMs and 25600 containers.

In the following we answer three questions: Q1) *"What is the overhead introduced by the nested virtualization?"* Q2) *"How much time is needed for deploying a DC?"* Q3) *"How much memory is needed to emulate a DC?"*

Q1) Microbenchmark: container-based nested virtualization introduces a reasonable overhead. One may wonder if container-based nested virtualization introduces significant overheads with respect to flat networks. To demonstrate the opposite, we tested several flat networks versus nested chains, both composed of a variable number k of devices. We define a nested chain as a network scenario composed of a hierarchy of k devices (called $\mathbf{D}_1, \ldots, \mathbf{D}_k$). In all hierarchies, $\mathbf{D}_1$ is located into the physical host, while device $\mathbf{D}_{i+1}$ $(i = 1, \ldots, k-1)$ is located into device $\mathbf{D}_i$. We performed each experiment five times, using a commodity workstation equipped with Intel® Core-i5™ 10600 CPU and 16 GB of RAM. In Fig. 2, the x-axis represents the number k of devices (or the depth of the k-th chain), the y-axis shows the startup time of the network scenario in seconds (using a log-scale). Although, network nesting introduces an exponential (linear in the log-scale) overhead with respect to the startup time of the flat network scenario, in real-world scenarios, where nesting levels are only a few, the absolute value of the overhead is acceptable. Moreover, such overhead is much lower than the one introduced by the startup of a single VM in public clouds (more than 30 secs) [17]. These data highlight the advantages of adopting our framework to recreate the servers-VMs-containers hierarchy in DCs' DTs.

Q1) Microbenchmark: `kathara/nested` has faster startup times with respect to standard DinD. To demonstrate that pre-loading Docker images speeds up the deploy-

ment time, we used the same nested chains and the same execution environment of the experiment above. As shown in Fig. 2, the `kathara/nested` image (blue curve), combined with the bind-mount of the parent Docker images, retains better startup times with respect to the official DinD [12] (red curve), that must re-download the required images at each level of the chain. The overhead of our solution is clearly highlighted in the 1-st, 2-nd, and 3-rd chains. We have that between the 1-st and the 2-nd chain, the overhead is introduced by loading the `kathara/nested` image in $\mathbf{D}_1$, and by the first bind-mount of the $\mathbf{D}_1$ image repository in $\mathbf{D}_2$. The additional overhead between the 2-nd and 3-rd chain is caused by the first bind-mount in a different image (`internal`) of the $\mathbf{D}_2$ image repository in $\mathbf{D}_3$. From the 4-th chain, the overhead is negligible since $\mathbf{D}_4$ bind-mounts the $\mathbf{D}_3$ image repository folder, which has already been bind-mounted for the `internal` image. In conclusion, our technique shows much better startup times with respect to the startup time of a standard VM in the cloud or of a nested container using DinD.

Q2) Deploy a fully-fledged DC in three minutes. In this experiment we show how our framework can deploy real-world-size fat-trees in a few minutes. Following the terminology presented in [16], [18], the fat-tree topology has two fundamental parameters: K, that is the number of north/south switch ports, and R, the redundancy factor (*i.e.*, the number of links between a node in the aggregation layer and the top nodes of a PoD). We performed the experiments for all the even values of K in the range 2–16, always setting $K = R$. All the experiments are performed on a local virtual cluster composed of 21 VMs, each with 4-core vCPUs and 8GB of vRAM. Fig. 3(a) shows the results. The green curve represents the startup time of different fat-tree topologies without nested virtualization (only physical devices are emulated). The blue curve represents the startup time with one level of nesting, deploying 10 devices, representing VMs, for each server. The red curve represents the startup time with two level of nesting, also deploying 10 containers for each emulated VM in the first nested layer. The results confirm that the overhead introduced by the three layers of virtualization is minimal, since the three curves are almost overlapped. Furthermore, the overall startup time is low, since our framework allows to deploy a $(16, 16)$ fat-tree (composed of 80 routers, 256 servers, $2.56k$ VMs and $25.6k$ containers) in about 180 seconds, which is the time needed for deploying about 6 VMs on common clouds [17].

