# DReAM - A Distributed Result-Aware Monitor for Network Functions Virtualization

Ricardo J Pfitscher*, Eder J Scheid*, Ricardo L dos Santos*
Rafael R Obelheiro†, Mauricio A Pillon†, Alberto E Schaeffer-Filho*, Lisandro Z Granville*
*Institute of Informatics – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
{rjpfitscher, ejscheid, rlsantos, alberto, granville}@inf.ufrgs.br
† Department of Computer Science - Santa Catarina State University - Joinville, Brazil
{rafael.obelheiro, mauricio.pillon}@udesc.br

***Abstract*—Network Functions Virtualization (NFV) is a key technology to reduce management costs as well as to improve scalability and elasticity of computer networks. Still, recent research efforts have been exposing additional management challenges. Concerning monitoring in particular, new types of entities and requirements are underexploited. To address these issues, we propose DReAM, a resource management architecture based on management by delegation and distributed monitoring, where each agent runs a diagnostic model to compute the network service state. In this paper, we describe DReAM's proposed architecture and its major components. We also discuss the feasibility of DReAM through experimental and analytical evaluations, where we observed application throughput, CPU utilization, communication overhead, scalability, and diagnosis complexity. We provide a trade-off analysis on the monitoring strategies in NFV scenarios. Our results indicate that a result-aware strategy is a better option when the monitored environment has more than 256 agents or when the diagnosis module induces at least 10% of CPU utilization.**

***Index Terms*—NFV monitoring, result-aware monitoring, distributed diagnostic.**

## I. Introduction

Network Functions Virtualization (NFV) takes advantage of virtualization technologies to decouple traditional Internet middleboxes functions, such as network address translation and load balancing, from dedicated hardware [1] [2]. As a result, network functions are hosted by and share physical resources of commercial of-the-shelf (COTS) servers. In NFV, the costs of deploying and maintaining networking infrastructures can be reduced [3]. In addition, by managing virtualized network functions (VNF), operators can resize and migrate networking resources on demand. It has been claimed [2] [4] that NFV, alongside Software-Defined Networking (SDN), form the basis for future networking architectures.

In NFV scenarios, the proper operation of VNFs has a direct influence on the user perception of application performance. Therefore, the management aspects of NFV need to be investigated. One of such management aspects is NFV monitoring. NFV shares some features that promoted cloud computing, but today these features are being employed in the context of networking functions as well. Although cloud monitoring solutions can be employed in NFV, additional entities introduced by NFV call for the development of new monitoring solutions tailored to NFV [5]. Such entities, with specific monitoring needs, include: Virtual Network Functions (VNFs) running inside virtual machines; Physical Network Functions (PNFs) running inside conventional network middleboxes; Network Function Virtualization Infrastructure (NFVI), including hosts, hypervisors, and commodity hardware; and Network Service (NS), which is a group of Network Functions that provides a particular service.

In addition to entity-specific monitoring, the characteristics of NFV introduce two additional monitoring requirements: (*R1*) VNFs can be chained with other VNFs and/or PNFs to form a NS [5], and thus a functionality to maintain and monitor the global NS state is desired; and (*R2*) because of the nature of their roles, network functions require high performance packet processing (up to 10 Gbps) [6] [7], and thus these network functions must be monitored in near real-time [1].

To comply with the aforementioned NFV monitoring requirements, we introduce DReAM (Distributed Result-Aware Monitor), an NFV monitoring architecture in which agents are responsible for monitoring NFV entities and for providing a diagnosis about the state of each entity. This diagnosis, hereafter called result, is consumed by NFV orchestrators to perform management decisions. Moreover, we employ a distributed strategy in which every agent inside an NS monitors its neighbor VNFs; we call it neighborhood monitoring. We also describe the critical elements of the DReAM architecture and provide experimental and analytical evaluations.

Our main contributions are: (*i*), an architecture that computes the global view of the state of network services (requirement *R1*) by means of distributed monitoring, where each monitor retrieves information about their neighbors' VNFs; (*ii*), the use of management by delegation (MbD) to enable monitors to quickly diagnose the state of each entity in a NFV infrastructure, which meets the need for near real-time state definition (requirement *R2*); and (*iii*), a trade-off analysis of the proposed monitoring strategy in terms of the number of NFV entities and the CPU utilization.

