NFV-Throttle: An Overload Control Framework for Network Function Virtualization

Domenico Cotroneo, Roberto Natella, Stefano Rosiello

***Abstract*—Network Function Virtualization (NFV) aims to provide high-performance network services through cloud computing and virtualization technologies. However, *network overloads* represent a major challenge. While elastic cloud computing can partially address overloads by scaling on-demand, this mechanism is not quick enough to meet the strict high-availability requirements of "carrier-grade" telecom services. Thus, in this paper we propose a novel overload control framework (*NFV-Throttle*) to protect NFV services from failures due to an excess of traffic in the short term, by filtering the incoming traffic towards VNFs to make the best use of the available capacity, and to preserve the QoS of traffic flows admitted in the network. Moreover, the framework has been designed to fit the service models of NFV, including *VNFaaS* and *NFVIaaS*. We present an extensive experimental evaluation on the NFV-oriented *Clearwater* IMS, showing that the solution is robust and able to sustain severe overload conditions with a very small performance overhead.**

***Index Terms*—Overload Control; Network Function Virtualization; NFVIaaS; VNFaaS; Cloud Computing; IP Multimedia Subsystem**

I. Introduction

NETWORK services experience an *overload condition* when they work with more traffic flows than their engineered capacity, causing resource exhaustion, and the disruption and unavailability of services. This condition leads to SLAs violations, thus potentially compromising customers' contracts. Overload and congestion control has been a key research topic in Telecom network appliances; most of proposed techniques are well assessed and are today part of networking industry standards and commercial products [1]–[3].

Telecom services are now rapidly changing to cut costs and energy consumption, to improve manageability, and to reduce time-to-market. To pursue these objectives, we are witnessing a shift of Telecom network functions from proprietary hardware appliances to software [4], adopting the *Network Function Virtualization* (NFV) paradigm [5], which leverages virtualization and cloud computing technologies.

In this new context, overload control techniques are required to evolve in order to achieve the same high-availability and performance requirements as before. In principle, virtual network functions could take advantage of cloud elasticity by scaling-out network services with on-demand resource allocation to face overload conditions. Unfortunately, cloud elasticity alone is not quick enough to meet the strict high-availability requirements of "*carrier-grade*" telecom services, which can only afford few tens of seconds of outage per month [6], [7]. As a matter of fact, scaling-out can require up to several minutes to allocate new VM replica [8], [9]; moreover, in the case of extreme overload conditions, an individual cloud datacenter may lack resources for scaling, thus requiring coordinated actions across several datacenters [10], [11]. For these reasons, NFV requires additional solutions for mitigating overloads in the *short-term* (i.e., within few tens of seconds), by rejecting or dropping the traffic in excess with respect to the capacity of the network.

Overload control solutions must also take into account the additional deployment constraints that are imposed by the "as-a-service" model of cloud computing, for both Virtual Network Function (VNF) and NFV Infrastructure (NFVI) providers. On the one hand, providers of *VNFaaS* must face the lack of control of the underlying public cloud infrastructure, limiting the opportunities to introduce overload control solutions at infrastructure level. On the other hand, NFVI providers have little visibility and control on VNF software, since it will be distributed and deployed as black-box VM images on their *NFVIaaS* [12]. In this case, overload control should not rely on the cooperation of VNF software.

We propose a novel overload control framework for NFV (*NFV-Throttle*) to protect NFV services from overloads within a short period of time, by tuning the incoming traffic towards VNFs in order to make the best use of the available capacity, and to preserve the QoS of traffic flows admitted by the network services. The architecture of the framework has been designed to fit into the *VNFaaS* and *NFVIaaS* services models, by taking into account the separation between service and infrastructure providers. To this purpose, the framework provides a set of modular *agents* to detect and to drop the traffic in excess, that can be installed either at VNF- or at NFVI-level without requiring changes to VNF software. Moreover, we provide general design guidelines, and a reference implementation, to let NFV designers to introduce user-defined heuristics for controlling the traffic drop/reject rates.

We performed an extensive experimental evaluation of the proposed solution on an NFV-oriented IMS, namely *Clearwater* [13], which has been designed to support massive horizontal scalability, and adopts popular cloud computing design patterns and technologies, including the OpenStack/KVM virtualization stack. We performed experiments for both VNFaaS and NFVIaaS use cases. The experimental results reveal that:

- In both cases, the solution is able to timely protect the VNF network from severe overload conditions, even up to 1000% of the engineered capacity, by sustaining a high system throughput;
- The overload control solution has a small performance cost: the CPU overhead is at most 1.5% under non-overload conditions, and less than 4% under the most

This work has been supported by Huawei Technologies Co., Ltd., and by UniNA and Compagnia di San Paolo in the frame of Programma STAR (project FIDASTE).

The authors are with the Department of Electrical Engineering and Information Technology (DIETI) of the Federico II University of Naples, Italy, and with Critiware s.r.l., Italy. E-mail: {name.surname}@unina.it

severe overload conditions (1000%); the memory overhead is up to few MBs and is constant.
- Overload control at the infrastructure-level can achieve the best performance; but overload control at the VNF-level can achieve comparable performance, and is a suitable solution for VNF providers that cannot control the underlying infrastructure. Moreover, network-level overload control is useful to notify users about an overload condition inside the network, in order to let them to gradually reduce the workload.
- Even if the Clearwater IMS already embeds overload control mechanisms [14]–[16], it can still experience software failures and low performance under severe overload conditions. The proposed solution is able to prevent these failures and outperforms these overload control mechanisms.

The paper is structured as follows. Section II discusses the problem of overload control in NFV, the goals of the proposed solution, and related work on overload control in cloud computing and computer networks. Section III presents the design of the proposed solution. Section IV provides the results of the experimental evaluation of the proposed solution. Section V concludes the paper.

II. Background and Related Work

A. The problem of overload control in NFV

We consider overload conditions in which the network traffic significantly exceeds the processing capacity of VNFs (such as during mass events). The overload may cause the disruption of already-established connections and the unavailability of high-priority services, thus violating SLAs. Even worse, an overload condition exposes the VNFs to cascading failures: the overload may create the conditions (e.g., resource exhaustion) that trigger subtle bugs in VNF software, such as memory leaks, buffer overruns, and race conditions, that cause the crash of VNF software. For example, we found several bugs of this kind in the software repository of the IMS studied in this paper [17, pulls #598, #517, #551]. In turn, a failure of a VNF replica increases the load for the remaining replicas, thus repeating the cycle and causing a cascade of failures. The problem is further exacerbated by user retries, in which the failed requests generate more and more traffic [6].

This problem is exemplified in Fig. 1. Typically, the network capacity is designed according to technical and economical considerations, in order to support some "Reference load" (point C1), for example in terms of amount of traffic per second. Under this load level, the network can perform well, and assures an "engineered throughput". However, when a mass event or a cascade failure occurs, the network becomes overloaded ("Overload condition", point C2 in the figure). The network does not have enough resources to process all the incoming flows. Thus, if the overload condition is not managed, the network throughput can significantly degrade (dashed curve in the figure). Ideally, using overload control, the network should maintain a steady throughput (for example, no lower than 90% of the engineered throughput, the continuous curve in the figure) even under an overload condition, by dropping or rejecting the traffic in excess, in order to accept only few traffic flows in the network, and by efficiently using its resources.

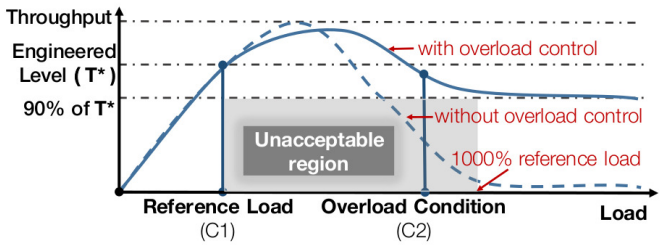


Fig. 1: Network throughput under overload conditions.

According to this view, the NFV overload control solution should consider the following requirements:

1. The NFV network should achieve an acceptable level of service (for example, not less than 90% of its engineered throughput) during severe overload conditions (such as 10 times the reference load).

2. The overload control solution should quickly react to an overload condition, in order to prevent violations of SLAs during the transition between a normal load and the overload condition. Since carrier-grade services can afford only few minutes of downtime per year, it is important to react to overloads within few tens of seconds at most.

3. The overload control solution should be integrated with the use cases and scenarios of NFV, including VNF providers, and NFVI providers. For VNF providers, it is desirable that the solution is transparent to VNF software, which can be developed by third-party vendors and whose source code may not be available. Moreover, the solution should allow NFVI providers to perform overload control at the infrastructure-level, without relying on cooperation of the VNF layer.

4. The overload control solution should introduce minimal overhead, and must not degrade the quality of service under normal load conditions (for example, it should not filter traffic when processing resources are available).

B. Related work

Performance issues in the cloud and in NFV represent a broad research field, where a variety of performance-related problems have been addressed [18]–[26]. Most of the existing approaches focused on overload prevention, that is, to scale a system in advance to avoid bottlenecks and to reserve resources for the expected workload. Many approaches provide models and algorithms for the optimal VNF placement [27]–[30] and routing [31], [32] to reduce cost and to guarantee the quality of service requirements [33]–[35]. The resource allocation is either decided off-line (i.e., before the VNFs are deployed), or, as in the case of Pham et al. [36], on-line to accomodate for dynamic changes of user demands. However, these solutions are meant for long-term capacity planning, but not for addressing sudden, unexpected overload conditions.

