Ann. Telecommun. (2018) 73:193–204
DOI 10.1007/s12243-017-0597-0

AutoSAC: automatic scaling and admission control of forwarding graphs

Victor Millnert[1] · Enrico Bini[2] · Johan Eker[1,3]

Received: 6 April 2017 / Accepted: 18 July 2017 / Published online: 3 August 2017

Abstract There is a strong industrial drive to use cloud computing technologies and concepts for providing timing sensitive services in the networking domain since it would provide the means to share the physical resources among multiple users and thus increase the elasticity and reduce the costs. In this work, we develop a mathematical model for user-stateless virtual network functions forming a forwarding graph. The model captures uncertainties of the performance of these virtual resources as well as the time-overhead needed to instantiate them. The model is used to derive a service controller for horizontal scaling of the virtual resources as well as an admission controller that guarantees that packets exiting the forwarding graph meet their end-to-end deadline. The Automatic Service and Admission Controller (AutoSAC) developed in this work uses feedback and feedforward making it robust against uncertainties of the underlying infrastructure. Also, it has a fast reaction time to changes in the input.

Keywords Cloud computing · Network function virtualisation · End-to-end deadline · Real-time · Feedback control · Feedforward control

✉ Victor Millnert
victor.millnert@control.lth.se

1 Lund University, Ole Römers väg 1, SE 223 63 Lund, Sweden

2 Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino, Italy

3 Ericsson Research, Lund, Sweden

1 Introduction

Over the last years, cloud computing has swiftly transformed the IT infrastructure landscape, leading to large cost-savings for deployment of a wide range of IT applications. Physical resources such as compute nodes, storage nodes, and network fabrics are shared among tenants through the use of virtual resources. This makes it possible to dynamically change the amount of resources allocated to a tenant, for example as a function of workload or cost. Initially, the cloud technology was mostly used for IT applications, e.g., web servers, databases, etc., but has now found its way into new domains. One of these domains is packets processed by a chain of network functions.

In this work, we are considering a chain of network functions through which packets are flowing. Every packet must be processed by each function in the chain within some specific end-to-end deadline. The goal is to ensure that as many packets as possible meet their deadline, while at the same time using as few resources as possible.

The goal is thus to derive a method for controlling the amount of resources allocated to each network function in the chain. Previously, this was usually done by statically allocating some amount of resources to each network function. Since the input is time-varying (see Fig. 1 for a trace of traffic flowing through a switch in the Swedish university network, SUNET), such a strategy usually lead to over-allocation of resources for long periods of time (yielding high costs and environmental footprint) as well as overload for shorter periods, when the input is large. To ensure that at least some packets meet their deadlines when the network function is overloaded, one has to use *admission control*, i.e., reject some packets.

Recently, a new option became available through the advances of virtualization technology for networking services.

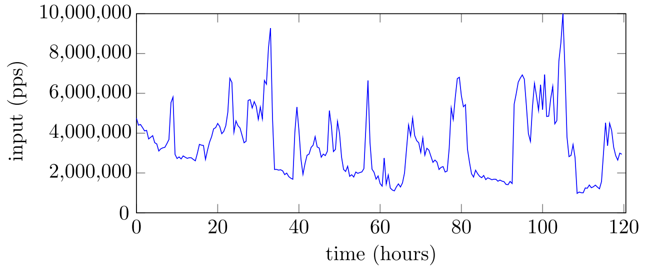

Fig. 1 Traffic flowing through a switch over 120 h. The traffic is normalized to have a peak of 10 million packets per second (pps)

The standardization body ETSI (European Telecommunications Standards Institute) addresses the standardization of these virtual network services under the name Network Functions Virtualization (NFV) [1]. These Virtual Network Functions (VNFs) consist of virtual resources, such as virtual machines (VMs), containers, or even processes running in the OS. Using such VNFs, it is possible to change the resources allocated to a network function by either vertical scaling (i.e., changing the capacity of the allocated VMs) or horizontal scaling (i.e., changing the number of allocated VMs). Horizontal scaling is considered in this work. These VNFs are connected in a graph topology (commonly called a Forwarding Graph), as illustrated in Fig. 2. In this figure, there are two forwarding graphs (corresponding to the blue and red arrows). The blue forwarding graph consists of VNF$_1$, VNF$_2$, VNF$_3$, and VNF$_5$ and the red forwarding graph consists of VNF$_1$, VNF$_2$, VNF$_4$, and VNF$_5$. Each of the VNF is given a number $m_i \in \mathbb{Z}^+$ of VMs, which are mapped onto the network function virtual infrastructure.

While the benefit of using NFV technologies is scalability and resource sharing, there are two drawbacks as follows:

a) *Starting a new virtual resource takes time*, since it has to be deployed to a physical server and it requires the execution of several initialization scripts and push/pulls before it is ready to serve packets,

b) *The true performance of the virtual resource differs from the expected performance*, since one does not know what else is running on the physical machines [2].

In this work, we

– develop a model of a service-chain of network functions and use it to derive a service-controller and admission-controller for the network functions,
– derive a service-controller controlling the number of virtual resources (e.g., VMs or containers) allocated to each network function by using feedback from the true performance of the instances as well as feedforward between the network functions,
– derive an admission-controller that is aware of the actions of the service-controller which it uses in order to reject as few packets as possible,
– evaluate the service and admission controller using a real-world traffic trace from the Swedish University Network (SUNET).

1.1 Related works

There are a number of works considering the problem of controlling virtual resources within data centers, and specifically for virtual network functions. However, many of them focus on orchestration, i.e., how the virtual resources should be mapped onto the physical hardware. Shen et al. [3] develop a management framework, vConductor, for realizing end-to-end virtual network services. In [4], Moens and De Turk develop a formal model for resource allocation of virtual network functions. A slightly different approach is taken by Mehraghdam et al. [5] where they define a model for formalizing the chaining of forwarding graphs using a context-free language. They solve the mapping of

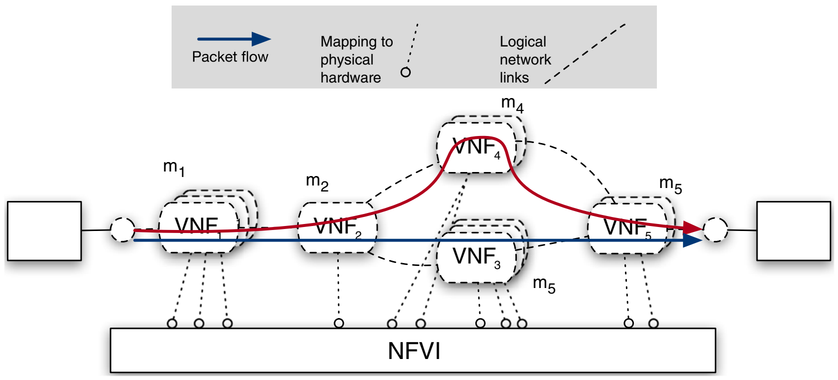

Fig. 2 Several virtual networking functions (*VNF*) are connected together to provide a set of services. A packet flow is a specific path through the VNFs. Connected VNFs are referred to as virtual forwarding graphs or service chains. The VNFs are mapped onto physical hardware, i.e., compute nodes and network fabrics and this underlying hardware infrastructure is referred to as Network Function Virtualization Infrastructure (*NFVI*), which is the physical servers and the communication fabric connecting them

the forwarding graphs onto the hardware by posing it as a MIQCP.

