MeDICINE: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments

Manuel Peuster
Paderborn University
manuel.peuster@uni-paderborn.de

Holger Karl
Paderborn University
holger.karl@uni-paderborn.de

Steven van Rossem
Ghent University iMinds, INTEC
steven.vanrossem@intec.ugent.be

***Abstract*—Virtualized network services consisting of multiple individual network functions are already today deployed across multiple sites, so called multi-PoP (points of presence) environments. This allows to improve service performance by optimizing its placement in the network. But prototyping and testing of these complex distributed software systems becomes extremely challenging. The reason is that not only the network service as such has to be tested but also its integration with management and orchestration systems. Existing solutions, like simulators, basic network emulators, or local cloud testbeds, do not support all aspects of these tasks.**

To this end, we introduce *MeDICINE*, a novel NFV prototyping platform that is able to execute production-ready network functions, provided as software containers, in an emulated multi-PoP environment. These network functions can be controlled by any third-party management and orchestration system that connects to our platform through standard interfaces. Based on this, a developer can use our platform to prototype and test complex network services in a realistic environment running on his laptop.

I. Introduction & Motivation

The emerging trend of network function virtualization (NFV) promises a new level of flexibility for the upcoming 5th generation of networks. It turns network functionality, previously implemented as proprietary hardware boxes, into software artifacts that are executed in virtualized environments. Multiple of these virtual network functions (VNF) are then connected and create complex network services (NS), which are controlled by a management and orchestration (MANO) system [1] [2]. Such orchestrators do not only manage the deployment of network services but also automate operational management, e.g., service scaling.

Virtualized network services can be distributed in the network and different functions of a service can be executed in different points of presence (PoPs). These PoPs may be full-fledged data centers but also smaller sites, like network edge routers or base stations, which offer a limited amount of compute resources to run arbitrary functions.

In such an environment, the creation of network services becomes a complex software development process that consists of two main parts. First, the development of the service and its functions as such. Second, the integration of the service with a MANO system that manages the service during its lifecycle. The second part involves the implementation and test of management interfaces but also the design, implementation, and validation of service-specific management components, like auto-scaling rules or placement strategies [1]. This complicates the overall development process. To reduce this complexity, extended tool support is required to reduce time-to-market, save costs, and improve the quality of service.

978-1-5090-0933-6/16/$31.00 © 2016 IEEE

A special problem in this process is the lack of supporting tools to locally prototype or test complete network services in end-to-end multi-PoP scenarios. These tools should allow testing a network service as such, e.g., by sending generated traffic through it, but also validating its interaction with a MANO system, e.g., dynamic reconfiguration or placement strategies. This is not possible with existing approaches which either rely on local cloud testbeds that lack multi-PoP support, simulations that only execute simplified versions of network functions, or network emulation tools that do not offer interfaces to interact with MANO systems.

There are both simple and complex use cases for such a development support tool which motivate our proposed solution. These use cases can be divided into two categories. First, use cases that check the functionality of the network service as such (*NF-UC*); this can already be done with existing emulation solutions but requires considerable manual effort, e.g, for adapting the emulated services for production environments. Second, use cases that check the interoperation between network service and a MANO system (*MANO-UC*), which can today only be done with complex cloud testbeds. Examples for both categories are as follows.

*a) **NF-UC1 (Single VNF)**:* A network service developer wants to deploy and test single VNFs in a local test environment by sending some generated traffic through them. Such VNFs should be executed as containers so that the same container images can directly be deployed in a production environment. During the test, a developer wants to interact with the running VNFs to, e.g., change configurations or monitor their behavior.

*b) **NF-UC2 (Complex services)**:* A developer wants to test entire complex services consisting of several chained VNFs. A local test environment should be able to execute such complex services so that end-to-end tests, e.g., sending traffic through the service's chain, can be performed.

*c) **MANO-UC1 (Service management)**:* There is the need to validate the service behavior in dynamic deployments in which the service is modified at runtime by a MANO system. To do so, a test environment needs to be able to interface with existing MANO systems. An example is to

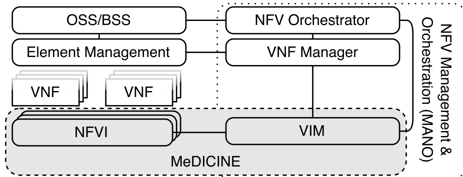


Fig. 1: The MeDICINIE platform in the simplified ETSI NFV reference architecture [3].

test reconfiguration mechanisms after adding additional VNF instances to the service (scale-out). Such tests should be performed in multi-PoP environments to also allow tests of placement optimization strategies and service management across multiple PoPs.