An active research area focuses on cloud elasticity to scale network functions [37], [38]. Indeed, cloud elasticity can be adopted in two ways: predictive and reactive. The predictive approach can be used when overload conditions can be predicted in advance, for instance by forecasting periodic workload variations (e.g., weekly and monthly trends), and by scaling the system in advance. To this purpose, researchers have been investigating novel elastic architectures for NFV,

such as the UNIFY framework [37], [39]. However, these solutions cannot counteract unexpected overload conditions, such as the ones caused by sudden traffic spikes and software bugs, which necessarily require reactive solutions. When used in the reactive way, cloud elasticity increases the system resources as soon as an overload has been detected, by deploying new VMs and/or triggering complex reconfigurations to accommodate the new workload [8]–[11]. However, cloud elasticity can require a long amount of time (e.g., up to several minutes). For this reason, cloud elasticity is not sufficient to react to overload condition in ultra-high availability applications such as NFV, and other reactive countermeasures are needed.

Our overload control framework complements these solutions by adding a new layer of defense against overloads (i.e., it can be used in conjunction with other strategies to scale the VNF network). For example, in the case of a sudden overload, our solution is quickly turned on and throttles the traffic, to only accept incoming traffic that could be served with a good quality of service with the current resources. In the meanwhile, the network of VNFs can be scaled and re-configured in order to increase the capacity of the network and to accommodate for more traffic.

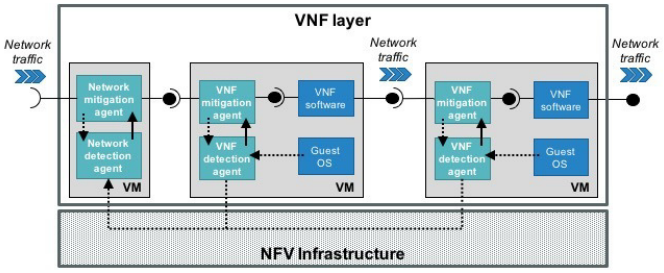
In general, traffic throttling solutions detect an overload state by monitoring the performance at the application layer (e.g., HTTP, SIP) and the resource consumption of the server (e.g, CPU and memory utilization). Under an overload condition, the filter protects the server from requests in excess. The filter can either reject the requests by replying with application-level messages (e.g., HTTP 503 "Service Unavailable"), or silently drop them. Traffic throttling solutions has been developed for internet applications as well as for carrier grade telecom appliances. Welsh et al. [14] proposed an adaptive overload control approach based on a *token bucket* and a closed control loop according to a service latency model. A similar approach has been included in the NFV software [15]. However, latency-based approaches suffer from poor performance due to cloud variability: the same kind of request can be served by a different group of nodes deployed on physical machines with different performance that are geographically distributed. Kasera et al. [40] discusses the performance of two different throttling algorithms in the context of carrier-grade telecom switches: the first algorithm is a variant of the *Random Early Discard* (RED [41]) in which the traffic is throttled according to the request queue size; the second is the *Occupancy* algorithm that ensures a target CPU utilization by throttling the traffic according to the CPU utilization and the rate of accepted calls.

In this work, we develop a traffic throttling framework that takes into account the NFV architecture and its uses cases, such as VNFaaS and NFVIaaS. One important difference is that, in previous approaches, the service to be protected was completely managed by only one provider; instead, in cloud computing, the management of the stack is divided between service and infrastructure providers, and services from several providers can be combined into service chains. According to these use cases, we propose an architecture in which the components can be deployed by different actors (e.g., NFVIaaS and VNFaaS providers). Moreover, cloud and NFV systems extensively adopt replication and load balancing on a much larger scale than the systems considered in previous studies. For example, our industrial partners have systems with thousands of replicas. Thus, we propose a hierarchical overload control architecture, including both global and local protections against overload conditions.

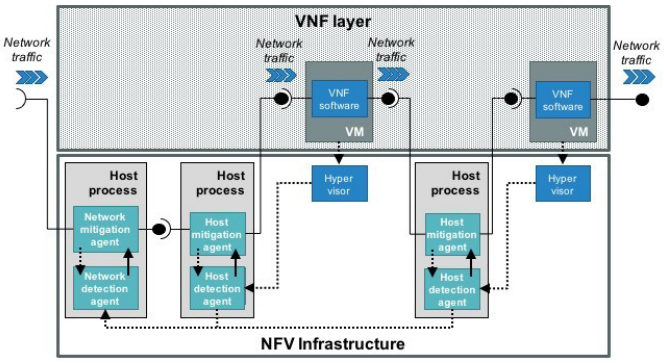
III. The Proposed Overload Control Solution

In the following, we describe an overload control solution aimed at fulfilling high-availability and performance requirements of telecom services, and at complying with the service models of NFV and cloud computing. In particular, we consider two main use-case scenarios:

1) A telecom operator designs a network service (e.g., to offer it as a service, *VNFaaS*), by assembling VNFs and composing them into a VNF service chain (see Fig. 2a). The VNFs can run VNF software developed in-house or provided by third-party NFV software vendors. The VNFs are deployed on an NFVI managed by a third-party NFVI provider (*NFVIaaS*). In this scenario, the telecom operator can customize the VNFs and deploy VMs on the NFVI, but it cannot change the underlying NFVI.
2) An NFVI provider manages an infrastructure (e.g., to offer it as a service, *NFVIaaS*) to host VNFs from telecom operators (see Fig. 2b). In this scenario, it is desirable (or even mandatory, if the VNFs are provided as black-boxes) to address overloads at the infrastructure level, without making changes to VMs.



(a) Deployment managed by the VNF provider.



(b) Deployment managed by the NFVI provider.

Fig. 2: Overview of the overload control solution.

The proposed solution is an overload control framework based on a set of *overload detection agents* and *overload mitigation agents*. These agents are software modules to be deployed inside the NFV network, and transparent to VNF software (Figure 2):

- **Overload detection agents** check whether the incoming traffic towards the VNF exceeds its capacity due to

a workload peak. If an overload condition occurs, the detection agent triggers an overload mitigation agent.
- **Overload mitigation agents** protect the VNF from incoming traffic in excess, by dropping it, or by only admitting a subset of users to the service, and it allows again the traffic once the overload condition disappears.

The overload detection and mitigation agents are further divided in three complementary types. The *VNF-level agents* protect individual VNFs from inside their VMs, and react to overload conditions by dropping traffic in excess. The *host-level agents* also protect individual VNFs by dropping traffic, but they run outside VMs. Finally, *network-level agents* protect the NFV network from overload condition that affect the whole network (i.e., overloads spread across several VNFs), and react by rejecting traffic and notifying the clients about the overload condition.

The agents can be deployed across the NFV network to support any of the two use-case scenarios mentioned before:

1) A telecom operator can install the detection and mitigation agents in the same VMs of the VNFs, or in dedicated VMs (Fig. 2a). The **VNF-level** agents collect resource utilization metrics from VMs, and forward traffic to VNF software through a transparent network tunnel, which drops traffic in excess in the case of an overload. Moreover, **network-level** agents are deployed on dedicated VMs, and are interposed between the NFV network and external networks. They detect overload conditions that are spread across several VNFs, and use a transparent network tunnel in order to forward network traffic and to reject traffic in excess.
2) An NFVI provider may not be allowed to install agents inside the VMs of VNFs, but has the opportunity to install agents in the physical hosts of the NFV infrastructure (Fig. 2b). **Host-level** detection and mitigation agents are deployed as processes or services running on the physical hosts. The host-level agents use a transparent network tunnel towards each VNF, by leveraging virtual networking mechanisms provided by the infrastructure, in order to protect an overloaded VNF from ingoing traffic in excess. In a similar way, **network-level** agents can be deployed on physical hosts and can be interposed between the NFV network and external networks.

Our framework encompasses two kinds of agent: i) VM-level agent, which implements a traffic control strategy at *local level*, depending on the node status, and ii) Network agent, which implements a traffic control strategy at *global level*, depending on the network status. Both agents are active at the same time, and cooperate each others to protect VNF from the different overload scenarios, as described in the following.

- The VNF-level agents are deployed on individual VNFs; instead, the network-level agents are deployed on the entry point of the service function chain, and throttle the traffic before it enters the service chain and it is forwarded to all the VMs containing the VNFs.
- The "global decision" performs a different, additional action than "local decisions", in order to further mitigate overload conditions. The VNF-level agents drop the traffic in excess (packets are not delivered to the VNF); instead, the network-level agents do not simply drop traffic, but they also send reject notifications to the clients, in order to notify them to reduce the traffic (e.g., to avoid avalanche effects caused by user retries).
- The "global decision" is made at a different time than the "local decisions". The VNF-level agents can make local decisions more frequently (e.g., every few seconds), since they only use metrics that are collected from the local VM, and can quickly mitigate the overload state. Instead, the network-level agents make a "global decision" less frequently (e.g., every few tens of seconds), since it needs to collect the state from the VNF-level agents and throttle if a majority of them is in overload state. This different timing allows the network-level agents to trigger the "global decision" only if the "local decisions" persist for a long time and affect several VNFs: in this way, the network-level agents do not over-react (e.g., the overhead of sending reject notifications to users) when an overload condition lasts for a very short time, or when there is a transient, random spike in resource utilization.
- Since the network-level overload control is triggered after a longer time period than the VNF-level overload control, it uses a different update rule (e.g., multiplicative instead of additive) in order to be able to increase/decrease the rejection rate more quickly, and to be able to react against very high load conditions for which the VNF-level overload control may not be sufficient.

The proposed overload control framework is designed to react to overload in the short term (e.g., few tens of seconds), and is complementary to elastic cloud computing mechanisms that expand the capacity of VNFs. The framework does not require to change VNF software and virtualization software, and can be transparently installed into NFV networks with third-party VNF software and virtualization technologies. The overload detection agents only rely on metrics that are widespread across guest OSs and hypervisors, and that are easily collectible through APIs or IPC channels exposed by the guest OSs and hypervisors, without modifying their internals. Moreover, the solution gives to NFV designers and administrators the ability to install agents only for specific VNFs, where overload control is most needed; to reuse the overload control framework across different types of network functions; to address overload either at VNF- or at host-level, and/or globally at the NFV network level. For example, the NFV designers can take into account special requirements of the users, such as to deploy VNF-level agents to protect a certain high-priority VNF; and to not deploy the network-level solution for protecting an individual VNF that is not replicated.