Scaling of virtual network functions is however studied by Mao et al. [6] where they develop a mechanism for auto-scaling resources in order to meet some user specified performance goal. Recently, Wang et al. [7] developed a fast online algorithm for scaling and provisioning VNFs in a data center. However, they are not considering timing-sensitive applications with deadlines for the packets moving through the chain, which is done by Li et al. [8] where they present a design and implementation of NFV-RT that aims at controlling NFVs with soft real-time guarantees, allowing packets to have deadlines.

The enforcement of an end-to-end deadline for a sequence of jobs is however addressed by several works, possibly under different terminologies. Di Natale and Stankovic [9] propose to split the E2E deadline proportionally to the local computation time or to divide equally the slack time. Later, Jiang [10] used time slices to decouple the schedulability analysis of each node, reducing the complexity of the analysis. Such an approach improves the robustness of the schedule, and allows to analyze each pipeline in isolation. Serreli et al. [11, 12] proposed to assign local deadlines to minimize a linear upper bound of the resulting local demand bound functions. More recently, Hong et al. [13] formulated the local deadline assignment problem as a MILP with the goal of maximizing the slack time.

An alternate analysis was proposed by Jayachandran and Abdelzaher [14], who developed several transformations to reduce the analysis of a distributed system to the single processor case. Or in [15] where Henriksson et al. proposed a feedforward/feedback controller to adjust the processing speed to match a given delay target.

2 Modeling the service-chain

In this section, we present a general model of the forwarding graph and virtual network functions presented in Section 1. We consider a *service-chain* consisting of *n functions* $F_1, \ldots, F_n$, as illustrated in Fig. 3. Packets are flowing through the service-chain and they must be processed by each function in the chain within some end-to-end deadline. A *fluid model* is used to approximate the packet flow and at time t there are $r_i(t) \in \mathbb{R}^+$ packets per second (pps) entering the ith function. In a recent benchmarking study, it was shown that a typical virtual machine can process around 0.1–2.8 million packets per second, [16]. Hence, in this work, the number of packets flowing through the functions is assumed to be in the order of millions of packets per second, supporting the use of a fluid model.

A function consists of several parts, as illustrated in Fig. 4: an *admission controller*, a *service controller*, $m_i(t)$ *instances*, a *buffer*, and a *load balancer*. It is assumed that all the parts of a function are located at the same location, e.g., the same data center or rack. In [17], Google showed that less than 1 µs of the latency in a data center was due to the propagation in the network fabric. Hence, communication delay within a function is neglected.

2.1 Admission controller

Every packet that enters the service-chain must be processed by all of the functions in the chain within a certain *end-to-end (E2E) deadline*, denoted $D^{\max}$. This deadline can be split into *local deadlines* $D_i(t)$, one for each function in the chain, such that the packet should not spend more than $D_i(t)$ time-units in the ith function. Should a packet miss its E2E deadline, it is considered useless. It is thus favorable to use admission control to drop packets that have a high probability of missing their deadline in order to make room for following packets. The goal of the admission controller is to guarantee that the packets that make it through the service-chain do meet their E2E deadline. It is assumed to be possible to do admission control at the entry of every function in the chain.

Packets are admitted into the buffer of function F_i based on the *admittance flag* $\alpha_i(t) \in \{0, 1\}$. If $\alpha_i(t) = 1$ incoming packets are admitted into the buffer, and if $\alpha_i(t) = 0$ they are rejected. We define the *residual rate* $\rho_i(t)$ to be the rate by which packets are admitted into the buffer:

$$\rho_i(t) = r_i(t) \times \alpha_i(t). \tag{1}$$

2.2 Service controller

At any time instance, function F_i has $m_i(t) \in \mathbb{Z}^+$ *instances* up and running. Each instance is capable of processing

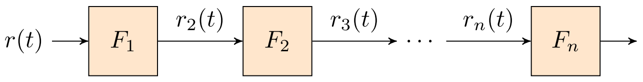
$$\begin{tikzcd} r(t) \arrow[r] & \boxed{F_1} \arrow[r, "r_2(t)"] & \boxed{F_2} \arrow[r, "r_3(t)"] & \cdots \arrow[r, "r_n(t)"] & \boxed{F_n} \arrow[r] & {} \end{tikzcd}$$

Fig. 3 Illustration of the service-chain

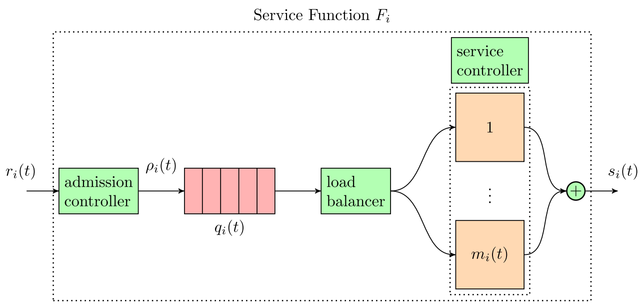

Fig. 4 Illustration of the structure and different entities of the function

packets and corresponds to a virtual machine, a container, or a process running in the OS. It is possible to control the number of running instances by sending a *reference signal* $m_i^{\text{ref}}(t) \in \mathbb{Z}^+$ to the service controller. However, as explained in Section 1, it takes some time to start/stop instances since an instantiation of the service is always needed. We denote this as the *time overhead* Δ_i. Hence, the number of instances running in the i'th function at time t is

$$m_i(t) = m_i^{\text{ref}}(t - \Delta_i). \tag{2}$$

The time-overhead is assumed to be symmetric here, but in the real-world it is usually faster to start an instance than it is to stop one. However, for increased readability they are considered equal in this work. It should be noted that it is straight forward to extend the theory to account for an asymmetric time-overhead.