*d) **MANO-UC2 (Feedback-based autoscaling):*** A network service developer wants to test custom autoscaling approaches. To support this, connected MANO systems have to be able to collect feedback data, e.g., monitoring information, from services executed in the test platform.

To cover the previously described use cases and overcome the shortcomings of existing development support tools (Sec. II), we introduce *MeDICINE* (Multi Datacenter servIce ChaIN Emulator), a novel prototyping platform for network services. Our platform is able to execute production-ready network functions in realistic multi-PoP environments and allows standard MANO systems to control the deployment, like in a real-world system. Fig. 1 shows the scope of our solution and its mapping to the ETSI NFV reference architecture in which it emulates the network function virtualization infrastructure (NFVI) and the virtualized infrastructure manager (VIM).

The remainder of this paper is organized as follows. First, we compare existing simulation, emulation, and testbed solutions in Section II and describe our platform in Section III. Section IV presents first experimental results and demonstrates the platform. Section V concludes.

II. Related Work

NFV development support is still a novel research direction with a limited amount of existing solutions, most of them focusing on SDN debugging rather than on prototyping of complex network services [4]. Other approaches are based on simulations to test and validate management solutions, e.g., placement algorithms, but they only provide very limited realism since the simulated network functions are only proxies and not real implementations used in production [5] [6] [7]. VLSP [8] offers more realism but the tested network functions are still limited to simple Java programs and not real NF implementations.

Emulation tools, like Mininet, are able to execute any network function implementation in its own virtualized network namespace [9] [10] [11]. However, moving these network functions into a production environments is still a time-consuming task and these tools lack the possibility to emulate PoPs or cloud sites, e.g., they have no functionality to stop and remove hosts at runtime. The ESCAPE platform overcomes some of these limitations by combining a MANO system with multi-PoP support (including Mininet and OpenStack) but it does not target development support or prototyping tasks. Its main focus is on orchestration between non-emulated PoPs [2].

Real cloud testbeds, which might be installed on a single physical machine, are typically not able to run services in arbitrary network topologies [12]. And even if they do, they come with considerable management overhead and only a limited number of PoPs that can be used for tests [13].

TABLE I: Feature matrix of existing approaches

Feature	Simulations	VLSP [8]	Mininet [9]	ESCAPE [2]	DevStack [12]	Cloud testbeds	MeDICINE
production ready NFs	-	-	o	+	+	+	+
multi PoP	+	+	o	o	-	o	+
arbitrary topologies	+	+	+	o	-	-	+
realistic NF performance	-	-	o	+	o	+	o
explicit chaining support	o	o	-	+	-	o	+
MANO system integration	o	o	-	+	+	+	+
run offline/local	+	+	+	o	o	-	+
test/prototyping support	+	+	+	-	o	o	+

+ : fully supported, o : partly supported, - : not supported

Table I presents an overview of features needed for NFV development support and shows which of them are provided by existing simulation, emulation, and testbed solutions. It shows that all existing approaches lack some features and that *MeDICINE* is the only solution that can execute production ready NFs in an emulated multi-PoP environment.