A. VNF-level design

The architecture of the overload control solution at VNF level is showed in Fig. 3, which includes a detection agent and a mitigation agent.

The **VNF-level Detection Agent** is a component deployed by a VNF provider inside a VM, in order to address overloads of an individual VNF (Fig. 2a). It collects resource utilization metrics from the VM, by using interfaces exposed by the guest OS (such as the *procfs* virtual filesystem of the Linux OS). Specifically, it collects metrics about the utilization of virtual CPU by the VM. These metrics include the *busy* virtual CPU ticks, consumed both by user-space applications (including

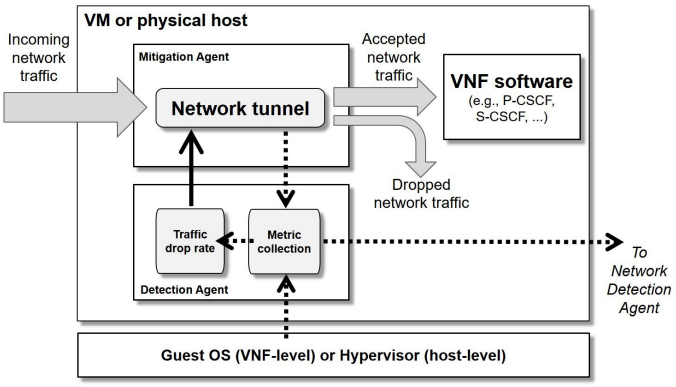


Fig. 3: Architecture of VNF- and host-level overload control.

Algorithm 1: VNF- and host-level overload control

Data: SP: sampling period
Data: N: size of the vector of samples
Data: *reference_cpu_usage*: "factor of safety" for virtual CPU usage
Result: $drop\_rate$ for incoming VNF traffic
begin
  while *True* **do**
    collect *cpu_ticks*, *incoming_traffic* and *accepted_traffic* measurements;
    update the *drop_rate*;
    send *cpu_ticks* to the Network-level Detection Agent;
    send the updated *drop_rate* to VNF-level Mitigation Agent;
    wait SP seconds;

VNF software) and by the guest OS (including system calls and interrupt service), and the *idle* CPU ticks of the VM. Moreover, the VNF-level Detection Agent collects from the VNF-level Mitigation Agent the throughput of VNF traffic, in order to correlate the CPU utilization with the amount of incoming network traffic.

The output of the VNF-level Detection Agent is represented by the *traffic drop rate*, i.e., the percentage of input VNF traffic not to be accepted. This rate is continuously tuned by the VNF-level Detection Agent, according to the general steps showed in Algorithm 1.

In general, the traffic drop rate should be null if the workload is within the capacity of the VNF (e.g., the CPU utilization is below a limit). If the CPU utilization approaches saturation, then the agent must drop part of the input traffic to reduce the load on the VNF software; the agent should only let in enough traffic in order to keep the CPU at full utilization, but without overloading it. In our framework, we allow NFV designers to plug-in their own user-defined heuristics to control the drop rate. In our reference implementation of the framework, we tune the drop rate proportionally to the CPU utilization and to the amount of input traffic, as discussed below.

The **VNF-level Mitigation Agent** acts as a network tunnel between the VNF software and other VNFs in the NFV network. The network traffic towards the VNF software is first forwarded to the VNF-level Mitigation Agent. In turn, the VNF-level Mitigation Agent connects to the VNF software, and it forwards the traffic to the VNF software.

This forwarding is accomplished by using network traffic forwarding mechanisms that are provided by the guest OS. For example, in the case of the Linux OS, the *iptables* network utility can be used to introduce a forwarding rule inside the guest OS, to redirect VNF traffic, according to the destination port, to a different network port that is exposed by the VNF-level Mitigation Agent.

The VNF-level Mitigation Agent is transparent to the VNF software, and has only a small impact on network latency and throughput, since it does not perform any traffic analysis or manipulation. The VNF-level Mitigation Agent only computes metrics on network throughput, and sends these metrics to the VNF-level Detection Agent.

When an overload condition occurs, the VNF-level Mitigation Agent filters out part of the input network traffic, in order to protect the VNF software from the traffic in excess (which is dropped and not forwarded to the VNF), according to the traffic drop rate configured by the VNF-level Detection Agent.

The VNF-level Mitigation Agent applies a traffic-matching rule on the contents of network traffic (such as, to a "type" field in the header), in order to identify which network traffic it should drop. For example, in the case of the SIP protocol, it is preferable to only drop "REGISTER" and "INVITE" requests in excess, and not to drop other types of messages. In this way, new users are prevented from registering, and the VNF software is protected from the overload caused by new users that try to enter in the network. Moreover, the users that are already registered are not affected by the traffic drop, and do not experience any degradation of the quality of service.

We implemented a reference version of the framework, by defining a set of heuristics to control the drop rate. In our implementation, the agent updates the *drop_rate* using a simple and robust rule: it allows all the incoming traffic if the CPU utilization is below a *reference value*; instead, if the CPU utilization exceeds the reference, the allowed traffic is scaled proportionally to the gap between the reference and the actual CPU utilization. The update rule is defined as:

$$\text{capacity} = \frac{\text{MEAN}[\text{accepted\_traffic}[1 \ldots N]]}{\frac{\text{MAX}[\text{cpu\_usage}[1 \ldots N]]}{\text{reference\_cpu\_usage}}} \qquad (1)$$

where *cpu_usage* is a sliding window of the latest N samples of the percentage of busy virtual CPU ticks (up to 100%, and strictly greater than 0); the *accepted_traffic* is the volume of traffic that is actually passed to VNF software by the agents.

When the volume of traffic in input to the VNF-level Mitigation Agent (i.e., *incoming_traffic*) exceeds the *capacity*, we evaluate the fraction of the traffic to drop as:

$$\text{drop\_rate} = 100 \cdot \left(1 - \frac{\text{capacity}}{\text{incoming\_traffic[N]}}\right) \qquad (2)$$

if incoming_traffic[N] > capacity; otherwise, drop_rate = 0.

In order to quickly react to an overload condition within a short time frame (e.g., 10 seconds for critical NFV networks), the VNF-level Detection Agent collects a new sample of resource utilization metrics (*cpu_usage*, *incoming_traffic*, and

accepted_traffic) at a high frequency (later in this study, we configure the agent to collect one sample every 2 seconds). Moreover, the VNF-level Detection Agent analyzes the most recent N samples (e.g., we consider the last $N = 5$ samples when sampling every 2 seconds) of virtual CPU utilization and of the network traffic throughput. The VNF-level Detection Agent first identifies the highest virtual CPU utilization sample among the recent samples, and compares it to the reference virtual CPU utilization.

The reference virtual CPU utilization is chosen by NFV designers or administrators: it represents a "factor of safety" for virtual CPU utilization, under which the VNF is designed to perform well (e.g., no service disruptions), as showed in Fig. 1, and which leaves a small amount of residual capacity to handle unexpected load conditions. We base our algorithm around a reference value since setting a reference is a frequent practice among designers and administrators (e.g., for monitoring and troubleshooting purposes). For example, the VNF can be designed and configured (e.g., during performance testing and capacity planning of virtual and physical resources) to have a virtual CPU utilization up to 90%, with good quality of service, under some reference workload.

If the virtual CPU utilization exceeds the reference virtual CPU utilization, the traffic allowed into the VNF (*capacity*) is reduced by the update rule (eq. (1)). The new value is obtained by scaling down the average of the most recent N samples of traffic volume accepted into the VNF; the scaling is proportional to the gap between the reference virtual CPU utilization and the actual virtual CPU utilization. Thus, the larger the gap, the lower the capacity, and the higher the traffic drop rate.

This computation is periodically repeated for each new sample of resource utilization. If the overload condition persists (i.e., the CPU utilization is still higher than the reference value), the *accepted_traffic* and the *capacity* will keep reducing, and the *drop_rate* will further increase. Instead, when the VNF leaves the overload condition (i.e., the virtual CPU utilization is below the reference value), the VNF-level Detection Agent will gradually increase the capacity and reduce the traffic drop rate, until it becomes zero (that is, all the input network traffic is again allowed in the VNF software). At each update, the traffic drop rate is sent to the VNF-level Mitigation Agent.

This approach is robust to false positives, since a sporadic increase of the virtual CPU (e.g., transient peaks in the samples that are not due to an overload condition, but are due to random effects) is quickly discarded since we adopt a relatively small window of samples (e.g., $N = 5$), which only causes to drop a small amount of traffic and a negligible impact on the quality of service. In the case of a larger window, the update rule can be changed by replacing the $\text{MAX}[\cdot]$ function with a percentile (such as the 90^{th} percentile among the N samples). Moreover, the VNF-level Detection Agent applies a moving-average filter to the samples of network traffic throughput, which lessens the effect of sporadic out-of-norm samples from network measurements.

B. Host-level design

The architecture of the overload control solution for this level includes a detection agent and a mitigation agent. These components enable NFVI providers to deploy the same level of protection of the VNF-level solution, but acting only on the NFV infrastructure layer.

The **Host-level Detection Agent** is a multi-threaded application, which can be deployed by the NFVI provider on top of the hypervisor as a privileged process (Fig. 2b). This agent replaces the VNF-level Detection Agent, in order to protect a VNF from traffic in excess when the deployment is managed by the NFVI provider. The Host-level Detection Agent monitors one or more VNFs in the NFV network. It is possible to deploy more than one Host-level Detection Agents on the same cloud infrastructure, where each Host-level Detection Agent monitors a subset of VNFs in the NFV network.