An instance is expected to be able to process packets at an *expected service rate* of $\bar{s}_i$ pps. However, as described in Section 1, the true capacity of the instance will differ from the expected one since there might be other loads running on the infrastructure (i.e., the *physical machine*). Hence, the *true capacity* of the jth instance in the ith function is given by

$$s_{i,j}^{\text{cap}}(t) = \bar{s}_i + \xi_{i,j}(t),$$

where $\xi_{i,j}(t)$ is the *machine uncertainty* for the jth instance in the ith function. It is given by

$$\xi_{i,j}(t) \in [\xi_i^{\text{lb}}, \xi_i^{\text{ub}}] \text{ pps}, \quad -\bar{s}_i < \xi_i^{\text{lb}} \le \xi_i^{\text{ub}} < \infty,$$

where ξ_i^{lb} and ξ_i^{ub} are lower and upper bounds of this machine uncertainty, assumed to be known. The machine uncertainty is also assumed to be fairly constant during the lifetime of the instance. Using this, one can express the true capacity of the ith function in the service-chain as

$$s_i^{\text{cap}}(t) = \sum_{j=1}^{m_i(t)} \bar{s}_i + \xi_{i,j}(t), \tag{3}$$

which together with the *average machine uncertainty*

$$\hat{\xi}_i(t) = \frac{1}{m_i(t)} \sum_{j=1}^{m_i(t)} \xi_{i,j}(t), \tag{4}$$

can be written as $s_i^{\text{cap}}(t) = m_i(t) \times (\bar{s}_i + \hat{\xi}_i(t))$. Note that it would be natural to allow the time-overhead Δ_i to also have some uncertainty. However, such uncertainty can be translated into a machine uncertainty.

2.3 Processing of packets

The packets in the buffer are stored and processed in a FIFO manner. Once a packet reaches the head of the queue the load balancer will distribute it to one of the instances in the function. Note that this is done continuously due to the fluid approximation. The rate by which the load balancer is distributing packets, and thus by which the function is processing packets, is defined as the *service rate*

$$s_i(t) = \begin{cases} \rho_i(t) & \text{if } q_i(t) = 0 \text{ and } \rho_i(t) \le s_i^{\text{cap}}(t) \\ s_i^{\text{cap}}(t) & \text{else} \end{cases} \tag{5}$$

where $\rho_i(t)$ is residual rate given by Eq. 1 and $q_i(t)$ is the number of packets in the buffer:

$$q_i(t) = P_i(t) - S_i(t), \quad q_i(t) \in \mathbb{R}^+, \tag{6}$$

where $P_i(t) = \int_0^t \rho_i(x)\mathrm{d}x$ is the total amount of packets that has been admitted into function F_i, and $S_i(t) = \int_0^t s_i(x)\mathrm{d}x$ is the total amount of packets that has been served by function F_i. Furthermore, the total amount of packets that has reached the ith function is given by $R_i(t) = \int_0^t r_i(x)\mathrm{d}x$.

2.4 Function delay

The time that a packet that exits function F_i at time t has spent inside that function is denoted the *function delay* $d_i(t)$:

$$d_i(t) = \inf\{\tau \ge 0 : \; P_i(t - \tau) \le S_i(t)\}. \tag{7}$$

The expected time that a packet entering the ith function at time t will spend in the function before exiting is defined as the *expected function-delay* $\bar{d}_i(t)$

$$\bar{d}_i(t) = \inf\Big\{ + \hat{\xi}_i(x))\mathrm{d}x\tau \ge 0 : \; P_i(t) \le S_i(t) + \int_t^{t+\tau} m_i(x) \times (\bar{s}_i + \hat{\xi}_i(x))\mathrm{d}x \Big\}. \tag{8}$$

Equation 8 can be interpreted as finding the minimum time $\tau \ge 0$ such that $S_i(t + \tau) = P_i(t)$, or in other words such that at time $t + \tau$ the function will have processed all the packets that have entered the function at time t.

Computing the expected function-delay $\bar{d}_i(t)$ requires information about $m_i(t)$ and $\hat{\xi}_i(t)$ for the future, whereas computing the expected function delay $d_i^{\text{ub}}(t)$ requires information about $m_i(t)$ for the future. Information about $m_i(t)$ up until time $t + \Delta_i$ is always known since $m_i(t + \Delta_i) = m_i^{ref}(t)$ and $m_i^{ref}(x)$ is known for $x \in [0, t]$. It is therefore possible to compute the expected function delay $\bar{d}_i(t)$ whenever it is shorter than the time-overhead Δ_i (which will be used later in Section 3 when deriving the admission controller and the service controller).

Note that the (expected) function delay does not distinguish between queueing delay and processing delay. In [17], Google profiled where the latency in a data center occurred and showed that 99% of the latency (≈85 µs) occurred somewhere in the kernel, the switches, the memory, or the application. It is very difficult to say exactly which of this 99% is due to processing or queueing, hence they are considered together as the function delay.

2.5 Concatenation of functions

The n functions in the service-chain are concatenated with the assumption of no loss of packets in the communication channel between them. Therefore, the input of function F_i is exactly the output of function F_{i-1}:

$$r_i(t) = s_{i-1}(t), \qquad \forall i = 2, 3, \ldots, n.$$

Finally, no communication latency between the functions is assumed. However, it is possible to account for it, and would be necessary should the different functions reside in different locations, i.e. different data centers. However, adding a communication latency is straightforward, and if such a communication latency (say C) were to be constant between the functions one could easily account for it by properly decrementing the end-to-end deadline: $\tilde{D}^{\max} = D^{\max} - C$, and then use the framework developed in this paper.

2.6 Problem formulation

The goal of this paper is to derive a service-controller and an admission-controller that guarantees that packets that pass through the service-chain meet their E2E deadline. This should be done using as few resources as possible while still achieving as high throughput as possible. This is captured in a simple, yet intuitive *utility function* $u_i(t)$. Later in Section 3, the utility function is used to derive an automatic service- and admission controller, denoted AutoSAC.

Utility function The utility function measures the *availability* $a_i(t)$ and the *efficiency* $e_i(t)$ of each function in the service chain. The availability is defined as the ratio between the service-rate and the input-rate of the function, and the efficiency is defined as the ratio between service-rate and the capacity of the function:

$$a_i(t) = \frac{\text{service}}{\text{demand}} = \frac{s_i(t)}{r_i(t - d_i(t))} \in [0,\ 1+\epsilon], \tag{9}$$

$$e_i(t) = \frac{\text{service}}{\text{capacity}} = \frac{s_i(t)}{s_i^{\text{cap}}(t)} \in [0,\ 1]. \tag{10}$$

The reason why $a_i(t)$ can grow greater than 1 is due to the buffer—it is possible to store packets for a short interval and then process them at a rate greater than what they arrived with. However, it is not possible to have $a_i(t) > 1$ for an infinite amount of time. In practice, ϵ is very small, and it is not possible to achieve a $a_i(t) > 1$ for any significant period of time.