The remainder of this paper is organized as follows. In Section II, we review MbD and distributed monitoring. We describe DReAM and the expected behavior of NFV orchestrators in Section III. Performance evaluation and discussion are presented in Section IV. In Section V, we review related work. Finally, we present conclusions and future work in Section VI.

978-1-5090-0679-3/16/$31.00 ©2016 IEEE

## II. DReAM Monitoring Strategy

In this paper, we address two main requirements of NFV monitoring: (*R1*) maintenance of a global state of the network services in a scalable way, and (*R2*) minimal delay in the notification of state changes with meaningful monitoring reports. R1 determines that the management system needs to maintain the information about network services, composed of multiple VNFs and PNFs, without overwhelming the network infrastructure. R2 determines that the monitoring solution must report the state changes of NFV entities as soon as they occur, enabling fast decision-making at the NFV orchestrator. Thus, DReAM relies on two fundamental network management techniques: *management by delegation* and *distributed monitoring*.

### A. MbD-based monitoring

In management by delegation (MbD) [8], top-level managers (TLM) delegate to mid-level managers (MLM) the responsibility of executing management tasks. In comparison to traditional centralized management, MbD improves scalability by employing a number of MLMs. In DReAM, MbD allows the replacement of a single orchestrator by a set of monitors that detect specific conditions of interest.

In a naïve solution, all entities would generate raw monitoring information that would be forwarded to a management system for post processing. Afterwards, and typically offline, these raw data would be analyzed by a human or system to identify issues and make decisions (*e.g.*, capacity adjustment). This strategy, based on post processing, delays the decision-making process. To tackle this problem, we propose a *result-aware monitoring*. Besides being a special case of MbD, result-aware monitoring is based on the concept of continuous distributed monitoring of state [9] [10].

In result-aware monitoring, each monitoring agent attaches a model to produce a diagnosis about an observed information. This diagnosis exposes the behavior of the monitored entity, which we call *result*. If the provided result is classified as an *issue* (*e.g.,* overloaded resource), such result is forwarded to the NFV orchestrator or management system (or both), to act upon it. This strategy reduces monitoring traffic and enables near real-time decision-making.

### B. Distributed monitoring strategy

To achieve scalable monitoring, DReAM employs a distributed monitoring strategy. In NFV environments, network functions (NFs) run either inside Virtual Machines (VMs) or inside physical machines. In our architecture, every VM and physical machine hosts a monitoring agent. To keep track of global state information, each agent monitors both its own resources (local monitoring) and its neighbors (other VNFs or PNFs that compose the NS).

When an agent monitors its own resources (*i.e.*, local monitoring), it collects metrics and requests to an internal diagnostic module to produce a state report (monitoring result) about the monitored VNF. On the other hand, when monitoring its neighbors, the agent periodically collects the state of each neighboring VNF; as such, if an issue is detected (*e.g.*, unreachable neighbor) an orchestrator is notified. This distributed strategy helps to achieve near real-time monitoring, since it immediately exposes state changes and advertises the overall state of the network service.

To illustrate our expected monitoring agents placement, consider the scenario depicted in Figure 1. Each agent is responsible for monitoring VNF resources and the neighborhood that composes the NS. The continuous arrows represent the network flows, which is determined by forwarding graphs. Dotted arrows are related to neighbor monitoring, in which agents request the state of their neighbors. Agents can run either inside a VM or directly in a host; this allows maintaining the state about hosts and VNFs. Supposing that the VNFs set $\{1, 2, 3\}$ composes a network service from ISP1 to ISP2, and that the respective DReAM agents are monitoring this NS. We consider that each agent performs local monitoring, and also requests the VNF state of other agent. With this neighborhood monitoring, we fulfill the requirement of monitoring the set of entities that composes an NS (R1).