The Host-level Detection Agent collects data on virtual CPU utilization using APIs provided by the hypervisor (for example, in the case of KVM, CPU utilization of VMs can be measured using the *taskstats* interface of the Linux kernel). The Host-level Detection Agent detects the traffic in excess towards a VNF, by using the same algorithm of the VNF-level Detection Agent (Alg. 1, and eq. (1) and (2) in section III-A). For each VM, it periodically samples the virtual CPU usage of the VM, and its network throughput; then, it tunes the traffic drop ratio of individual VNFs to drop traffic. Therefore, the Host-level Detection adopts a similar architecture to the VNF-level Detection (Fig. 3).

Finally, the Host-level Detection Agent aggregates the information about the overload state of VNFs that it monitors, and sends periodic update messages to the Network-level Detection Agent, as discussed later in this section.

The **Host-level Mitigation Agent** is an application that executes in the same environment of the Host-level Detection Agent. In the case of NFVI providers, such as in NFVIaaS (Fig. 2b), the Host-level Mitigation Agent can be used to drop the traffic in excess towards individual VNFs, in a similar way to the VNF-level Mitigation Agent (section III-A). This is achieved by configuring network traffic forwarding mechanisms of the virtualization infrastructure to establish a network tunnel. This approach leverages the privileged access of NFVI providers to the infrastructure, and overcomes the lack of access to the VNF software. Moreover, this approach can potentially achieve a lower overhead than the VNF-level solution, since the traffic in excess can be dropped before it is forwarded to the VMs.

When the Host-level Detection Agent detects an overload condition, it triggers the Host-level Mitigation Agent to drop the traffic in excess. The amount and the type of traffic to drop is configured by the Host-level Detection Agent as described in section III-A: the Host-level Detection Agent updates the traffic drop ratio according to a rule that uses resource utilization metrics, and it applies traffic-matching rules to identify which traffic should be dropped.

C. Network-level design

The architecture of the overload control solution for this level includes a detection agent and a mitigation agent, as shown in Fig. 4.

The **Network-level Detection Agent** is a multi-threaded application, which executes in a dedicated VM in the same

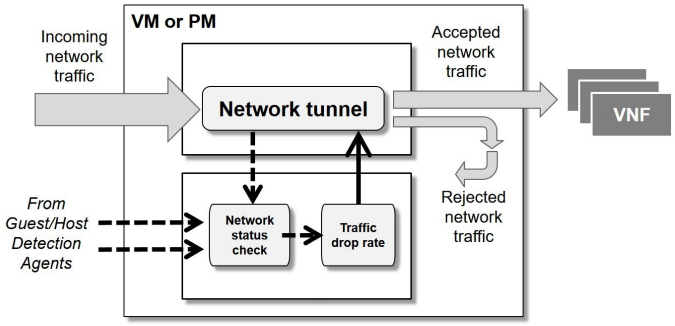


Fig. 4: Architecture of network-level overload control.

cloud infrastructure of the VMs running VNF software. Alternatively, it can execute as a privileged process on a physical machine of the NFVI.

The Network-level Detection Agent collects the status of all VNFs in the NFV network, and checks the presence of an overload condition (Alg. 2), according to overload notifications (i.e., drop rates greater than zero) coming from Host-level or VNF-level Detection Agents. The criteria for detecting a network-level overload condition can be configured by the administrators of the NFV network: a simple criterion is to count the number of VNFs affected by overload, and detect an overload state when overloaded VNFs are the majority. Another possible criterion is to compute a weighted count of the number of overloaded VNFs, by taking into account the relative importance of VNFs in the NFV network.

The **Network-level Mitigation Agent** acts as a network tunnel at the boundary of the NFV network. It receives the traffic that was originally intended for the NFV network, and forwards it to the VNFs. This forwarding is accomplished by installing the Network-level Mitigation Agent into a load balancer, placed at the boundaries of the NFV network, either in a dedicated VM, or in a physical machine. Thus, the Network-level Mitigation Agent is transparent to the VNFs. Moreover, the Network-level Mitigation Agent has only a small impact on network latency and throughput, since it does not perform any traffic analysis or manipulation. The traffic in excess is not forwarded to VNFs.

Since these agents are deployed at the boundary of the VNF network, they can prevent users from entering the VNF network by explicitly rejecting them, e.g., by replying with an overload notification to clients, in order to prevent them to generate more traffic. For example, in the case of the SIP protocol, the Network-level Mitigation Agent can reply with a "*503 Service Unavailable*" response in order to notify clients about the overload state. Moreover, the Network-level Mitigation Agent applies a traffic-matching rule on the contents of network traffic (such as, to a "type" field in a packet header), in order to identify which network traffic it should reject (such as, session initiation requests). This approach differs from the previous VNF- and host-level agents, which just dropped traffic without notifying the user.

The Network-level Detection agent iterates with a longer period than then VNF-/Host-level agents, since it is meant to mitigate overloads that persist for a longer time (e.g., we consider a time period of 30 seconds for Network-level detection). For such persistent overloads, it is worth to spend additional resources (CPU and network) to send notifications to reject the users, rather than simply dropping their traffic.

In the network-level agents, the traffic is rejected according to a *traffic rejection rate*, which is periodically updated by the Network-level Detection Agent. In general, the traffic rejection rate is gradually increased when the VNFs are in an overload condition, and it is decreased otherwise. Similarly to the previous agents, the framework allows NFV designers to adopt their own heuristics for controlling the rejection rate. In our reference implementation, we adopt a heuristic that updates the rejection rate according to a multiplicative function (differing from VNF-/Host-level Detection, which instead used an additive update rule). Since the Network-level overload control reacts after a longer time period than the VNF-/Host-level detection, then it should be able to increase/decrease the rejection rate more quickly than them, in order to be able to react against very high load conditions.

The traffic rejection rate is given by:

$$\text{capacity} = \begin{cases} \text{capacity}/(\alpha+\gamma) & \text{if overloaded} \\ \text{capacity} \cdot (\beta-\gamma) & \text{otherwise} \end{cases} \quad (3)$$

$$\text{reject\_rate} = 100 \cdot \left(1 - \frac{\text{capacity}}{\text{incoming\_traffic}[N]}\right)[\%] \quad (4)$$

in which the *reject_rate* is capped between 0% and 100%. The *incoming_traffic* is the volume of traffic in input, and α and β are constants, with $\alpha > 1$ and $\beta > 2$. The γ coefficient is a variable factor, which tunes the reject rate according to the persistence of the overload condition. It is defined as:

$$\gamma = \frac{\text{dropped\_traffic}[N]}{\text{incoming\_traffic}[N]} \quad (5)$$

that is, γ represents the fraction of traffic that has been rejected during the last sampling period. This coefficient has been introduced to keep the reject rate low if the overload condition lasts for a short amount of time, in order to soften the impact of sporadic false positives in overload detection; and, at the same time, this coefficient serves to keep the reject rate high if the overload condition is severe and persists over time. When the current fraction γ of rejected traffic is null or low, $\alpha + \gamma$ is closer to 1, thus the capacity decreases with a smaller step, and increases with a larger step. Thus, the approach avoids to reject too much traffic when the overload condition is short and sporadic. Instead, when the γ is high (which happens when the overload condition already lasted for a relatively long time), the capacity decreases with a larger step, and increases with a smaller step. In this way, if the overload condition disappears only for a small amount of time, the reject rate is still kept high; the full capacity is restored only once the network becomes stable and non-overloaded.

IV. Experimental evaluation on a NFV IMS

A. Overview of the NFV System

To evaluate the overload control framework, we performed an experimental analysis on an NFV-oriented IP Multimedia Subsystem (IMS). We evaluate the ability of the overload control framework in the context of a real NFV software, in terms of performance under overload conditions, overhead of the framework, and failures of NFV software.

Algorithm 2: Network-level overload control

Data: SP: sampling period
Data: $VNF[1 \dots M]$: overload state of monitored VNFs
Result: *reject_rate* for incoming VNF Network traffic
begin
  while *True* **do**
    foreach *node k in VNF* **do**
      collect overload state for $VNF[k]$
    if *majority of VNFs is overloaded* **then**
      decrease *capacity*;
    else
      increase *capacity*;
    update and send the *reject_rate* to the Network-level Mitigation Agent;
    wait SP seconds;

The Clearwater IMS [13] is an open-source implementation of the IMS core standard [42]. IMS functions are implemented in software and packaged in VMs, and are designed to take full advantage of virtualization and cloud computing technology: thus, it is possible to deploy a Clearwater instance on industry-standard IT virtualization platforms. It is modular, and its deployment can be customized by only enabling specific functions. All components can scale out horizontally using simple, stateless load-balancing based on DNS. Moreover, Clearwater follows common design patterns for scalable and reliable web services, by keeping most components largely stateless, and by storing long-lived state in clustered data stores. Clearwater is a large software project, mostly written in C++ and Java, and including several subsystems. The architecture of Clearwater core is showed in Fig. 5, and includes the following components:

- ***Bono*** (P-CSCF): The Bono nodes are the first point of contact for an UE (User Equipment), and they represent the edge proxy providing P-CSCF standard interfaces to IMS clients. They are replicated to form a cluster of proxy nodes, allowing load balancing of requests at the boundary of the system.
- ***Sprout*** (S-CSCF and TAS): The Sprout nodes are horizontally-scalable SIP registrars and authoritative routing proxies. These nodes implement the S-CSCF and I-CSCF interfaces of the IMS standard. Furthermore, they implement a distributed cache, using Memcached [43], for storing registration data and other long-lived information.
- ***Homestead***: The Homestead nodes are horizontally-scalable, redundant mirrors for the HSS (Home Subscriber Server) data store, using Apache Cassandra [44], for retrieving authentication credentials and user profile information. HSS mirrors are part of both the S-CSCF and I-CSCF interfaces, and provide Web services (over HTTP) to the Sprout layer.
- ***Homer***: A Homer node is a XML Document Management Server (XDMS) to store service settings documents for each user of the system, using Apache Cassandra as the data store.
- ***Ralf*** (Rf-CTF): The Ralf nodes provide charging and billing functions, used by Bono, Sprout and Homestead nodes to report events occurring when the CSCF chain is traversed.