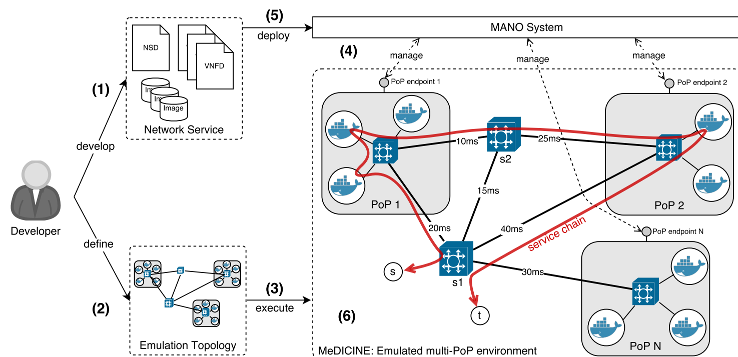
III. MeDICINE Platform

A. Background: Containernet

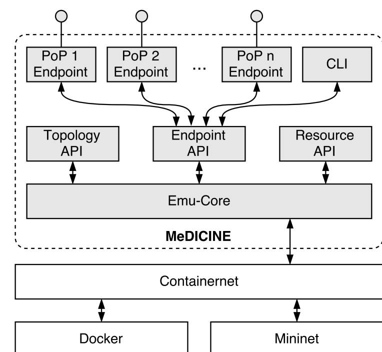
We base the implementation of *MeDICINE* on a tool called Containernet [14] which was also developed by us. It extends the Mininet emulation framework and allows us to use standard Docker containers as compute instances within the emulated network. An extension of Containernet to support full-featured VMs (qemu) is left for future work. Containernet allows adding and removing containers from the emulated network at runtime, which is not possible in Mininet. This concept allows us to use Containernet like cloud infrastructure in which we can start and stop compute instances (in form of containers) at any point in time. Another feature of Containernet is that it allows to change resource limitations, e.g., CPU time available for a single container, at runtime and not only once when a container is started, like in normal Docker setups.

B. Overview and Workflow

To fulfill the previously defined use cases, *MeDICINE* provides the following key features. First, it exploits Mininet's topology API to define arbitrarily complex multi-PoP environments with realistic link properties, like delay, bandwidth



(a) General idea and workflow of the system. Example of a running emulation environment with three PoPs, eight allocated compute instances executing VNFs, and a service chain setup consisting of three chained VNFs distributed across two PoPs through which generated traffic is sent from s to t.



(b) System architecture and components with N active PoP endpoints offering control interfaces to an external MANO system.

Fig. 2: The MeDICINE system

limitation, and loss rate. Second, it uses standard Docker containers to execute network functions, allowing a developer to directly deploy the prototyped container images to production PoPs after they have been tested locally. Third, it provides cloud-like interface endpoints, e.g., an OpenStack Heat-like interface, to control each emulated PoP in the platform. This allows developers to connect their local prototyping environment to existing MANO tools.

Fig. 2a shows the general idea of *MeDICINE* and depicts the high-level developer workflow. First, the service developer defines a network service, consisting of function (VNFD) and service (NSD) descriptors as well as Docker files or pre-built images that contain the network functions to be tested (1). The actual format of this network service and is descriptors depends on the used MANO system that deploys these services in our platform. Second, the developer defines a multi-PoP topology on which he wants to test the service (2) and starts the *MeDICINE* platform with this topology definition (3). After the platform has been started, the developer connects the MANO system of his choice to the emulated PoPs by using a flexible endpoint API (4) described in Sec. III-E. Now, the network service can be deployed on the platform by pushing it to the MANO system (5) which starts each network function as a Docker container in an emulated PoP, connects it to the emulated network, and sets up its forwarding chain. Finally, the service is deployed and runs inside our platform (6).

In this stage, a developer can directly interact with each running container through Containernets's interactive command line interface (CLI), e.g., to view log files, change configurations, or run arbitrary commands, while the service processes traffic generated by tools like *iperf*. Furthermore, a developer and the MANO system can access arbitrary monitoring data generated by the platform or the network functions.

C. System Architecture

The system design of *MeDICINE* is highly customizable. It offers plugin interfaces for most of its components, like API endpoints, container resource limitation models, or topology generators.

Fig. 2b shows the main components of our system and how they interact with each other. The *emulator core* component implements the emulation environment, e.g., the emulated PoPs. It is the core of the system and interacts with the *topology API* to load topology definitions. The flexible *endpoint API* allows our system to be extended with different interfaces that can be used by MANO systems to manage and orchestrate emulated services (Sec. III-E). The *resource management API* allows to connect resource limitation models that define how much resources, like CPU time and memory, are available in each PoP (Sec. III-G). Finally, we provide an easy-to-use CLI that allows developers to interact with our platform.