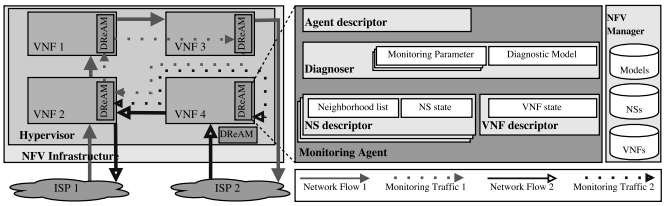



Figure 1. DReAM monitoring scenario and agents architecture. The left side of figure exposes an NFV environment with two network services. The right side depicts the DReAM monitoring agent architecture.

To determine the monitored neighbor VNF, the monitoring agent computes a hash function. The orchestrator defines a list of forwarding graphs of which an agent is a part. Then, consider that each graph has a list of identified nodes, where an $id$ represents the node's order in the graph. Thus, we define a function $g(id)$ that returns the monitored neighbor as:

$$g(id) = id + mod(t, N) + 1 \tag{1}$$

where $t$ is the time in minutes since midnight (*e.g.,* at 1 a.m. $t = 60$), and $N$ is the number of VNFs in the graph. The reason for expressing the time in minutes is not to be affected by clock skew on the order of a few seconds. As one can notice, the function $g(id)$ returns values between $id + 1$ and $id + N$. Thus, we define $f(id)$ as a function of $g(id)$:

$$f(id) = \begin{cases} g(id) & \textbf{if} \quad g(id) \leq N \\ g(id) - N & \textbf{if} \quad g(id) > N \end{cases} \tag{2}$$

if $f(id)$ returns a value equal to $id$ we simply add one to it.

The proposed hash function has two key features. First, it covers the entire NS, which means that, at a particular moment, all the VNFs monitor each other. Second, given the time dependency in $mod(t, N)$, the function provide a dynamic behavior, updating the monitoring structure at each instant $t$. Thus, even if an agent fails, all the VNFs' states will be known.

## III. DReAM Components

We present an abstract view of DReAM components, especially regarding the diagnoser and the monitored entities. Therefore, developers can modify these modules according to their needs. The right side of Figure 1 presents the architecture of DReAM. For the lack of standardization, we decided to use the nomenclature of ETSI specifications [5]. As our focus is on NFV monitoring, we represent the orchestrator/manager as modules from another system.

The monitoring module is responsible for providing an interface to other agents, collecting neighborhood state, and updating the state of NFV services and VNFs. Thus, each monitoring agent can be aware of the state of others. This entity has two main operations: *get_state()*, used by remote agents to request another agent's current state; and *update_diagnoser()*, used by the orchestrator to update a diagnostic model (*e.g.*, adjust thresholds, change monitored metrics, and add expected running processes in the operating system) on a set of VNFs. Each component of this module is explained as follows.

*1) Agent descriptor:* The agent descriptor component describes the monitor regarding version, security, compatibility and status. The stored data depends on developers implementation, which includes, but is not limited to agent version, access control credentials, permissions and supported metrics.

*2) Diagnoser:* This component defines the state of an element in the monitoring agent. For this, the diagnoser implements a diagnostic model. This model could evaluate if a monitoring parameter complies with a service deployment flavor [5] requirement. A flavor determines an assurance parameter (*e.g.*, number of calls per second (cps)). The monitoring parameter component is used to store metrics about the monitored resources. A resource can be an application, an operating system, or a physical device. For diagnosis purposes, each metric and the related measurements are used by the diagnoser to define the monitoring parameter's state. These metrics can be at different levels: application-level (*e.g.*, requests per second), OS-level (*e.g.*, number of processes), or at the level of physical resources (*e.g.*, free memory).

A set of diagnosers can be implemented in a single monitor, where each diagnostic model can be used for a different context (*i.e.*, applications, services, or resources). It can also provide diagnostic models for resource provisioning (*e.g.*, diagnostic model for memory provisioning [11]), or a model to identify or prevent SLA violations (*e.g.*, a model that checks application requirements [12]). More complex diagnosis include *(i)* deadlock detection, where the diagnoser will compute a resource allocation graph to find cycles and, *(ii)* bottleneck discovering, where a multi-agent historical data must be evaluated. After the diagnosis phase, the diagnosers produce a state for each monitored element: the NS and the VNF, stored in their respective descriptors.