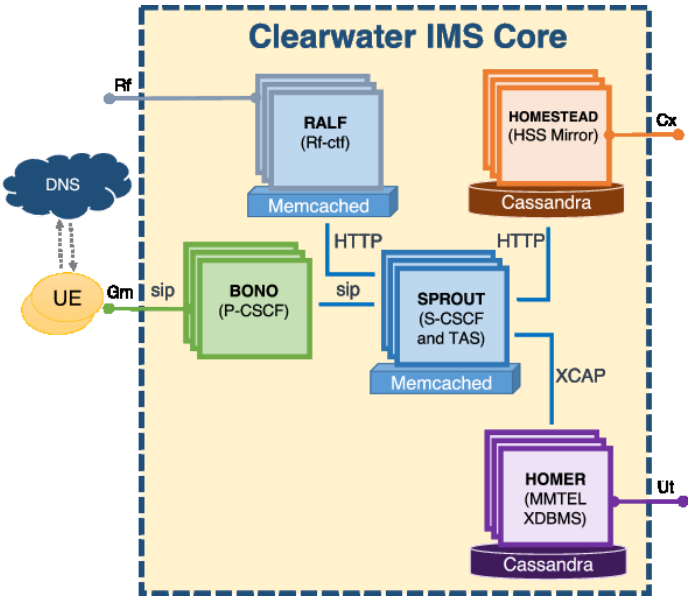


Fig. 5: Architecture of the Clearwater IMS

Our framework does not require any change to the Clearwater architecture or to the VNF software. The NFV-Throttle agents are stand-alone components deployed as processes in dedicated VMs and in the physical nodes, according to the two use cases discussed in Section III, Fig. 2.

The Clearwater IMS includes a throttling mechanism, which rejects requests in excess to avoid overloading a node [15], [16]. It uses a *token bucket* to control the rate of requests that a node is allowed to process. The token replacement rate is tuned by measuring the latency for processing requests, and by comparing, every twenty requests, this measure with a configured latency target. Clearwater adopts a variation of the algorithm proposed by Welsh and Culler [14], by using a smoothed mean latency to compare with the latency target. In our experiments, we evaluate our overload control framework with respect to this overload control solution included in the Clearwater IMS.

We treat the VNF software as a black-box, and in all of the evaluation experiments of Section IV-D, we enable the Clearwater's built-in throttling mechanism. Thus, we compare the performance of the IMS by considering the case where the IMS is only protected by its built-in mechanism, and the case where the *NFV-Throttle* agents are deployed in the system.

The agents of the *NFV-Throttle* framework are deployed both on the VMs (VNF-level agents) and on the hosts (Host-level and Network-level agents). The agents have been developed in C, respectively as background daemons at VNF-level and Host-level, and as extensions of the OpenSIPS [45] proxy at Network-level. The VNF-level and Host-level detection agents collect CPU utilization metrics from the *proc* FS, and network utilization metrics from the mitigation agents. The mitigation agents, both at VNF-level and at Host-level, act as a filtering proxy for all the network traffic destined to a specific VNF, using *iptables* NAT rules. The network traffic

accepted by the agent is forwarded to the VNF, on behalf of the originator. In the case of UDP traffic (such as in the case of SIP clients and P-CSCF VNF instances), UDP datagrams are forwarded by the agent on behalf of the source, by replacing the source IP and port with the ones of the SIP client; thus, the destination (e.g., P-CSCF) can directly reply to the source. In the case of TCP traffic (such as between a P-CSCF VNF and a S-CSCF VNF), during the TCP handshake phase, the mitigation agents establishes two connections, respectively with the source (e.g., P-CSCF) and the destination (e.g., S-CSCF). Then, the agent reads and writes data from both the two streams, acting as a man-in-the-middle. Note that, for both UDP and TCP, only a fixed set of *iptables* rules is required, regardless of the number of clients and connections.

B. Experimental testbed

The experimental testbed (Fig. 6) consists of four host machines: three Dell PowerEdge R520 servers, equipped with two 8-Core 2.2 GHz Intel Xeon CPU, 64GB DDR3 RAM, two 500GB SATA HDD, two 1-Gbps Ethernet NICs, 8-Gbps Fiber Channel HBA; one Dell PowerEdge R320 server with a 4-Core 2.8 GHz Intel Xeon CPU, 8GB DDR3 RAM, two 500GB SATA HDD, two 1-Gbps Ethernet NICs, 8-Gbps Fiber Channel HBA; A PowerVault MD3620F disk array with 4TB of network storage with a 8-Gbps Fiber Channel link.

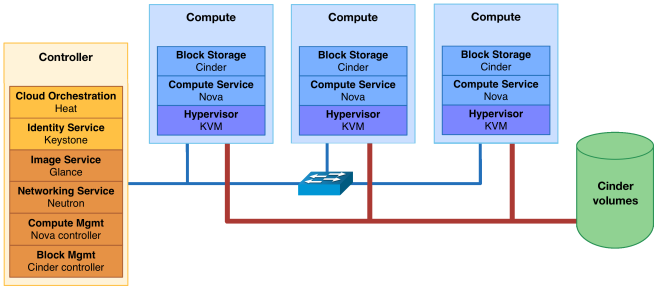


Fig. 6: Experimental testbed

The hosts are connected to a 1-Gbps Ethernet network for general-purpose traffic, and another 1-Gbps Ethernet network for management traffic. The virtual disks of VMs are stored on three distinct GlusterFS partitions of the PowerVault SAN, which are mounted on the hosts through the Fiber Channel link.

The hosts are configured with CentOS Linux 7 and the KVM hypervisor. The testbed is managed using the *OpenStack* virtualization platform, version Juno [46]. The Dell PowerEdge R320 serves as OpenStack Controller and Network node; the three Dell PowerEdge R520 servers represent the OpenStack Compute and Storage nodes, and run the VMs of the Clearwater IMS. The OpenStack services include: *Nova*, which manages the compute domain; *Neutron*, which manages virtual networks among VMs; *Cinder*, which controls the lifecycle of VM volumes; *Glance*, which manages the cloud images of VMs; *Heat*, which orchestrates, through a native REST API, the virtual IMS deployment; *Horizon*, which supports the Web-based management dashboard.

C. Experimental plan

We evaluate the proposed overload control framework in the context of the Clearwater IMS case study, by performing experiments with stressful workloads and resource contention. In particular, we evaluate:

- The ability of the framework to assure a high throughput (up to the maximum capacity of the system), in terms of register attempts per second (RAPS) and call attempts per second (CAPS) that are successfully handled with no failures (i.e., requests that are neither timed-out nor rejected).
- The resource overhead introduced by the framework, in terms of **CPU** and **memory footprint** consumed by the agents of the overload control framework.

The experiments use a mix of SIP registrations and call setup requests. The workload is generated using the *SIPp* traffic generator [47] to emulate SIP subscribers. Each couple of subscribers will attempt to register or renew the registration every 5 minutes on average. After a successful registration, a subscriber can either attempt to setup a call to the other (with 16% of probability), or remain idle until the next registration renewal (with 84% of probability). The call hold time is configured to 60s. We calibrate the number of VNF instances with a preliminary *capacity planning* using 400k subscribers. These numbers have been suggested by our industrial partners as a realistic baseline for testing an IMS service, and on which we impose overload conditions.

We tuned the number of VNF instances to have at most 80% virtual CPU utilization on average, and no failed requests. The IMS can handle this workload with 10 replicas of Bono, Sprout, and Homestead, 4 replicas of Ralf, and 1 replica of Homer and DNS. Each replica runs on a distinct VM with 1 virtual CPU. In this experimental setup, 400k subscribers represent the **engineered capacity** of the IMS (see also Fig. 1). The IMS experiences an overload condition when the number of subscribers exceeds this engineered capacity ($\gg$ 400k subscribers). In these cases, the CPU becomes the performance bottleneck. Moreover, an overload condition happens when the IMS compete for resources with other services that are deployed on the same physical infrastructure.

We consider three high-workload scenarios to evaluate the performance of overload control under a peak of subscribers. Every scenario is executed four times, respectively with: the plain Clearwater IMS; the VNF-level overload control; the Host-level overload control; and the Network-level overload control. In total, we perform 12 high-workload experiments. We adopt the following workloads (TABLE I):

- **Small overload** (480K subscribers): the load is at 120% with respect to the engineered level, and saturates the maximum capacity of the testbed. At this load level, the overload control solution should throttle a small part of service requests, in order to preserve the QoS for subscribers that are already registered in the IMS before the overload.
- **Medium overload** (1M subscribers): the load is 250% with respect to the engineered level, and above the maximum capacity of the testbed. At this load level, the overload control solution should throttle a large amount of requests to prevent a significant throughput degradation.

TABLE I: Workloads for evaluating the overload control solution.

# Subscribers	Load Level	RAPS	CAPS
400k (Engineered Level)	**100%**	**1,379**	**111**
480k (Small Overload)	120%	1,655	133
1M (Medium Overload)	250%	3,448	278
4M (High Overload)	1000%	13,793	1,111

- **High overload** (4M subscribers): the load is ten times higher (1000%) than the engineered level. At this load level, there is a significant resource pressure since a considerable amount of connections must be handled, thus exposing the IMS software to potential crashes due to resource exhaustion.

Each experiment lasts 1 hour, and is divided in three phases:

- **Load generation (Ramp-up)**: When the experiment starts, 400k subscribers are created in the initial 15 minutes (*Initial ramp-up period*). The system can handle this load without failures. This load is generated by a set of 10 SIPp instances, for all the duration of the experiment. This phase is common to all the experiments.
- **Overload generation**: This phase starts at the 20th minute, and lasts for 30 minutes. In this phase, additional subscribers, over the engineered level (TABLE I), are introduced in a short amount of time (*Overload Ramp-up period*). Then, all the subscribers constantly generate requests for call setup and registration renewal.
- **Overload termination (Ramp down)**: This phase starts at the 50th minute, and lasts until the end of the run. In this phase, each subscriber that fails to register or make a call, will not attempt to retry and will leave the system.