D. Topology Definition

To test network services in realistic multi-PoP scenarios, test topologies define available PoPs, their resources, as well as the network and its properties between them. In contrast to classical Mininet topologies, our *MeDICINE* topologies do not describe single network hosts connected to the emulated network but available PoPs. In the most simplified case, such a PoP emulates just a single node, i.e., a router with attached compute and storage facilities like a Blade server. A more sophisticated node represents a small data center, which comprises several servers and is internally connected by a single SDN switch. More abstractly, an emulated PoP can also be a complex data center whose internal connection is simplified into a big-switch abstraction (as shown in Fig. 2a). For all these versions, we assume that the MANO system has full control over whether a particular VNF is executed at a particular PoP but does not care about internals of the PoPs.

A *MeDICINE* topology allows to add an arbitrary number of SDN switches between PoPs (Fig. 2a, *s1* and *s2*). These SDN switches as well as any SDN switches within each PoP can be either controlled by standard SDN controllers, by custom controller implementations provided by the network service developer, or by the MANO system itself. Thus, complex

network and forwarding setups with a high number of inter-PoP switches can be emulated.

We based our topology API on Mininet's Python API which has the benefit that developers can use scripts to define or algorithmically generate topologies. Listing 1 shows an example topology script defining two PoPs that are interconnected by a single switch. It shows how the PoPs are connected and how the link setup is done (lines 1–8).

E. Flexible Endpoint API

After an emulation topology is defined, MANO systems need a way to start and stop compute instances within the emulated PoPs. To do so, we introduce the concept of *flexible API endpoints* (Fig. 2b). Such an API endpoint is an interface to a PoP that provides typical infrastructure-as-a-service (IaaS) semantics to manage compute instances. Instead of fixing our design to a single interface implementation, we provide an abstract API and allow users of the platform to implement their own endpoints on top of it. This has the benefit that our platform can be integrated with any MANO system as long as an API endpoint that provides the expected interfaces is created. Examples for such endpoints are OpenStack Nova or Heat-like interfaces, OpenVIM-like interfaces, or any other open or proprietary interface to which a MANO system should be connected.

These API endpoints are assigned to PoPs in the topology scripts (Listing 1 lines 15–20). The default approach is adding one endpoint to each PoP so that each emulated PoP provides its own management endpoint (Fig. 2b). From the perspective of the MANO system, this looks exactly like a real multi-PoP environment offering a heterogenous set of management interfaces towards the available PoPs.

F. Networking and Chain Management

To emulate a fully working network service, we need to deploy its VNFs and set up the forwarding path between them, as shown in the service chain of Fig. 2a. Since the emulated PoPs consist of SDN switches, a service developer can have full control over the forwarding paths of the service's network traffic. Setting up a chain where traffic is steered along a defined path is now a matter of setting the correct forwarding entries in the SDN switches with an SDN controller. To support developers, we provide a simplified API that brings Service Function Chaining (SFC) functionality into *MeDICINE* while hiding the complexity of low-level SDN protocols. This API allows to chain running containers by calling a single method, i.e., `setChain(vnf1, ... , vnfN)`. Our solution uses VLAN tags as identifier to differentiate chains in multiple emulated services, similar to ongoing research regarding the use of Network Service Headers (NSH) [15]. An internal graph representation of the topology and its attached containers is kept to compute the forwarding chain with the fewest hops or the smallest delay.

G. Resource Models

Even though cloud systems provide virtually infinite compute resources to their customers, realistic scenarios, especially with small PoPs, look different. Such PoPs offer limited compute, memory, and storage resources which have to be considered by a MANO system when placement and scaling decisions are taken. To emulate such resource limitations, *MeDICINE* offers the concept of *flexible resource models* assigned to each PoP (Listing 1 lines 10–13). These models are called whenever containers are allocated or released and they compute CPU time, memory, and storage limits for each of them. These models are also able to reject allocation requests, indicating that there are no free resources left on a given PoP. A generic API allows developers to easily create their own resource models. For example, a telco operator that deploys services in its own PoPs has more control about available resources than a web service provider that buys cloud resources from a third party, like Amazon. However, *MeDICINE* can be used as a prototyping platform in both cases. The telco operator might, e.g., use a resource model in which resources are strictly reserved whereas the web service provider uses a model in which the service's performance is influenced by other random 3rd party services. Models for other operational metrics, like prizing models, can also be implemented, e.g., increase prizes for resources if a PoP is highly utilized.