A low availability corresponds to a large percentage of the incoming load being rejected by the admission controller, since there is not enough capacity to serve them. A low efficiency, on the other hand, corresponds to over-provisioning of resources. It is therefore difficult to achieve both high availability and high efficiency. The availability and efficiency is combined into a utility function $u_i(t)$:

$$u_i(t) = a_i(t) \times e_i(t) = \frac{s_i^2(t)}{s_i^{\text{cap}}(t) \times r_i(t - d_i(t))}. \tag{11}$$

Note that the utility function as well as the availability and efficiency function have the good property of being normalized making it easy to compare the performance of service-chains having different input load. To evaluate the performance between service-chains of different lengths and over different time-horizons the *average utility* $U(t)$ is defined:

$$u(t) = \frac{1}{n}\sum_{i=1}^{n} u_i(t), \qquad U(t) = \frac{1}{t}\int_0^t u(x)\mathrm{d}x. \tag{12}$$

While the utility function (11) uses the product of the availability and efficiency one might argue that they should not have equal weight when computing the utility. A natural choice to achieve that would be to have a convex combination of them:

$$\tilde{u}_i(t) = \lambda_i a_i(t) + (1-\lambda_i)e_i(t), \qquad \lambda_i \in [0, 1], \tag{13}$$

where λ_i corresponding to the relative importance of achieving a high availability or a high efficiency. The method used in Section 3 to derive a control-strategy using utility function (11) will also apply to the alternative utility function (13).

3 Controller design

In this section, an automatic service- and admission-controller (AutoSAC) is derived. Figure 5 illustrates an overview of the different parts of AutoSAC and the information flow it uses. It measures the incoming load, current queue size, and the true performance in order to estimate how much service rate it will need as well as to estimate how long it will take an incoming packet to pass through the function. It also uses feedforward to functions down the chain in order to make them react faster to changes in the input load. For instance, when the ith function increases its service rate, it sends a signal to the $i+1$th function letting it know that in Δ_i time-units, it will get an increase in incoming traffic rate. Finally, due to the time overhead needed to start new instances there will be a need to do admission control, however, in order to not discard unnecessarily many packets it uses feedback from the queue size and the true performance of the functions to estimate how much time it will take a new packet to pass through the function, then it does the admission control based on this estimate.

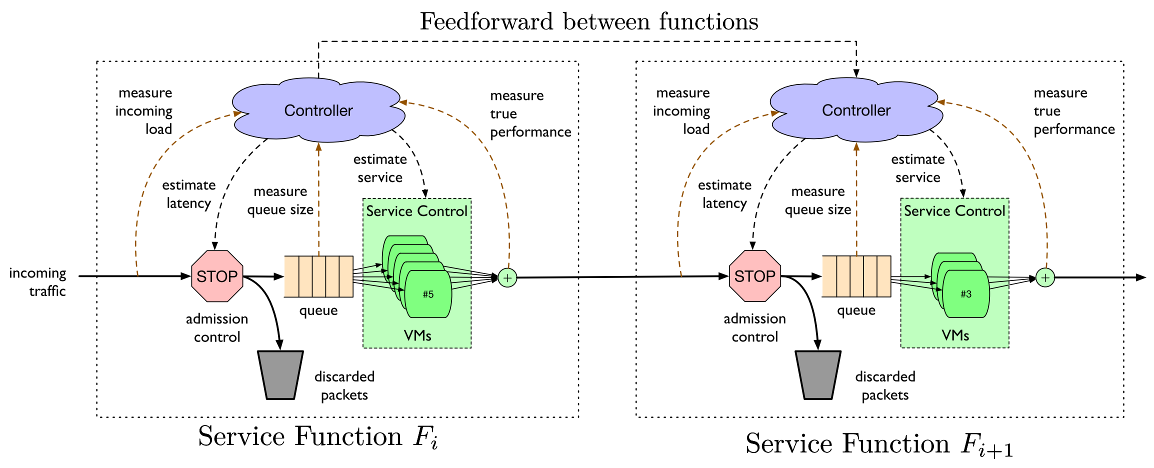

Fig. 5 Overview of the automatic service- and admission-controller (AutoSAC) highlighting that it uses feedback from the true performance of the functions to estimate the time it will take incoming packets to pass through the function. It also utilizes feedforward between the functions to ensure faster reactions to changes in the incoming traffic load

The difficulty when deriving AutoSAC lies in the different time-scales for starting/stopping instances, the E2E deadlines, and the rate-of-change of the input. They are all assumed to be of different orders of magnitudes, given by Table 1. However, these timing assumptions will be exploited when deriving AutoSAC later.

The admission controller is derived in Section 3.1 and the service controller in Section 3.3. In Section 3.4, a short discussion of the properties of AutoSAC is presented.

3.1 Admission controller

Every request that enters the service chain has an *end-to-end deadline* $D^{\max}$. It has to pass through every function in the chain within this time. Furthermore, each function can impose a *local deadline* $D_i(t)$ for the packet entering the ith function at time t. One can therefore use either the local deadline to do a *decentralized* admission control at the entry of each of the functions in the chain, or the global deadline for a *centralized* admission control. In this work, we will use a decentralized approach, shown below, but will also derive a policy for a centralized admission control in Section 3.2.1; however, only the decentralized policy will be evaluated in Section 4.

Table 1 Timing assumptions for the end-to-end deadline, the change-of-rate of the input, and the overhead for changing the service-rate. These timing assumptions are used when deriving the automatic service- and admission-controller

Parameter	Timing assumption
Long-term trend change of the input	1 min–1 h
Service-rate change overhead Δ_i	1 s–1 min
Request end-to-end deadline $D^{\max}$	1 µs–100 ms

3.2 Decentralized admission control

For the decentralized admission control, each function can compare the local deadline with the upper bound of the expected delay $d_i^{\text{ub}}(t)$ it will take a new packet to pass through the function. If the worst-case expected delay is larger than the local deadline the admission controller should drop the packet. This results in the following policy for the admittance-flag $\alpha_i(t)$:

$$\alpha_i(t) = \begin{cases} 1 & \text{if } D_i(t) \geq d_i^{\text{ub}}(t) \\ 0 & \text{if } D_i(t) < d_i^{\text{ub}}(t) \end{cases} \tag{14}$$

where the upper bound on the expected function delay $d_i^{\text{ub}}(t)$ is given by

$$d_i^{\text{ub}}(t) = \inf\Big\{ + \xi_i^{\text{lb}})\text{d}x\tau \geq 0 : \; P_i(t) \leq S_i(t) + \int_t^{t+\tau} m_i(x) \times (\bar{s}_i + \xi_i^{\text{lb}})\text{d}x \Big\}. \tag{15}$$