D. Experimental results

1) Node level: We first consider the case in which overload control is performed only at the VNF-level (i.e., by installing an agent inside VMs, as discussed in Section III-A). Fig. 7 shows the performance of the Clearwater IMS at varying levels of overload, respectively at 120%, 250% and 1000% load with respect to the engineered capacity.

The graphs of Fig. 7 shows the registration throughput on the left side, and the call throughput on the right side. Each graph shows three curves: the input load, in terms of registration and call requests per second, and the throughput of successful requests, respectively with NFV-trottle enabled, deployed at VNF-level (in yellow), and without our overload control solution (i.e., only with Clearwater's built-in overload control, in red).

With an overload level of 120% (Fig. 7a and 7b) the registration and call throughput are close to the input request rate, both with and without the proposed overload control framework. In both cases, the capacity of the IMS has been saturated. However, in the case without our overload control framework, the call throughput exhibits a significant variability, and tends to be lower than the input rate of requests. This behavior is a consequence of the problem discussed in Section II: even if resources are fully utilized, they do not necessarily produce useful work, since the system attempts to manage too many users but cannot provide an acceptable QoS to any of them. Instead, the overload control solution has been able to avoid service failures for already-established sessions, by rejecting the requests in excess during the overload phase.

With higher overload levels (Fig. 7c – 7d at 250% load, and Fig. 7e – 7f at 1000% load), and without our overload control framework, the impact of overload is even more severe. We observed that, despite the built-in overload protection, most of the nodes exhibit *failures due to resource exhaustion*, causing the crash of VNFs and performance degradation of the IMS. This results in a significant performance degradation, with registration throughput lower than 200 RAPS and call throughput lower than 1 CAPS in the worst case.

In the same scenarios, with the overload control framework, the registration and call throughput are stable around the engineered level, which is the maximal throughput attainable by the IMS. Moreover, there are no failures of the VNFs, since the overload control framework is implemented outside VNF software and is more robust to huge overload conditions. In particular:

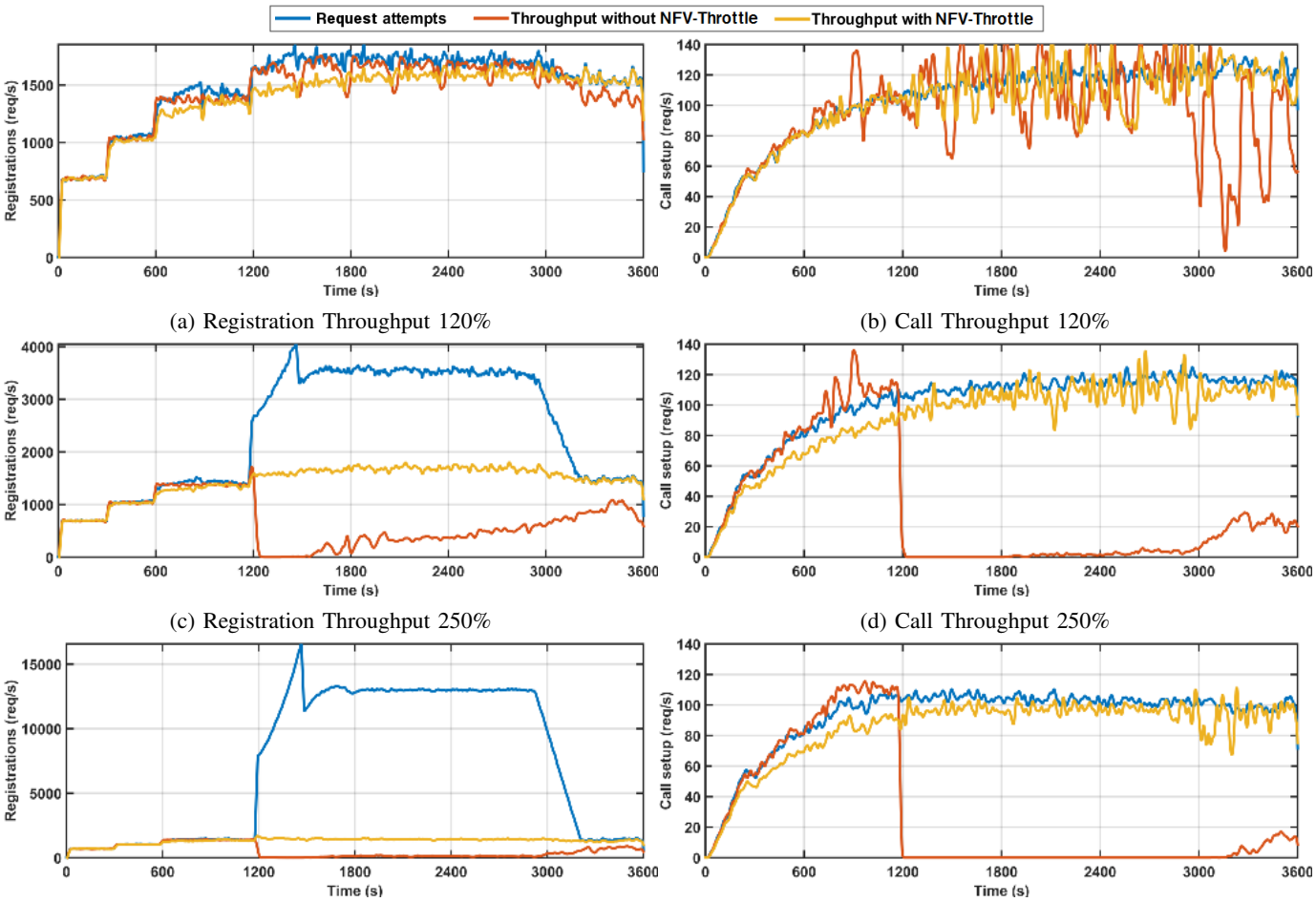
- Load level of 250%: (1) the registration throughput reaches on average 1664.9 RAPS, which is 137% more than the case without the mitigation and 20% more than the engineered level; (2) the call throughput reaches on average 114.60 CAPS, which is 194% more than the case without the mitigation and close to the engineered level.
- Load level of 1000%: (1) the registration throughput is 1439.5 RAPS, which is 152% more than the case without the mitigation and 4% higher than the engineered level; the call throughput reaches on average 97.17 CAPS, which is 200% more than the case without the mitigation and close to the engineered level.

2) Host level: We performed the same experiments using the Host-level overload detection and mitigation, which replace their VNF-level counterparts. The results obtained with Host-Level overload control are comparable to VNF-level overload control in Fig. 7, thus they are not shown here for the sake of brevity. It is worth to mention that the overload control framework, under an overload level of 250%, can achieve a registration throughput 136% higher than the case without the mitigation, and 18% more than the engineered level. Similar results were also obtained under an overload level of 1000%. Again, overload control prevented IMS failures due to resource exhaustion.

3) Network Level: Fig. 8 shows the performance measurements obtained with Network-level overload control. As for the previous cases (VNF-level and Host-level), the overload control at Network-level is able to sustain a high throughput in all overload scenarios. Moreover, with a load level of 250% and 1000%, the control solution is able to avoid resource exhaustion and crashes of IMS components.

These experiments point out an additional benefit of Network-level overload control. During the overload, the Network-level mitigation agent rejects the registration requests in excess by replying to the clients. During the first 10 minutes of overload (in the period 1200s-1800s in the graphs), the rate of incoming registration requests gradually decreases due to rejections, and stabilizes again around the engineered capacity.

Fig. 9 summarizes the results, by providing aggregated statistics (median, upper and lower quartiles, minimum and



(a) Registration Throughput 120%

(b) Call Throughput 120%

(c) Registration Throughput 250%

(d) Call Throughput 250%

(e) Registration Throughput 1000%

(f) Call Throughput 1000%

Fig. 7: Registration and Call Throughput for each overload level (i.e., 120%, 250% and 1000%) at Node Level.

maximum) obtained respectively with overload control at VNF-level, Host-level, and Network-level. Figure 9a shows the performance of the IMS in terms of registration throughput at different loads (from 400k to 4M subscribers), while Fig. 9b shows the performance of the IMS in terms of call throughput.