Q3) Minimal startup memory footprint. Fig. 3(b) illustrates the average memory usage of a VM that compose the cluster for the same experiments performed in Fig. 3(a). Notice that the deployed topologies have the Border Gateway Protocol (BGP) configured as routing protocol, but do not run any other service. For this reason, the three curves are completely overlapped since the memory requirements for running the nested containers is less than the minimum amount of memory assigned by Kubernetes to each emulated device at level 1. Moreover, the used memory is less than the 0.5% of the worker total memory in the $(16, 16)$ topology, leaving enough space for running services inside the virtual network.

V. Conclusions

We propose a framework for container-based nested virtualization, which is fundamental to faithfully emulate the servers-VMs-containers hierarchy. We showed that our framework allows having a real-life-size datacenter up and running in a few minutes, with acceptable resource requirements. We believe that our framework can be exploited in many use cases which require to test real software implementations in an isolated virtual environment. It provides a friendly language that can support the creation of lightweight digital twins, allowing to quickly test software implementations, network configurations, and interactions between services.

We leave as future work to extend the framework by creating a layer of connection between the emulated datacenter and its physical counterpart.

Acknowledgement. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889).

References

[1] H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully Emulating Large Production Networks," in *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, October 2017, pp. 599–613.

[2] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang, "NetPilot: Automating Datacenter Network Failure Mitigation," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, aug 2012.

[3] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu, "A Large Scale Study of Data Center Network Reliability," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 393–407.

[4] Uptime Institute, "Uptime Institute Global Data Center Survey Results 2022 ," 2022, https://uptimeinstitute.com/resources/research-and-reports/uptime-institute-global-data-center-survey-results-2022.

[5] H. Hong, Q. Wu, F. Dong, W. Song, R. Sun, T. Han, C. Zhou, and H. Yang, "NetGraph: An Intelligent Operated Digital Twin Platform for Data Center Networks," in *Proceedings of the ACM SIGCOMM 2021 Workshop on Network-Application Integration*, ser. NAI'21. Association for Computing Machinery, 2021, p. 26–32.

[6] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, "Performance Evaluation of Docker Container and Virtual Machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020, third International Conference on Computing and Network Communications (CoCoNet'19).

[7] M. Scazzariello, L. Ariemma, and T. Caiazzi, "Kathará: A Lightweight Network Emulation System," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.

[8] Mininet Team, "Mininet: An Instant Virtual Network on your Laptop (or other PC)," http://mininet.org.

[9] Nokia, "containerlab," https://containerlab.dev/.

[10] X. Zhang, N. Prabhu, and R. Tessier, "NestedNet: A Container-based Prototyping Tool for Hierarchical Software Defined Networks," in *2020 International Workshop on Rapid System Prototyping (RSP)*, 2020.

[11] M. Scazzariello, L. Ariemma, G. D. Battista, and M. Patrignani, "Megalos: A Scalable Architecture for the Virtualization of Large Network Scenarios," *Future Internet*, vol. 13, no. 9, 2021.

[12] Docker Inc., "Docker-in-Docker," https://hub.docker.com/_/docker.

[13] Compunet Research Group @ Roma Tre University, "kathara/nested image," https://github.com/KatharaFramework/Docker-Images/tree/kathara/nested.

[14] Compunet Research Group Roma Tre University, "Paper Experiments," https://github.com/KatharaFramework/Testbed.

[15] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, Oct 1985.

[16] T. Caiazzi, M. Scazzariello, and L. Ariemma, "VFTGen: a Tool to Perform Experiments in Virtual Fat Tree Topologies," in *IM 2021 - 2021 IFIP/IEEE International Symposium on Integrated Network Management*, 2021.

[17] J. Hao, T. Jiang, W. Wang, and I. K. Kim, "An Empirical Analysis of VM Startup Times in Public IaaS Clouds," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 398–403.

[18] T. Przygienda, A. Sharma, P. Thubert, B. Rijsman, D. Afanasiev, and J. Head, "RIFT: Routing in Fat Trees," Internet Engineering Task Force, Internet-Draft draft-ietf-rift-rift-16, Sep. 2022.