*3) NS descriptor:* ETSI [5] defines a Network Service Descriptor (`nsd`) to store the information about a network service. In addition to the attributes defined by ETSI, each agent in DReAM stores its perception about the network service and a neighborhood list. The NS'es state can be defined by the monitoring agent in two ways: retrieving the neighbor's state or by applying the diagnostic model over the set of monitoring parameters.

*4) VNF descriptor:* As in the `nsd` definition, the ETSI defines a Virtual Network Function Descriptor (`vnfd`) to store information about VNFs. For this element, we improve the ETSI's specification by adding the state information. The monitoring agent refreshes this state by locally monitoring and requesting the diagnoser to compute the VNF's state.

The orchestrator and the manager will lead with global management decisions. The former deals with resource allocation and VNF placement, the latter is in charge for the lifecycle management of VNF instances and overall coordination [5]. In the DReAM architecture, the NFV manager stores the diagnostic models and the information about the NSes. Then, it uses this information to configure the neighbor list and the diagnoser in each agent. It also receives the states from agents and sends reports to the orchestrator. In our prototype implementation, the manager also solves state changing occurrences by scaling-up/down the provisioned resources.

## IV. Evaluation

To attest DReAM's functionalities, we performed both experimental (Sec. IV-A) and analytic evaluations (Sec. IV-B).

### A. Experimental evaluation

We implemented a prototype using a diagnosis model for CPU provisioning (Table I). This model periodically samples two OS-level CPU metrics (%Idle and %Steal). Steal time represents the fraction of time that a VM wants to run (*i.e.*, has ready processes) but its virtual CPU is not scheduled on a physical CPU due to contention. Idle represents the percent of time that the processor was not used. Average values obtained over five samples are compared to thresholds and the resource is classified in three levels: over-provisioned, under-provisioned, or correctly provisioned.

The intuition behind the model is that a VNF will be correctly provisioned when its CPU is moderately used, without unduly wasting resources. Thus the under-provisioned status occurs in two cases: when average CPU utilization is above 80%, or when average steal is above 5%. The CPU is correctly provisioned when utilization is between 50% and 80%. When average CPU utilization is below 50%, we analyze the number of samples where the CPU was fully utilized (above 95%); if this number is 20% or more, the VNF is correctly provisioned (since the load is fluctuating a lot, we cannot scale down the CPU), otherwise it is over-provisioned.

Table I
DIAGNOSIS MODEL FOR DEFINING THE VNF STATE

| Steal | CPU | Fully usage | Diagnosis |
|---|---|---|---|
| $\geq 5\%$ | - | - | under-provisioned |
| $<5\%$ | $\geq 80\%$ | - | under-provisioned |
| $<5\%$ | $< 80\%$ and $\geq 50\%$ | - | correctly |
| $<5\%$ | $< 50\%$ | $\geq 20\%$ | correctly |
| $<5\%$ | $< 50\%$ | $< 20\%$ | over-provisioned |

*1) Test environment:* Our test environment comprises three machines: (*A*) Dell PowerEdge R620 with two Intel(R) Xeon(R) CPU E5-2630 v2 at 2.60GHz processors and 64 GiB RAM; (*B*) Dell Optiplex 9020 with an Intel(R) Core(TM) i7-4770S CPU at 3.10GHz and 8 GiB RAM; and (*C*) Raspberry Pi 2 with a quad core ARM processor at 700 MHz with 1 GiB RAM. The first machine, hosting the hypervisor, runs a Debian 7 64-bit Linux with Xen 4.1; the second, the orchestrator, runs a 3.13.0 Ubuntu Linux; the third runs a Debian 3.18 Linux. All machines are connected to a 100 Mbps Ethernet LAN. Even if we did not adhere to the 10 Gbps VNF throughput requirement, we argue that the network capacity of our environment does not tamper the diagnosis process. We used virtual machines running 3.2.0-4-amd64 Linux with a dedicated VCPU and 1 GiB RAM. They are connected by the hypervisor's virtual link at 1 Gbps.

We evaluated DReAM in a typical NFV use case [5], which is composed of three network functions: a firewall, a traffic shaper, and a deep packet inspector (DPI); and by two services: a TCP service and a UDP service. Figure 2 presents the disposition of services and functions: the firewall blocks all UDP datagrams, the traffic shaper limits the traffic to 1 Mbps, and the DPI collects network statistics. TCP traffic is generated using RUBiS [13], and UDP traffic using Iperf.