To showcase how a *MeDICINE* resource model looks like, we provide two example CPU limitation models that are implemented in our prototype; memory and storage models that use the same ideas are available as well. The goal of the presented models is to limit the overall available CPU capacity of each PoP in a way such that the utilization of one PoP does *not influence* other PoPs. The presented models use the notion of *compute units* (CU) to describe the relative amount of CPU time allocated to a single container. E.g., a container that requests 4 CUs will get twice as much CPU time

```python
# create two PoPs
p1 = net.addPoP("My PoP 1")
p2 = net.addPoP("My PoP 2")
# create an intermediate SDN switch
s1 = net.addSwitch("s1")
# connect PoPs: p1 <-> s1 <-> p2
net.addLink(p1, s1, delay="10ms")
net.addLink(p2, s1, delay="50ms", loss=2)
# init. and assign resource models for each PoP
r1 = ResModelA(max_cu=24, max_mu=80, max_su=90)
r1.assignPoP(p1)
r2 = ResModelB(max_cu=80, max_mu=120, max_su=280)
r2.assignPoP(p2)
# instantiate and start cloud interface for each PoP
api1 = HeatCloudApiEndpoint(port=8004)
api1.connectPoP(p1)
api1.start()
api2 = HeatCloudApiEndpoint(port=8005)
api2.connectPoP(p2)
api2.start()
# run the emulation
net.start()
```

Listing 1: Example *MeDICINE* topology with two PoPs connected to Heat-like cloud endpoints and example resource models.

as a container requesting 2 CUs. Using this, relative resource requirements can be described independently from absolute available resources. Further, we define:

- $E_{\text{cpu}} \in (0, 1]$ percentage of physical CPU time available for emulation. For example, all containers together will not use more than 60% of the physical CPU if $E_{\text{cpu}} = 0.6$.
- $N \in \mathbb{N}_{>0}$ number of PoPs in the emulated topology.
- $\text{mc}_p \in \mathbb{N}_{>0}$: number of *CUs* available in PoP p.
- $\text{ac}_p \in \mathbb{N}$: number of *CUs* allocated in PoP p.
- $\text{nc}_c \in \mathbb{N}_{>0}$: number of *CUs* requested for a container c.
- $P_c \in [0, 1]$ percentage of physical CPU time assigned to container c.

Based on this, we define a CPU limitation function as $f : \text{nc}_c \times p \rightarrow P_c$ and use it to introduce the following example resource models.

1) Model A (Fixed Limit): Our first model assigns a fixed amount of available CUs to each PoP in the system. If not enough CUs are left in a PoP when a new container should be started, the instantiation request is rejected. As a result a PoP can never be over-utilized. Eq. 1 shows this model and how it computes the CPU time (P_c) assigned to a container that requests nc_c CUs in PoP p.

$$f_p(\text{nc}_c) = \begin{cases} \frac{E_{cpu}}{\sum_{i=1}^{N} \text{mc}_i} \cdot \text{nc}_c, & \text{if } \text{ac}_p + \text{nc}_c \leq \text{mc}_p \\ 0 \text{ (reject)}, & \text{else} \end{cases} \quad (1)$$

2) Model B (Cloud-like Over-Provisioning): Our second model does not have fixed CU limits per PoP. Instead, it allows CPU over-provisioning, which is a typical concept in IaaS clouds. For example, OpenStack Nova sets its default *cpu_allocation_ratio* property to 16 : 1 [16]. This results in situations in which allocated compute instances get less resources than initially requested. This means that a VNF might slow down if another VNF is started in the same PoP.

To model this behavior, we scale the limits of all containers within the same PoP by an over-provisioning factor defined as the fraction of available and currently used CUs in a PoP (Eq. 2). Further, we update the limits of all containers of PoP p whenever a new container is added or removed from p. As a result, the available CPU time for each container in an over-utilized PoP p is reduced if more CUs than available are used ($\text{ac}_p > \text{mc}_p$). However, the limits of running containers in other PoPs are not changed and thus our model creates a realistic environment in which separated PoPs do not influence each other.

$$f_p(\text{nc}_c) = \frac{E_{cpu}}{\sum_{i=1}^{N} \text{mc}_i} \cdot \underbrace{\frac{\text{mc}_p}{\max\{\text{mc}_p; \text{ac}_p\}}}_{\text{over-prov. factor}} \cdot \text{nc}_c \quad (2)$$

This provides a playground for realistic placement tests, e.g., an overloaded PoP might motivate a MANO system to reassign its containers to other PoPs with better performance. As a result, the previously over-utilized PoP is relieved and the performance of containers it is hosting improves.