This is the worst case of the expected delay given by Eq. 8, i.e., when every instance is processing packets at the lower bound of its possible service-rate, hence leading to the upper bound on the expected delay. One should note here that in order to compute the upper bound (15) one need information about $m_i(x)$ for $x \in [t, t+\tau]$. However, as mentioned earlier, as long as $\tau \leq \Delta_i$ this information is available since the number of instances running at time t is decided by the control signal computed Δ_i time units ago, i.e. $m_i(t) = m_i^{ref}(t - \Delta_i)$, implying that $m_i(x)$ is known for $x \in [0, t + \Delta_i]$. This is illustrated in Fig. 6 where $P_i(t)$ shows the cumulative amount of packets that has been let into the function, and $S_i(t)$ the cumulative amount of served packets up until time t. From time t until $t + \Delta_i$, it shows a shaded blue region, highlighting that the exact service is uncertain in this area. However, it is possible to compute an

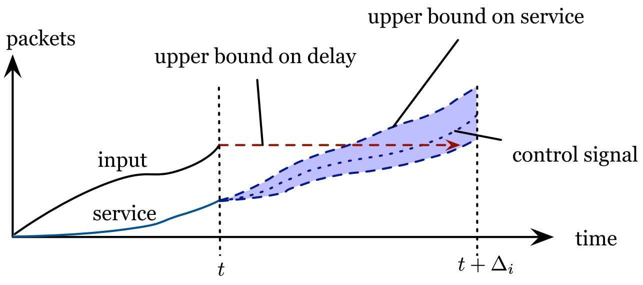

Fig. 6 Illustration of how the upper bound of the expected function delay is found. $P_i(t)$ is the cumulative amount of packets that has been admitted into the function, $S_i(t)$ is the cumulative amount of served packets. The *shaded blue region* illustrates the reachable space of $S_i(t)$ for the time up till $t+\Delta_i$. It is upper bounded by $S_i^{\text{ub}}(t)$ and lower bounded by $S_i^{\text{lb}}(t)$. The lower bound is then used to compute the upper bound on the expected function delay $d_i^{\text{ub}}(t)$.

upper bound $S_i^{\text{ub}}(t) = S_i(t) + \int_t^{t+\tau} m_i(x) \times (\bar{s}_i + \xi_i^{\text{ub}})dx$ and a lower bound $S_i^{\text{lb}}(t) = S_i(t) + \int_t^{t+\tau} m_i(x) \times (\bar{s}_i + \xi_i^{\text{lb}})dx$ of this uncertainty region and hence a lower bound and an upper bound on the expected delay.

3.2.1 Centralized admission control

In contrast to the decentralized admission control, it might be advantageous to drop packets as soon as possible (in order to not waste any resources on packets that are dropped later) in the service chain if there is a possibility that they will miss their global deadline. To do so, one has to compare the *expected worst-case end-to-end delay* $D^{\text{ub}}(t)$ for a packet entering the chain at time t with the global deadline $D^{\max}(t)$, leading to the following policy:

$$\alpha_1(t) = \begin{cases} 1 & \text{if } D^{\max} \geq D^{\text{ub}}(t) \\ 0 & \text{if } D^{\max} < D^{\text{ub}}(t) \end{cases}. \tag{16}$$

Computing $D^{\text{ub}}(t)$ in Eq. 16 is straightforward, but before doing so, one has to compute the *worst-case service rates* for all the functions down the chain. At any time $x \geq t$ (with t being the current time) the *worst-case predicted service-rate* for functions $i = 1, 2, \ldots, n$ is:

$$s_i^{\text{lb}}(x) = \begin{cases} s_{i-1}^{\text{lb}}(x) & \text{if } q_i(x) = 0 \text{ and } s_{i-1}^{\text{lb}}(x) \leq m_i(x) \times (\bar{s}_i + \xi_i^{\text{lb}}), \\ m_i(x) \times (\bar{s}_i + \xi_i^{\text{lb}}) & \text{else.} \end{cases} \tag{17}$$

where $s_0^{\text{lb}}(x) = 0$, since we cannot predict the future input-rate of the first function. With t being the current time, the *worst-case predicted cumulative-service* of function i at time $x \geq t$ is then given by:

$$S_i^{\text{lb}}(t, x) = S_i(t) + \int_t^{x+t} s_i^{\text{lb}}(z)\text{d}z, \qquad i = 1, 2, \ldots, n \tag{18}$$

Using this, the *expected worst-case end-to-end delay* $D^{\text{ub}}(t)$ is given by

$$D^{\text{ub}}(t) = \inf\{\tau \geq 0 : \; P_1(t) \leq S_n^{\text{lb}}(t+\tau)\}, \tag{19}$$

where $P_1(t) = \int_0^t \rho_1(x)dx$ is the cumulative amount of requests that has been admitted into the first function. One should note that $S_n^{\text{lb}}(x)$ in Eq. 19 could be expressed in a very neat way using Network Calculus [18, 19], but due to lack of space we decided to not introduce the theory of Network Calculus in this paper.

3.3 Service controller

The goal for the service-controller is to find $m_i^{ref}(t)$ such that the utility function is maximized once the reference signal is realized in Δ_i time-units, i.e., such that $u_i(t+\Delta_i)$ is maximized. In this section, it will be assumed that the utility function used is the one defined in Eq. 11; later in Section 3.3.1, it will be derived for the alternative utility function (13). Recall that the utility function (11) is given by

$$u_i(t) = a_i(t) \times e_i(t) = \frac{s_i^2(t)}{s_i^{\text{cap}}(t) \times r_i(t - d_i(t))}.$$