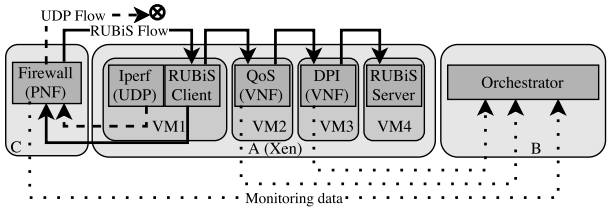



Figure 2. Evaluation scenario. The continuous lines represent the RUBiS flows. The dashed lines represent the UDP traffic and the dotted lines represents the monitoring traffic data.

*2) Methodology:* In the experimental evaluation, we measured four metrics: (*i*) RUBiS throughput in requests per second; (*ii*) the delay of detecting a state change, which is computed as the difference between the event occurrence and the orchestrator detection; (*iii*) the CPU utilization in the manager/orchestrator; and (*iv*), the number of exchanged bytes during all workload run (60 seconds individually). For each test, we performed 30 runs. The standard errors are related to a confidence level of 95%. In our outlier treatment, we excluded the values with more than two standard deviations, which resulted in 1% of data rejection.

To capture the network traffic, we used `Tshark`, which is a network protocol analyzer. In our case, we filtered the data by the IP address of our monitored agents. We used the `ps` tool from Linux, which displays information about a specific process, to measure CPU utilization with a granularity of one second. It should be noted that the value obtained refers to a fraction of the overall machine capacity. For example, in an eight processor machine, if `ps` reports CPU utilization equal to 10%, this will be equivalent to 80% of one processor.

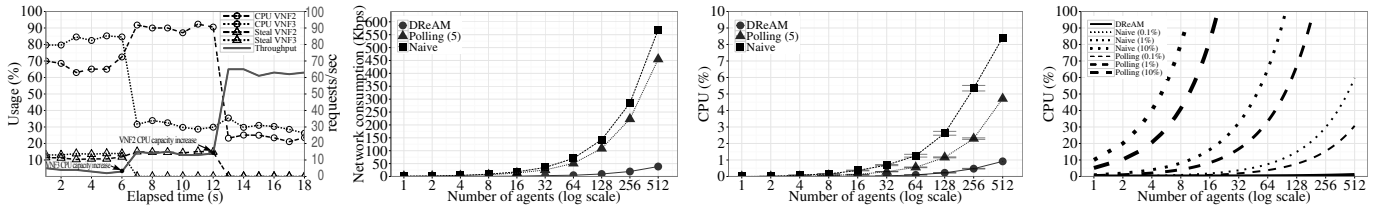
*3) Results:* Our first experiment verifies if result-aware monitoring can be used to adjust the resource capacities of the VNFs in an NS. VM1 and VM2, which host the QoS and DPI VNFs, had their CPUs initially capped at 10%. We then started RUBiS, Iperf, and the monitoring agents. Figure 3(a) shows CPU utilization and RUBiS throughput during 18 seconds of a single run. In this test, the manager waits 5 sec after receiving a state change message to adjust resource capacities.

When the experiment starts, both monitoring agents detect underprovisioned situations: VNF2 has 70% CPU utilization and 11% of steal time, while VNF3 has 84% CPU and 15% steal. Thus, at $t = 6$ s the manager acts and sets the CPU cap for VNF3 to 100%; this results in a increase in throughput from 5 to 15 reqs/s. At $t = 7$ s VNF2 is still underprovisioned, with 90% CPU and 15% steal. So, at $t = 12$ s its CPU cap is also set to 100%, which lowers the CPU usage and bumps the throughput to 63 reqs/s. From $t = 18$ s onwards, both VNFs are in the overprovisioned state, and the manager could adjust the resources according to its policy. The throughput behavior shown in Figure 3(a) confirms that our diagnosis model is able to detect the resource provisioning states of VNFs in an NS.