Implementation-wise, our system does not use Docker's default CPU share limitation API since it is not sufficient for our use case. The first reason for this is that it only limits the CPU share if two or more containers want to utilize the entire CPU at the same time. It does not limit the CPU time if only one container is utilized and the competing ones are idle. Instead, we use the CPU bandwidth control functionalities of Linux's *completely fair scheduler* (CFS) [17]. The second reason for our custom implementation is that Docker's default API does only allow to set limitations when a container is started but not to update limits at runtime. We bypass these shortcomings by directly manipulating the *cgroup* system properties.

IV. EVALUATION

We did a couple of experiments to prove the overall concept, validate the behavior of the introduced resource models, and to showcase the system. The experiments use topologies with one and two PoPs in which Docker containers that run a workload generator (*stress*) are started. Thus, every container always tries to fully utilize the CPU. The overall available CPU time for containers is set to $E_{\text{cpu}} = 0.5$ and the experiments are executed on a machine with Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz and 16GB memory.

The goal of our first experiment is to validate that the measured CPU usage of containers in a single PoP is aligned to the theoretical CPU usage computed by our models. During the experiment a new container (requesting 1 CU) is allocated every 20 s during the experiment until 8 containers are requested. After additional 20 s, these containers are terminated one by one. The maximum limit of available CUs in the PoP is set to 4 CUs so that some of the requests are rejected (model A) or the PoP is over-utilized (model B). Fig. 3 shows the aggregated CPU usage for the entire PoP as well as the average CPU usage for a single container, both measured with Docker's status API that returns detailed CPU time statistics. Additionally, the expected CPU utilization calculated by our models is plotted. The results show that the measured CPU utilization for all containers in the PoP is close to the limits computed by the model. The bottom graph in Fig. 3a shows how model A rejects requests after 4 containers are allocated. Model B, in contrast, accepts all 8 containers since it allows over-provisioning (Fig. 3b). It also shows how the available avg. CPU time for containers is reduced when the PoP is over-utilized. An interesting observation in Fig. 3b are the spikes in the measured CPU usage. We found that they happen during the reconfiguration of CPU limits of already running containers which is needed by model B and happens whenever a container is added or removed from the system. Investigating this issue and quantifying its effects will be part of future work.

The second experiment shows how the presented resource models provide resource isolation between PoPs. It emulates a topology with two PoPs, each with a limit of 2 CUs. During the experiment, the avg. physical CPU time available for a single container is measured for different numbers of *stress* containers running in PoP1. The number of *stress* containers

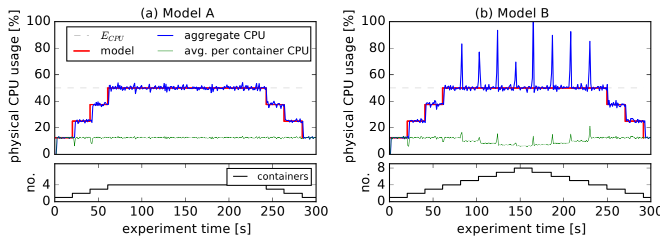


Fig. 3: Modeled vs. measured container CPU usage

running in PoP2 is fixed to two containers. With this, we can observe how the changing number of containers in PoP1 influences the performance of containers in PoP2. Fig. 4 shows the results for different resource models. Fig. 4a shows what happens when no resource limitation model is used. All containers compete for the entire physical CPU time and the performance of containers in PoP2 is reduced when more containers are added to PoP1. The same happens in Fig. 4b with the difference that it uses a common resource model for both PoPs. Thus, the containers do not influence each other until the maximum of two containers is running in PoP1. Fig. 4c shows the behavior of model A which does not allow over-provisioning and rejects all requests when two containers are already running in a PoP. The behavior of model B is shown in Fig. 4d. The figure validates that the model enables resource isolation between PoPs even when PoP1 is over-utilized and the CPU time for each of its containers is reduced. Fig. 4d shows that there are still some minimal influences to the performance of PoP2 when containers are added to PoP1, i.e., PoP2's performance is reduced by around 2% when PoP1 is over-utilized by a factor of $16\times$.