As explained in the introduction of this section, the input load is assumed to change relatively slowly over a time interval of a few milliseconds. Hence, one can approximate

$$r_i(t - d_i(t)) \approx r_i(t), \tag{20}$$

since the goal of both the admission controller and the service controller is to keep $d_i(t)$ in the order of milliseconds or less. Therefore, it is possible to approximate the utility function with

$$u_i(t) \approx \frac{s_i^2(t)}{s_i^{\text{cap}}(t) \times r_i(t)}.$$

Furthermore, the service rate $s_i(t)$ can be approximated to be either at the capacity of the function, $s_i^{\text{cap}}(t)$, or at the input rate $r_i(t)$

$$s_i(t) \approx \min\{s_i^{\text{cap}}(t), \; r_i(t)\}. \tag{21}$$

where the min is used since the function cannot process packets at a faster rate than what they are entering the function for a prolonged period of time. Likewise, it cannot process packets at a rate higher than the capacity of the function when the input were to be higher than this. This leads to the utility function being approximated as

$$u_i(t) \approx \begin{cases} \dfrac{(s_i^{\mathrm{cap}}(t))^2}{s_i^{\mathrm{cap}}(t) \times r_i(t)} = \dfrac{s_i^{\mathrm{cap}}(t)}{r_i(t)}, & \text{if } s_i^{\mathrm{cap}}(t) \le r_i(t) \\ \dfrac{r_i^2(t)}{s_i^{\mathrm{cap}}(t) \times r_i(t)} = \dfrac{r_i(t)}{s_i^{\mathrm{cap}}(t)}, & \text{else} \end{cases}$$

With $s_i^{\mathrm{cap}}(t)$ given by Eq. 3 and the average machine uncertainty $\hat{\xi}_i(t)$ given by Eq. 4 the utility function can finally be approximated as

$$u_i(t) \approx \begin{cases} \dfrac{m_i(t)(\bar{s}_i + \hat{\xi}_i(t))}{r_i(t)}, & \text{if } m_i(t)(\bar{s}_i + \hat{\xi}_i(t)) \le r_i(t) \\ \dfrac{r_i(t)}{m_i(t)(\bar{s}_i + \hat{\xi}_i(t))}, & \text{else} \end{cases} \tag{22}$$

Since the goal is to find $m_i^{ref}(t)$ in order to maximize $u_i(t+\Delta_i)$, one needs knowledge of $\hat{\xi}_i(t+\Delta_i)$ and $r_i(t+\Delta_i)$ which is not available. However, one can assume that the machine uncertainty will be fairly constant during Δ_i time-units such that $\hat{\xi}_i(t+\Delta_i) \approx \hat{\xi}_i(t)$. Furthermore, one has to estimate the future input-rate to the function. For the first function, F_1, this can be done by using the derivative of the (preferably low-pass filtered) input-rate:

$$\hat{r}_1(t) = r_1(t) + \Delta_1 \frac{\mathrm{d}r_1(t)}{\mathrm{d}t}.$$

For the succeeding functions, $i = 2, \ldots, n$, the input-rate will change in a step-wise fashion and can therefore not approximate it with the expression above. However, since $r_i(t) = s_{i-1}(t)$ and $m_{i-1}(x)$ is known for $x \in [0,\ t+\Delta_{i-1}]$ (with t being the current time), one could estimate the future input-rate $\hat{r}_i(t)$ with

$$\hat{r}_i(t) \approx \min\left(s_{i-1}^{\mathrm{cap}}(t+\Delta_{i-1}),\ \hat{r}_{i-1}(t)\right), \quad i = 2, \ldots, n.$$

Note that $s_{i-1}^{\mathrm{cap}}(t+\Delta_{i-1})$ is used here, instead of $s_{i-1}^{\mathrm{cap}}(t+\Delta_i)$. The reason is that if $\Delta_i > \Delta_{i-1}$ one does not have enough information to compute $s_{i-1}^{\mathrm{cap}}(t+\Delta_{i-1})$. However, one can use the assumption that $\Delta_i \approx \Delta_{i-1}$. Furthermore, since

$$s_{i-1}^{\mathrm{cap}}(t+\Delta_{i-1}) \approx m_{i-1}^{\mathrm{ref}}(t) \times (\bar{s}_{i-1} + \hat{\xi}_{i-1}(t))$$

one can summarize the predicted input $\hat{r}_i(t)$ as

$$\hat{r}_i(t) = \begin{cases} r_i(t) + \Delta_i \frac{\mathrm{d}r_i(t)}{\mathrm{d}t}, & i = 1 \\ \min\left\{m_{i-1}^{\mathrm{ref}}(t) \times (\bar{s}_{i-1} + \hat{\xi}_{i-1}(t)),\ \hat{r}_{i-1}(t)\right\}, & \text{else} \end{cases} \tag{23}$$

With this, one can define $\kappa_i(t) \in \mathbb{R}^+$ to be the real number of instances needed to exactly match the predicted incoming rate:

$$\kappa_i(t) = \frac{\hat{r}_i(t)}{\bar{s}_i + \hat{\xi}_i(t)}. \tag{24}$$

The control signal, i.e., the number of instances that should be started, $m_i^{ref}(t)$ can then be found by solving

$$m_i^{ref}(t) = \begin{cases} \arg\max\limits_{x \in \mathbb{Z}^+} \{x/\kappa_i(t)\}, & \text{if } x \le \kappa_i(t) \\ \arg\max\limits_{x \in \mathbb{Z}^+} \{\kappa_i(t)/x\}, & \text{else} \end{cases}$$

where $x \in \mathbb{Z}^+$ is the number of instances and $\kappa_i(t)$ given by Eq. 24. Here, one can see that the first case of the above equation is maximized when x is as large as possible, but since this case is only valid when $x \le \kappa_i(t)$ it leads to $x = \lfloor \kappa_i(t) \rfloor$. Similarly, the second case is maximized when x is as small as possible, and since this case is valid for $x \ge \kappa_i(t)$ it leads to $x = \lceil \kappa_i(t) \rceil$, leading to the final control-law:

$$m_i^{ref}(t) = \begin{cases} \lfloor \kappa_i(t) \rfloor, & \text{if } \lfloor \kappa_i(t) \rfloor \lceil \kappa_i(t) \rceil \ge \kappa_i^2(t) \\ \lceil \kappa_i(t) \rceil, & \text{else} \end{cases} \tag{25}$$

where again $\kappa_i(t) = \frac{\hat{r}_i(t)}{\bar{s}_i + \hat{\xi}_i(t)}$ is the real number of machines that is necessary to exactly match the predicted incoming traffic.

3.3.1 Alternative utility function

Using the same method described in Section 3.3, one can derive a control-law for the alternative utility function (13):

$$\begin{aligned} \tilde{u}_i(t) &= \lambda_i a_i(t) + (1-\lambda_i) e_i(t) \\ &= \lambda_i \frac{s_i(t)}{r_i(t - d_i(t))} + (1-\lambda_i) \frac{s_i(t)}{s_i^{\mathrm{cap}}(t)}. \end{aligned}$$

By using the approximation (20) for the input rate, (21) for the service rate, (23) for predicting the input rate, and finally (3) for estimating the maximum capacity along with Eq. 4 for the machine uncertainty, one arrives at the following control-law:

$$m_i^{ref}(t) = \begin{cases} \arg\max\limits_{x \in \mathbb{Z}^+} \left\{\lambda_i \dfrac{x}{\kappa_i(t)} + (1-\lambda_i)\right\}, & \text{if } x \le \kappa_i(t) \\ \arg\max\limits_{x \in \mathbb{Z}^+} \left\{\lambda_i + (1-\lambda_i) \times \dfrac{\kappa_i(t)}{x}\right\}, & \text{else} \end{cases}$$

where $\kappa_i(t) = \frac{\hat{r}_i(t)}{\bar{s}_i + \hat{\xi}_i(t)}$.