In the second test, we access the DReAM architecture with respect to the near real-time state changing detection requirement. Since related work on NFV often relies on traditional monitoring strategies [14] [15] [16] [17], or do not detail their monitoring solution [7] [18], we decided to compare DReAM with two common strategies: a naïve and a polling one. In the former, agents forward data to the orchestrator as soon as they are measured. In the latter, agents summarize the measured data in a set of five values (monitoring period) and send it to the orchestrator. In both, the orchestrator is responsible for computing the diagnosis. The experiment consists in emulating a workload in the clients, and, at random periods of time, forcing a state change in an arbitrary VNF (with a synthetic program), capturing the event's timestamp. Where, we found that, in terms of delay, naïve and DReAM are statistically equivalent, and the polling depends on the monitoring period.

We also measured the CPU utilization in the server. From the results, we observed that the CPU utilization in the manager was imperceptible (*i.e.*, less than 0.01% in all cases). Thus, we conclude that our diagnostic model is simple and does not stress the server processor. Given this reason, there is no significant difference among the strategies in terms of delay in processing the VNF states. We also designed a scenario for evaluating the scalability of each strategy, we measured two metrics: the CPU utilization and the number of bytes transmitted in the network. Also, we observe the relationship among these metrics and an increasing number of agents. We considered the worst case scenario for DReAM, in which all agents detect an event of state change simultaneously.

The scalability results are plotted in Figures 3(c) and 3(b). DReAM has lower network overhead than the other strategies. Moreover, increasing the number of agents had a significant impact in naïve and polling strategies, whereas DReAM scaled better. Despite using this simple diagnosis, all strategies support more than 512 agents without exceeding




(a) Resource provisioning (b) Network consumption (c) CPU utilization (d) CPU utilization prediction

Figure 3. Evaluation results. (a) resource provisioning according to DReAM's monitoring agents responses. (b) network consumption evolution according to the number of monitoring agents. (c) CPU utilization according to the number of agents in each monitoring strategy. (d) CPU utilization prediction according to the number of agents and the CPU utilization for one agent diagnosis (the thicker is line the bigger is diagnosis usage).

more than 10% of CPU utilization, which is negligible in terms of performance degradation. In those figures, the x axis is in a logarithmic scale to provide better visualization with different numbers of agents; it should be noted, however, that the relationship between the variables is actually linear (*e.g.*, double the number of agents will roughly double network traffic). Thus, we applied a linear regression model to extrapolate this behavior and determine when the DReAM strategy can provide a representative benefit for an NFV administrator. In other words, we wanted to find out at what load of diagnosis DReAM would be the only solution able to meet the near real-time requirement for a certain NFV environment.

### B. Analytical evaluation

In our analytical analysis, we extrapolate the scalability behavior according to different values of diagnosis CPU utilization. For that, we first obtained, through a linear regression, the functions that provide the manager's CPU utilization in relation to the number of monitoring agents:

$$f_{naive}(x) = 0.0178 \times x, R^2 = 0.98 \quad (3)$$

$$f_{polling(5)}(x) = 0.0091 \times x, R^2 = 0.99 \quad (4)$$

$$f_{DReAM}(x) = 0.0018 \times x, R^2 = 0.98 \quad (5)$$

where, $f_{strategy}(x)$ is the CPU utilization for a given strategy for $x$ number of agents, and the R-squared is the coefficient of determination. Next, we calculated the proportionality factor between the functions, considering the naïve strategy as the baseline. Thus, we found that:

$$f_{polling(5)}(x) = f_{naive}(x) \times 0.0091/0.0178 \quad (6)$$

$$f_{DReAM}(x) = f_{naive}(x) \times 0.0018/0.0178 \quad (7)$$

These equations, however, do not expose a relation with the initial CPU diagnosis utilization. Therefore, we investigated how this utilization is related to the slope of $x$. Consider that the multiplicative factor in Equations 3 – 5 are related to the sum of two variables, the CPU utilization for the diagnosis of one agent $d$, and an strategy overhead $o$. Thus, in the case of the naïve strategy, we have $d + o = 0.0178\%$. Looking at our results, we found that the CPU utilization for diagnosing one agent is $d = 0.00105\%$, and consequently, $o = 0.01667\%$. Thus, we can express the CPU utilization for diagnosing one agent with each strategy:

$$f_{naive}(x, d) = (d + 0.01667) \times x, R^2 = 0.94 \quad (8)$$

$$f_{polling(5)}(x, d) = 0.5143 \times f_{naive}(x, d), R^2 = 0.98 \quad (9)$$

Finally, using Equations 5, 8, and 9 we predicted the CPU utilization in the manager for the three strategies according to the utilization of one agent diagnosis. Thus, we extrapolated for the following values of $d$: 0.1%, 1%, and 10%. Figure 3(d) presents the total CPU utilization according to the number of agents and the value of $d$. Given the distributed characteristic of DReAM, the CPU utilization for the agent does not have an impact on the server side, and so, it is not affected by the value of $d$. Therefore, we only plot one DReAM curve.

Results show that naïve and polling strategies are directly affected by the consumption of one single agent. For example, if the diagnosis consumes 1% of CPU, the naïve strategy saturates the orchestrator with more than 64 agents, and possibly affects the delay for a state change detection. Also, the polling strategy saturates the orchestrator CPU with less then 256 agents. With a lower consumption, *e.g.*, less than 1%, all strategies present high scalability, supporting more than 512 agents, but DReAM has the advantage of generating a smaller network overhead. In summary, DReAM is superior to the other strategies in two situations: when diagnosis processing takes more than 10% of CPU for one agent, because the naïve and polling strategies only support a small number of agents (respectively 8 and 16); or when the NFV environment is large (more than 256 agents), because the naïve and polling strategies will delay the state change identification.

## V. Related work

Maini *et al.* [16] present an overview of the challenges related to NFV management and orchestration (MANO), and also provide an architecture for virtual resource placement. In the architecture, monitors measure applications data and deliver it to a manager, which in conjunction with an orchestrator carry out the provisioning of virtual machines. Despite producing raw data for the manager, in our work, the monitor agent delivers monitoring results to the orchestrator. DReAM reduces the computational load in the orchestrator.

Xilouris et al. [15] present T-NOVA, an approach that uses the concept of NFaaS (Network Functions as a Service), in which network operators can deploy and offer NFs on demand. In terms of management and orchestration, the authors indicate

that real-time monitoring is needed for automatic resource provisioning in NFV. Our approach differs from T-NOVA in terms of monitoring; in DReAM, monitoring agents provide results about measured data, while in T-NOVA they only send raw monitored data to a management system.

Clayman *et al*. [14] defined an architecture to orchestrate VNFs placement over NFV environments. The system is supported by a Monitoring Manager that collects and reports the resources behavior. Regarding monitoring, each virtual node has a probe to monitor the usage of network and computing resources. The probes leverage a polling strategy, each agent sends a set of attributes and values to the manager in a predefined interval. We show that the DReAM strategy is more scalable than a polling one.

De Turck and Moens propose a placement model for VNFs in [19]. The authors presented and evaluated a formal model for resource allocation of virtualized network functions within NFV environments. The main idea is to allocate a service chain, which is a set of VMs and service requests, over a network infrastructure. In that work, some low-level and high-level metrics, such as CPU utilization and requests per second, respectively, were used to define where to place service chain requests. However, the authors do not focus on the continuous measurement of these metrics after VNF placement and therefore do not support NFV monitoring. Instead, our solution covers the monitoring part and can use their model as an orchestrator for VNF provisioning.

Gember *et al*. [17] proposed Stratos, an orchestration layer for implementing middleboxes (MB) in the cloud. Their solution allows tenants to specify logical middlebox deployments and provide a complete management cycle for the chained MBs. Despite Stratos being able to address critical management aspects like scaling, placement and balancing of virtual MBs, they leverage a polling monitoring strategy, which we show is impractical to acomplish with the real-time monitoring requirement (R2). However, as the concept of virtual MBs akin to Virtual Network Functions, DReAM can be applied to the process of gathering metrics in large scale scenarios.

## VI. Conclusions

With the objective of addressing the novel management requirements related to NFV, we proposed DReAM (Distributed Result-Aware Monitor), a monitoring solution where agents are responsible for monitoring NFV entities and for providing a diagnosis of their states. DReAM is based on two central managements concepts: MbD and distributed monitoring. Although these two concepts have been extensively discussed in the literature, we have not found any solution that applies both in a unified way in an NFV context.