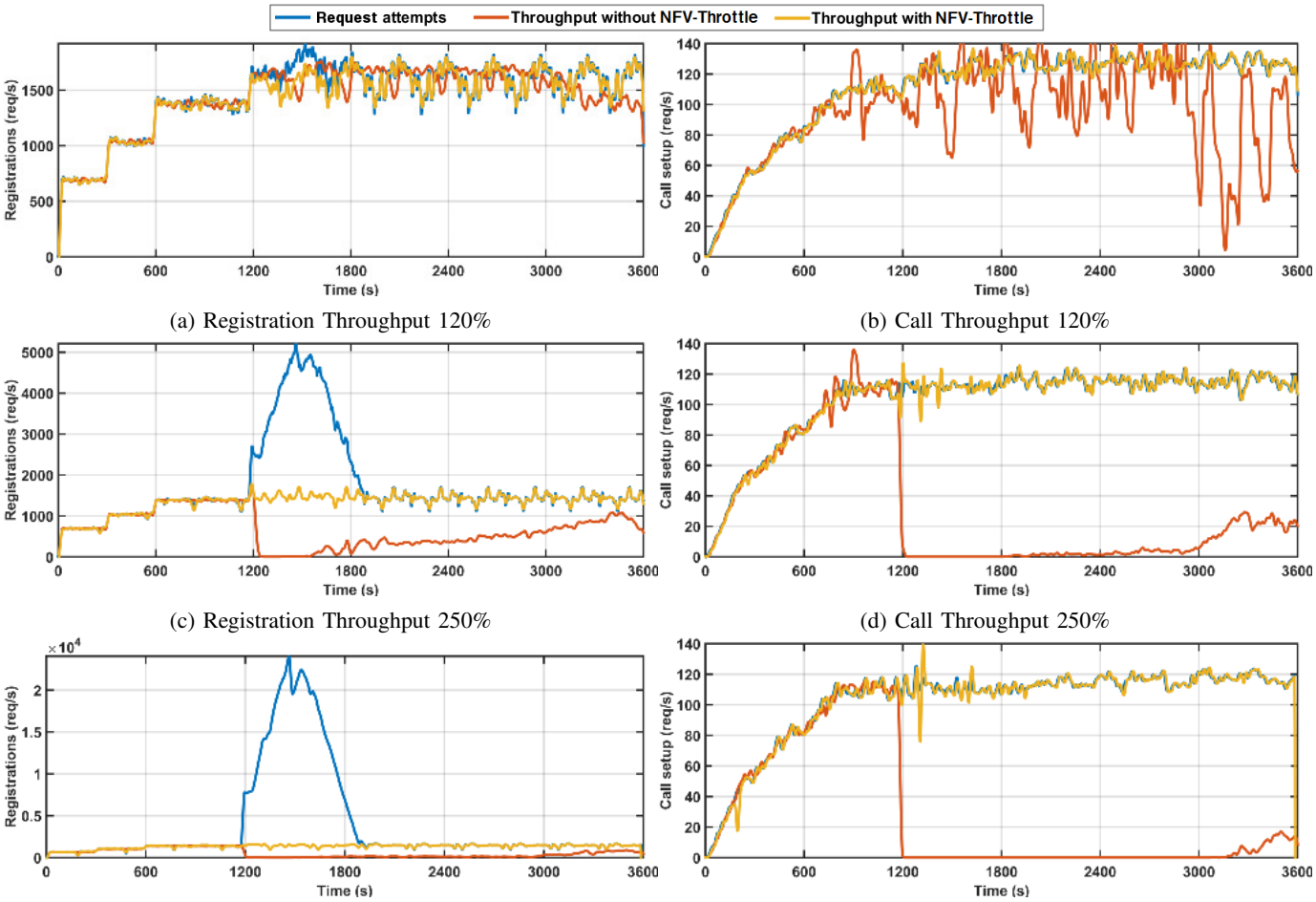
At the engineered level (400k subscribers), there are no significant differences between the three cases and the engineered capacity (TABLE I). In all cases, during the first 20 minutes of the experiments, when the load is within the engineered level, the performance with and without overload control are closely matching. Thus, the overload control does not have negative side effects on the IMS when there is no overload condition.

In overload conditions (more than 400k subscribers), the Host-level overload control provides the best average registration and call throughput compared to VNF-level and Network-level control. The performance gap between VNF-level and Host-level can be explained by observing that the VNF-level control incurs in the overhead of transmitting all of the traffic to VMs, and to discard the traffic in excess in the VM. The Host-level solution acts in the hypervisor rather than the VM, thus avoiding this additional overhead. Thus, when feasible, the Host-level solution should be preferred to the VNF-level one. The Host-level solution can be adopted in the case of NFVIaaS *providers*, which have access to the infrastructure, while it may not be feasible for NFVIaaS *consumers*, which can only deploy VNF-level solutions.

The Network-level overload control also exhibits lower performance than the Host-level one, in particular with respect to the registration throughput. The performance of Network-level overload control is mainly affected by the detection mechanism. The main factor is that detection is distributed and uses a longer sampling period compared to the Host-level solution (which are respectively $30s$ and $10s$), since the Network-level solution needs to collect information from several nodes. Thus, the Network-level solution has a slightly higher detection latency, and it is thus more exposed to oscillations of the workload. Moreover, there are sporadic cases in which the workload is not uniformly balanced across the replicas. Since the Network-level solution detects an overload when a majority of nodes is overloaded, these cases lead to sporadic delays in overload detection.

4) Overhead Evaluation: The mitigation agents act as a lightweight filtering proxy for the network traffic destined to VNFs. Since all the traffic, both TCP and UDP, will be processed by the mitigation agent, we analyze the performance overhead of this critical component. Moreover, since the implementation of the proxy agent is different between the UDP and the TCP transport protocols, we separately analyze both of them.

We performed experiments with the High Overload sce-



(a) Registration Throughput 120%

(b) Call Throughput 120%

(c) Registration Throughput 250%

(d) Call Throughput 250%

(e) Registration Throughput 1000%

(f) Call Throughput 1000%

Fig. 8: Registration and Call Throughput for each overload level (i.e., 120%, 250% and 1000%) at Network Level.

nario (that is, 1000% the engineered level), as discussed in Section IV-C. This scenario is the most stressful among our experiments, and thus represents a worst-case for our overload control framework.

The plot in Figure 10 shows the CPU usage of the *Linux process* that executes the mitigation agent during the experiments, and that tunnels UDP datagrams. The Figure 10 shows the amount of CPU time spent in the kernel within the process (i.e., the *system* time), the time spent in user space (i.e., the *user* time), and the overall time consumption of the process (i.e., the *total* time).

When the workload is within the engineered level of traffic ($900s - 1200s$), the CPU overhead is very little ($\approx 1.5\%$). When the workload reaches ten times the engineered level, the mitigation agents drops the UDP traffic in excess, and its overhead is less than 4%. The memory consumption during the whole experiment is fixed at 3.5MB, since the agent does not allocate any dynamic memory.

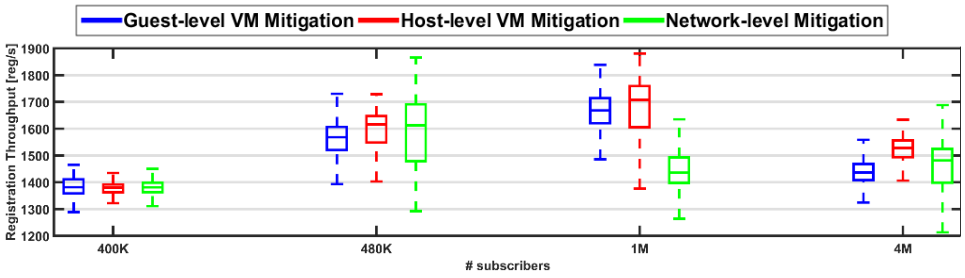
We repeated the same experiment, at maximum level of overload (i.e., 1000% the engineered level), by analyzing the CPU usage of the process under TCP traffic. The plot in Fig. 11 shows the CPU consumption of the mitigation agent during the experiment. At the engineered level of traffic ($900s - 1200s$), the overhead is again very little ($\approx 1.5\%$). At ten times the engineered level, when dropping the TCP traffic in excess, the overhead of the mitigation agent is less than 3%.

The CPU consumption is lower than the UDP case, since the TCP connections with the clients are *persistent*. In fact, the overhead peak in the TCP case happens during the creation of the pool of connections (at $1000s$) and during their shutdown (at $3000s$). It is important to note that, in case of overload, the TCP agent does not reject or close the TCP connections, but it transparently filters application-level messages from the connection stream.

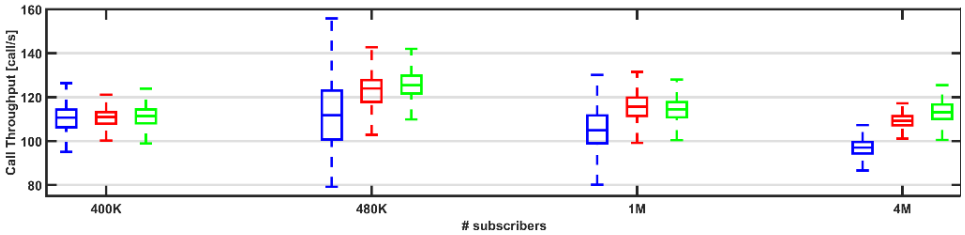
Moreover, in the case of TCP, the memory consumption is dependent of the number of TCP connections that are currently active. During the peak of the traffic, the maximum memory consumption was 16MB, greater than the UDP mitigation agent.

V. Conclusion

In this paper, we proposed a novel framework, *NFV-Throttle*, for overload control in NFV services. This framework has been designed to support the service models of NFV (in particular, NVFIaaS and VNFaaS), by providing a set of *overload detection and mitigation agents* to be deployed either on VMs or on the physical hosts. These agents adopt simple and robust rules to control traffic drop and reject, by analyzing CPU utilization and the network traffic volume.



(a) Registration Throughput



(b) Call Throughput

Fig. 9: Mitigation performance at different operational levels (i.e., node, host and network level)

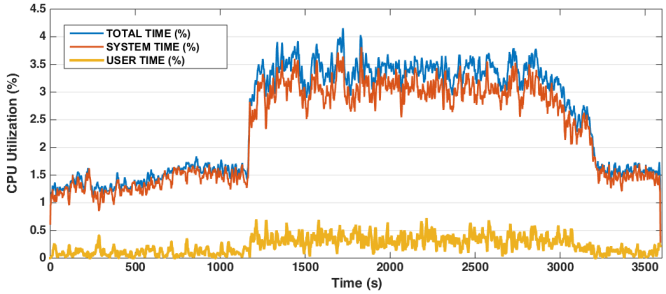


Fig. 10: CPU Consumption of the UDP mitigation proxy

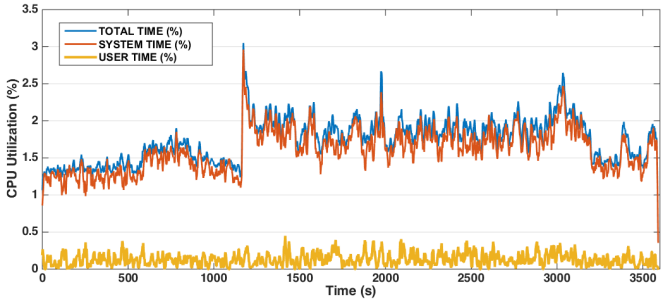


Fig. 11: CPU Consumption of the TCP mitigation proxy

We evaluated the proposed framework in the context of Clearwater, an open-source, NFV-oriented IMS product. In our experiments, we considered stressful overload conditions with high workloads (up to 1000% of the nominal capacity of the system). In all the scenarios, the proposed framework is able to achieve a high throughput, comparable to the maximum throughput under normal conditions, with a negligible memory and CPU overhead. Moreover, the overload control framework avoids failures of the NFV software that are triggered by stress and resource exhaustion. We also analyzed the relative benefits and complementarity of VNF-level, host-level, and network-level overload control. The host-level control achieves the best performance, since it avoids the overhead of forwarding the traffic in excess to the VMs; however, the VNF-level control achieves comparable results, and can be applied in scenarios in which the physical infrastructure cannot be modified; finally, the network-level control allows to reject traffic at the boundaries of the NFV network, thus enabling the network to send notifications to neighbours about overload conditions, in order to gradually reduce the traffic in excess.