One can see that the upper case is maximized when x is as large as possible within that case, i.e., with $x = \lfloor \kappa_i(t) \rfloor$, while the lower case is maximized when x is as small as possible, i.e., with $x = \lceil \kappa_i(t) \rceil$. The remaining question is then which of the two cases that yield the largest utility. Fortunately, this it is easy to evaluate, resulting in the final control-law for the alternative utility function:

$$m_i^{ref}(t) = \begin{cases} \lfloor \kappa_i(t) \rfloor, \text{ if } \lambda_i \lfloor \kappa_i(t) \rfloor \lceil \kappa_i(t) \rceil + (1-2\lambda_i)\kappa_i(t) \lceil \kappa_i(t) \rceil \geq (1-\lambda_i)\kappa_i^2(t) \\ \lceil \kappa_i(t) \rceil, \text{ else} \end{cases} \tag{26}$$

where again $\kappa_i(t) = \frac{\hat{r}_i(t)}{\bar{s}_i + \hat{\xi}_i(t)}$. Comparing the two control-laws (25) and Eq. 26, one can see that the alternative control-law (26) is equivalent to the Eq. 25 when the efficiency and availability are considered equally important, i.e., when $\lambda_i = 1/2$.

3.4 Properties of AutoSAC

There are several interesting properties captured by the admission controller and service controller presented in this section. First of all, the admission controller (14) ensures, by design, that every packet that is admitted into a function, and thus exits the function, meets its deadline. Therefore, no packets that exit the service-chain will miss their end-to-end deadline.

The service-controller given by Eq. 25 captures both the feedback used from the true performance of the instances (when computing $\hat{\xi}_i(t)$) as well as feedforward information about future input coming from functions earlier in the service-chain (when computing $\hat{r}_i(t)$). This makes it robust against machine uncertainties but also ensures that it reacts fast to sudden changes in the input. For instance, given a service-chain of six functions, function F_5 will know that in Δ_4 time-units, F_4 will have $m_4^{\text{ref}}(t)$ instances running and can thus start as many instances as needed to process this new load.

4 Evaluation

In this section, the automatic service- and admission-controller (AutoSAC) developed in Section 3 is evaluated. First, in Section 4.1, by illustrating how a randomly generated service chain of three functions performs when it is given a 5-h traffic trace. Later, in Section 4.2, AutoSAC is compared with two other "state-of-the-art" methods for scaling cloud services. The comparison is done using a Monte Carlo simulation where the parameters of a five function service chain are randomly generated and then simulated, again using a real traffic trace as input.

The real-world trace of traffic data used as input was gathered over 120 hours from a port in the Swedish University NETwork (SUNET) and then normalized to have a peak of 10,000,000 packets per second as shown in Fig. 1. The simulation was written in the open-source language Julia [20]. The code and traffic trace used for this simulation is provided on GitHub.[1]

4.1 Example chain

For this example, a service chain with three functions where the E2E deadline was set to 30 ms, which in turn was split into local deadlines of 10 ms for each function. The other parameters (i.e., $\bar{s}_i$, Δ_i, ξ_i^{lb}, and ξ_i^{ub}) for every function in the service-chain are generated randomly. The expected service-rate $\bar{s}_i$ was chosen uniformly at random from the interval [100,000, 200,000] pps. The time-overhead Δ_i was drawn uniformly at random from the interval [30, 120] seconds. The machine uncertainty was chosen to be in the range of $\pm 30\%$ of the expected service-rate $\bar{s}_i$. The lower bound of the machine uncertainty was drawn from the interval $[-0.3\bar{s}_i, 0]$ pps and likewise, the upper bound was drawn from $[0, 0.3\bar{s}_i]$ pps.

In Fig. 7, one can see how the service chain scales the number of instances up/down in order to react to the input load. In Fig. 8, one can see how the average utility changes over the course of the simulation. One thing to notice is that the average utility over the service chain remains stable above 0.95 despite large variations in the input.

4.2 Comparing AutoSAC with state-of-the-art

In this section, we will evaluate AutoSAC through a Monte Carlo simulation with $15 \cdot 10^4$ runs where it is compared against two state-of-the-art methods for auto-scaling VMs in industry; *dynamic auto-scaling* (DAS) and *dynamic over-provisioning* (DOP). However, since these two methods do not use any admission control, they are also augmented with the admission controller presented in Section 3.1. The two augmented methods are denoted by "DAS with AC" and "DOP with AC." Hence, in total, the method presented in Section 3 is compared with four other methods.

Dynamic auto-scaling (DAS) This method is currently being offered to customers using Amazon Web Services [21]. It allows the user to monitor different metrics (e.g.,

[1]https://github.com/vmillnert/ICC17simulation

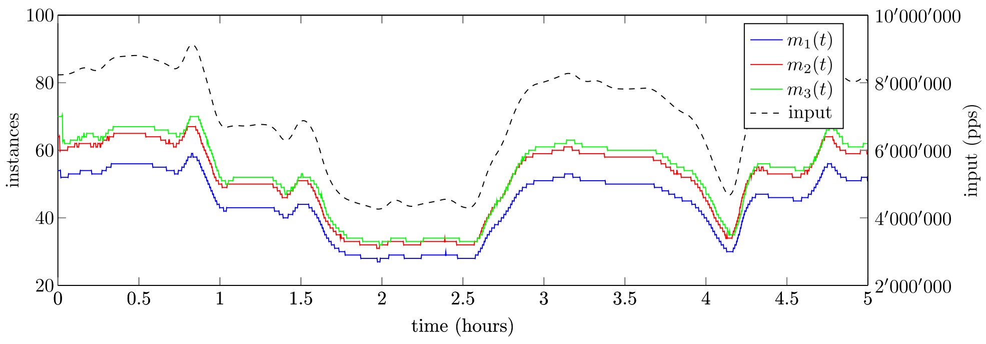

Fig. 7 Simulation of a service chain with three functions where the parameters of each function were randomly generated. One can see how each function reacts to changes of the input rate and automatically scales the number of instances of each function up and down. Feedforward between the functions in the chain ensures a fast reaction since they can scale up/down before the changes occur in their input

CPU utilization) of their VMs using CloudWatch. One can then use it together with their auto-scaling solution to achieve *dynamic auto-scaling*. This allows the user to scale the number of VMs as a function of these metrics. One should note that the CPU utilization can be considered the same as the efficiency metric $e_i(t)$ defined in Eq. 10. For the Monte Carlo simulation, the following rules were used:

- add a VM if the efficiency is above 99%,
- remove a VM if the efficiency is below 95%,

which might seem as a high and tight interval, but it is necessary in order to achieve a high utility.