We evaluated a prototype of DReAM in an experimental NFV scenario. With a diagnostic model based in CPU utilization metrics, we show that by applying our monitoring approach an manager can improve the NS throughput by detecting incorrect resource provisioning. Our results also showed that, with small complexity diagnostic models, there is no significant difference in terms of the delay for detecting state changes among DReAM, polling and naive strategies. In addition, we performed an analytical evaluation to extrapolate the CPU utilization for more complex diagnosis; we concluded that the DReAM strategy is the best solution when the environment has more than 256 agents, or in complex diagnosis. As future work, we will address novel diagnostic models applicable to the NFV context.

## References


[1] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "NFV: State of the art, Challenges, and Implementation in Next Generation Mobile Networks (vEPC)," *Network, IEEE*, vol. 28, no. 6, pp. 18–26, 2014.

[2] R. Jain and S. Paul, "Network Virtualization and Software Defined Networking for Cloud Computing: a Survey," *Communications Magazine, IEEE*, vol. 51, no. 11, pp. 24–31, 2013.

[3] X. Costa-Pérez, A. Festag, H.-J. Kolbe, J. Quittek, S. Schmid, M. Stiemerling, J. Swetina, and H. Van Der Veen, "Latest Trends in Telecommunication Standards," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 2, pp. 64–71, 2013.

[4] A. Manzalini and R. Saracco, "Software Networks at the Edge: a shift of paradigm," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–6.

[5] E. G. Specification, "Network Function Virtualisation; Management and Orchestration," European Telecommunications Standards Institute, Tech. Rep. DGS/NFV-MAN001, december 2014.

[6] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the Art of Network Function Virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, 2014, pp. 459–473.

[7] J. Hwang, K. Ramakrishnan, and T. Wood, "Netvm: High Performance and Flexible Networking Using Virtualization on Commodity Platforms," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 1, pp. 34–47, 2015.

[8] G. Goldszmidt and Y. Yemini, "Distributed Management by Delegation," in *Proceedings of the 15th International Conference on Distributed Computing Systems*. IEEE, 1995, pp. 333–340.

[9] S. Meng, L. Liu, and T. Wang, "State Monitoring in Cloud Datacenters," *Knowledge and Data Engineering, IEEE Transactions on*, 2011.

[10] G. Cormode, "The Continuous Distributed Monitoring Model," *ACM SIGMOD Record*, vol. 42, no. 1, pp. 5–14, 2013.

[11] R. J. Pfitscher, M. A. Pillon, and R. R. Obelheiro, "Customer-Oriented Diagnosis of Memory Provisioning for IaaS Clouds," *ACM SIGOPS Operating Systems Review*, vol. 48, no. 1, pp. 2–10, 2014.

[12] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, and R. Buyya, "SLA-based virtual machine management for heterogeneous workloads in a cloud datacenter," *Journal of Network and Computer Applications*, 2014.

[13] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Springer-Verlag New York, Inc., 2003, pp. 242–261.

[14] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca, "The dynamic placement of virtual network functions," in *Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–9.

[15] G. Xilouris, E. Trouva, F. Lobillo, J. Soares, J. Carapinha, M. McGrath, G. Gardikis, P. Paglierani, E. Pallis, L. Zuccaro *et al.*, "T-NOVA: A marketplace for virtualized network functions," in *European Conference on Networks and Communications (EuCNC)*. IEEE, 2014, pp. 1–5.

[16] E. Maini and A. Manzalini, "Management and Orchestration of Virtualized Network Functions," in *Monitoring and Securing Virtualized Networks and Services*. Springer, 2014, pp. 52–56.

[17] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," Tech. Rep., 2013.

[18] J. Soares, M. Dias, J. Carapinha, B. Parreira, and S. Sargento, "Cloud4NFV: A platform for Virtual Network Functions," in *IEEE 3rd International Conference on Cloud Networking (CloudNet)*. IEEE, 2014, pp. 288–293.

[19] H. Moens and F. D. Turck, "Vnf-p: A Model for Efficient Placement of Virtualized Network Functions," in *10th International Conference on Network and Service Management (CNSM)*. IEEE, 2014, pp. 418–423.