REFERENCES

[1] E. McMurry and B. Campbell, "Diameter Overload Control Requirements," RFC 7068, 2013. [Online]. Available: https://tools.ietf.org/html/rfc7068

[2] J. T. Wroclawski, "Specification of the Controlled-Load Network Element Service," RFC 2211, 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc2211.txt

[3] *Network management controls*, ITU-T Std. E-412, 2003.

[4] M. Kind, R. Szabó, C. Meirosu, and F.-J. Westphal, "Softwarization of carrier networks," *Information Technology*, vol. 57, no. 5, pp. 277–284, 2015.

[5] ETSI Industry Specification Group, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action," 2012.

[6] ETSI Industry Specification Group , "NFV Resiliency Requirements," 2015.

[7] Quality Excellence for Suppliers of Telecommunications Forum (QuEST Forum), "TL 9000 Quality Management System Measurements Handbook 4.5," Tech. Rep., 2010.

[8] G. Galante and L. C. E. de Bona, "A Survey on Cloud Computing Elasticity," in *Utility and Cloud Computing (UCC), 2012 IEEE 5th International Conference on*, 2012, pp. 263–270.

[9] P. C. Brebner, "Is your Cloud Elastic Enough?: Performance Modelling the Elasticity of Infrastructure as a Service (IaaS) Cloud Applications," in *Performance Engineering, 3rd ACM/SPEC International Conference on*, 2012, pp. 263–266.

[10] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Scheduling strategies for optimal service deployment across multiple clouds," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1431–1441, 2013.

[11] S. Sotiriadis, N. Bessis, P. Kuonen, and N. Antonopoulos, "The Inter-Cloud Meta-Scheduling (ICMS) Framework," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, 2013, pp. 64–73.

[12] ETSI Industry Specification Group, "NFV Management and Orchestration," 2014.

[13] Project Clearwater. [Online]. Available: http://www.projectclearwater.org/

[14] M. Welsh and D. E. Culler, "Adaptive Overload Control for Busy Internet Servers," in *USENIX Symposium on Internet Technologies and Systems*, 2003.
[15] Clearwater performance and our load monitor. [Online]. Available: http://www.projectclearwater.org/clearwater-performance-and-our-load-monitor/
[16] Tuning overload control for telco-grade performance. [Online]. Available: http://www.projectclearwater.org/overload-control-2/
[17] Project Clearwater bugtracking system. [Online]. Available: https://github.com/Metaswitch/sprout/pulls
[18] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-art and Research Challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.
[19] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "NFV-VITAL: A framework for characterizing the performance of virtual network functions," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015, pp. 93–99.
[20] P. Naik and D. K. Shaw, "NFVPerf: Online Performance Monitoring and Bottleneck Detection for NFV," in *Network Softwarization (NetSoft), 2016 2nd IEEE Conference on*, 2016.
[21] D. Cotroneo, L. De Simone, A. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, "Network Function Virtualization: Challenges and Directions for Reliability Assurance," in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, 2014, pp. 37–42.
[22] D. Cotroneo, L. D. Simone, A. K. Iannillo, A. Lanzaro, and R. Natella, "Dependability evaluation and benchmarking of Network Function Virtualization Infrastructures," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, 2015, pp. 1–9.
[23] C. Sauvanaud, K. Lazri, M. Kaaniche, and K. Kanoun, "Anomaly Detection and Root Cause Localization in Virtual Network Functions," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th Conference on*. IEEE, 2016, pp. 196–206.
[24] T. Niwa, M. Miyazawa, M. Hayashi, and R. Stadler, "Universal fault detection for NFV using SOM-based clustering," in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*. IEEE, 2015, pp. 315–320.
[25] A. Montazerolghaem, M. H. Yaghmaee, A. Leon-Garcia, M. Naghibzadeh, and F. Tashtarian, "A load-balanced call admission controller for ims cloud computing," *Network and Service Management, IEEE Transactions on*, vol. 13, no. 4, pp. 806–822, 2016.
[26] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *Network and Service Management, IEEE Transactions on*, vol. 13, no. 3, pp. 518–532, 2016.
[27] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *Network and Service Management (CNSM), 2014 10th International Conference on*. IEEE, 2014, pp. 418–423.
[28] 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), 2014 IEEE*. IEEE, 2014, pp. 1–9.
[29] B. Németh, J. Czentye, G. Vaszkun, L. Csikor, and B. Sonkoly, "Customizable real-time service graph mapping algorithm in carrier grade networks," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015, pp. 28–30.
[30] F. Carpio, S. Dhahri, and A. Jukan, "Vnf placement with replication for load balancing in nfv networks," *arXiv preprint arXiv:1610.08266*, 2016.
[31] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015, pp. 171–177.
[32] J. Elias, F. Martignon, S. Paris, and J. Wang, "Efficient orchestration mechanisms for congestion mitigation in nfv: Models and algorithms," *IEEE Transactions on Services Computing*, 2015.
[33] P. Vizarreta, M. Condoluci, C. Mahuca, T. Mahmoodi, and W. Kellerer, "Qos-driven function placement reducing expenditures in nfv deployments," in *IEEE ICC*, 2017.
[34] M. Casazza, P. Fouilhoux, M. Bouet, and S. Secci, "Securing virtual network function placement with high availability guarantees," *arXiv preprint arXiv:1701.07993*, 2017.
[35] F. B. Jemaa, G. Pujolle, and M. Pariente, "Qos-aware vnf placement optimization in edge-central carrier cloud architecture," in *Global Communications Conference (GLOBECOM), 2016 IEEE*. IEEE, 2016, pp. 1–7.
[36] T.-M. Pham, T.-T.-L. Nguyen, S. Fdida, and H. T. T. Binh, "Online load balancing for network functions virtualization," *arXiv preprint arXiv:1702.07219*, 2017.
[37] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, and A. Csaszar, "Elastic network functions: opportunities and challenges," *IEEE Network*, vol. 29, no. 3, pp. 15–21, 2015.
[38] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015, pp. 255–260.
[39] S. Van Rossem, W. Tavernier, B. Sonkoly, D. Colle, J. Czentye, M. Pickavet, and P. Demeester, "Deploying elastic routing capability in an SDN/NFV-enabled environment," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, 2015, pp. 22–24.
[40] S. Kasera, J. Pinheiro, C. Loader, M. Karaul, A. Hari, and T. LaPorta, "Fast and robust signaling overload control," in *Network Protocols, 2001. Ninth International Conference on*. IEEE, 2001, pp. 323–331.
[41] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking (ToN)*, vol. 1, no. 4, pp. 397–413, 1993.
[42] ETSI 3GPP, "Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; IP Multimedia Subsystem (IMS); Stage 2," ETSI, Tech. Rep., 2013.
[43] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, 2004.
[44] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
[45] F. E. Goncalves and B.-A. Iancu, *Building Telephony Systems with OpenSIPS*. Packt Publishing Ltd, 2016.
[46] K. Jackson, *OpenStack Cloud Computing Cookbook*. Packt Publishing, 2012.
[47] R. Gayraud, O. Jaques *et al.*, "SIPp: SIP load generator," 2010. [Online]. Available: http://sipp.sourceforge.net/

Domenico Cotroneo (Ph.D.) is associate professor at the Federico II University of Naples. His main interests include software fault injection, dependability assessment, and field-based measurements techniques. He has been member of the steering committee and general chair of the IEEE Intl. Symp. on Software Reliability Engineering (ISSRE), PC co-chair of the 46th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN), and PC member for several other scientific conferences on dependable computing including SRDS, EDCC, PRDC, LADC, and SafeComp.

Roberto Natella (Ph.D.) is a postdoctoral researcher at the Federico II University of Naples, Italy, and co-founder of the Critiware s.r.l. spin-off company. His research interests include dependability benchmarking, software fault injection, and software aging and rejuvenation, and their application in operating systems and virtualization technologies. He has been involved in projects with Finmeccanica, CRITICAL Software, and Huawei Technologies. He contributed, as author and reviewer, to several journals and conferences on dependable computing and software engineering, and he has been in the steering committee of the workshop on software certification (*WoSoCer*) held with recent editions of the ISSRE conference.

Stefano Rosiello received his MSc degree with honors in Computer Engineering in 2015 from the Federico II University of Naples, Italy, working on reliability evaluation for Network Function Virtualization infrastructures. In the same year he joined the Ph.D. course in Information Technology and Electrical Engineering (ITEE) at the same university working within the Dependable Systems and Software Engineering Research Team (DESSERT) group. His main research activity focuses on overload control in carrier-grade network function virtualization and cloud infrastructures. His research interests also include experimental reliability evaluation, dependability benchmarking and fault injection testing.