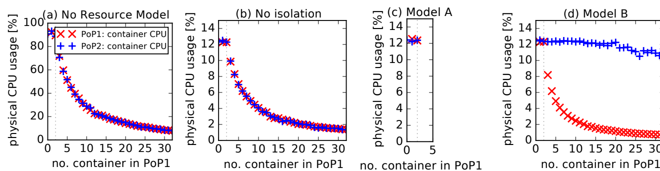


Fig. 4: Cross-PoP resource isolation using different resource models

V. Conclusion and Future Work

We introduced *MeDICINE*, a novel prototyping platform for NFV that goes beyond its initial NFV use cases and is an excellent prototyping and test platform for distributed cloud services. The results of our experiments show that *MeDICINE* can simulate resource limitations in multi-PoP environments while ensuring resource isolation between PoPs. By using standard Docker containers to execute network functions within our emulation platform, *MeDICINE* allows developers to directly move their tested services into production environments without additional changes.

We believe that *MeDICINE* is an important step towards a fully integrated development support toolchain for network service development. Its code is open-source and available as part of the emulation platform of the 5G-PPP project SONATA [18]. We will continue our work in several directions, e.g., demonstrating the integration with different MANO systems, network service and VNF test automation, improved resource models with performance prediction, and advanced service chaining techniques.

Acknowledgments

This work has been partially supported by the SONATA project, funded by the European Commission under Grant number 671517 through the Horizon 2020 and 5G-PPP programs (www.sonata-nfv.eu) and the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

References

[1] H. Karl, S. Dräxler, M. Peuster, A. Galis, M. Bredel, A. Ramos, J. Martrat, M. S. Siddiqui, S. van Rossem, W. Tavernier *et al.*, "DevOps for network function virtualisation: an architectural approach," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 9, pp. 1206–1215, 2016.

[2] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso, "Multi-domain service orchestration over networks and clouds: A unified approach," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. ACM, 2015, pp. 377–378.

[3] ETSI, "GS NFV 002: Network Functions Virtualisation (NFV): Architectural Framework," 2014.

[4] I. Pelle, T. Lévai, F. Németh, and A. Gulyás, "One tool to rule them all: A modular troubleshooting framework for sdn (and other) networks," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. ACM, 2015, pp. 24:1–24:7.

[5] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[6] W. Zhao, Y. Peng, F. Xie, and Z. Dai, "Modeling and simulation of cloud computing: A review," in *Cloud Computing Congress (APCloudCC), 2012 IEEE Asia Pacific*, Nov 2012, pp. 20–24.

[7] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, 2008.

[8] L. Mamatas, S. Clayman, and A. Galis, "A service-aware virtualized software-defined infrastructure," *Communications Magazine, IEEE*, vol. 53, no. 4, pp. 166–174, 2015.

[9] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010.

[10] P. Wette, M. Drxler, and A. Schwabe, "Maxinet: Distributed emulation of software-defined networks," in *Networking Conference, 2014 IFIP*, 2014, pp. 1–9.

[11] J. Ahrenholz, "Comparison of core network emulation platforms," in *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, Oct 2010, pp. 166–171.

[12] "OpenStack DevStack," http://docs.openstack.org/developer/devstack/, 2016.

[13] M. Keller, C. Robbert, and M. Peuster, "An evaluation testbed for adaptive, topology-aware deployment of elastic applications," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. ACM, 2013, pp. 469–470.

[14] M. Peuster, "Containernet," https://github.com/mpeuster/containernet.

[15] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," https://tools.ietf.org/html/draft-ietf-sfc-nsh-04, Tech. Rep., 2015.

[16] "OpenStack Nova Guide," http://docs.openstack.org/openstack-ops/content/scaling.html, 2016.

[17] P. Turner, B. B. Rao, and N. Rao, "CPU bandwidth control for CFS," in *Linux Symposium*, vol. 10. Citeseer, 2010, pp. 245–254.

[18] SONATA Consortium, "SONATA Emulator," https://github.com/sonata-nfv/son-emu.