Dynamic over-provisioning (DOP) A downside with DAS is that it reacts slowly to sudden changes in the input. A natural alternative would therefore be to instead do *dynamic over-provisioning*, where one measures the input to each function and allocate virtual resources such that there is an expected over-provision by 10%.

Monte Carlo simulation The five methods are compared using a Monte Carlo simulation with $15 \cdot 10^4$ runs. For every run, 1 h of input data was randomly selected from the total of 120 h shown in Fig. 1. Furthermore, in every run, a new service-chain with five functions was generated using the method described in Section 4.1. The end-to-end deadline was chosen to 50 ms, which in turn was split into local deadlines of 10 ms for each function.

The evaluation of the Monte Carlo simulation is based on the *average utility* $U(t) = \frac{1}{t}\int_0^t \sum_{i=1}^n u_i(x)\mathrm{d}x$. Since a packet that misses its deadline (which is possible when using DAS or DOP) is considered useless, it is evaluated as a dropped packet when exiting the function. It therefore impacts the availability metric and in turn the utility. Should all packets miss their deadlines in function F_i for a time interval τ, then $a_i(t) = 0 \quad \forall t \in \tau$, i.e., the availability would be evaluated as 0 during this time-interval since the output of the function is considered useless.

Results The mean of the average utility $U(t)$ for all the simulation runs is presented in Fig. 9 for each of the five methods. One can see that AutoSAC achieves a utility that is 30–40% better than that of DAS and DOP. The main reason for this is that they are lacking admission control leading to packets missing their deadlines, which eventually results in a low utility.

When augmenting DAS and DOP with the admission controller derived in Section 3.1, the performance is

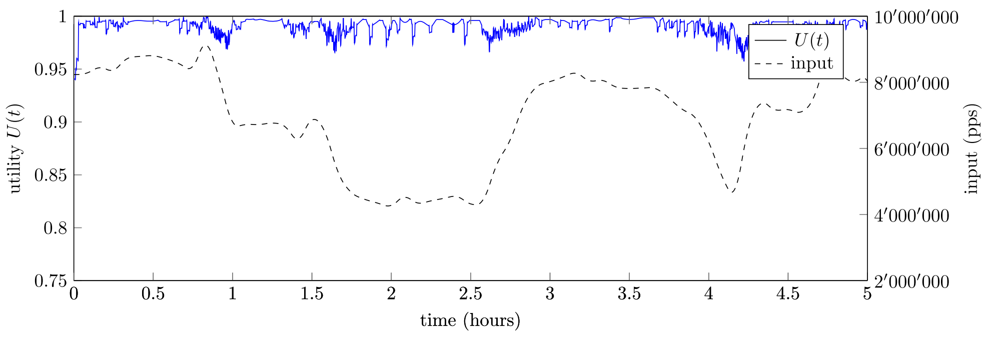

Fig. 8 Average utility for the entire service chain simulated in Section 4. The input is the same as for Fig. 7. One can see that the average utility remains above 0.95 throughout the simulation with small drops when there are large changes in the input rate. However, due to the feedback and feedforward properties, AutoSAC is able to quickly react to these changes and quickly recover a high utility

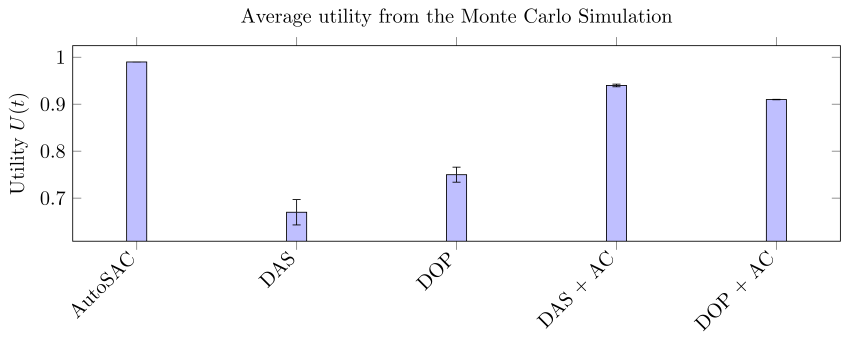

Fig. 9 Results from the Monte Carlo simulation. AutoSAC performs 30–40% better than DAS and DOP. The main reason is the admission controller used in AutoSAC. When augmenting DAS and DOP with this admission controller, their performance is increased by more than 20%. However, AutoSAC still outperforms the augmented methods by 5–10% since it uses feedforward, making it faster to react to input changes, as well as feedback making it more robust to machine uncertainties

increased by 20–40%, purely as a result of not having these sudden drops in performance. However, AutoSAC still performs 5–10% better, due to the feedforward property of AutoSAC which gives it a faster reaction time to changes in the input as well as the feedback property leading to better prediction and robustness against the machine uncertainties.

5 Summary

In this work, we have developed a mathematical model for a NFV Forwarding Graphs residing in a Cloud environment. The model captures, among other things, the time needed to start/stop virtual resources (e.g., virtual machines or containers), and the uncertainty of the performance of the virtual resources which can deviate from the expected performance due to other tenants running loads on the physical infrastructure. The packets that flow through the forwarding graph must be processed by each of the virtual network functions (VNFs) within some end-to-end deadline.

A utility function is defined to evaluate performance between different methods for controlling NFV Forwarding Graphs. The utility function is also used to derive an automatic service- and admission-controller (AutoSAC) in Section 3. It ensures that packets that exit the forwarding graph meet their end-to-end deadline. The service-controller uses feedback from the actual performance of the virtual resources making it robust against uncertainties and deviations from the expected performance. Furthermore, it uses feedforward between the VNFs making it fast to react to changes in the input load.

In Section 4, AutoSAC is evaluated and compared against four other methods in a Monte Carlo simulation with $15 \cdot 10^4$ runs. The input load for the simulation is a real-world trace of traffic data gathered over 120 h. The traffic is normalized to have a peak of 10,000,000 packets per second. AutoSAC is shown to have better performance than what is offered in the cloud industry today. We also show that when augmenting the industry-methods with the admission controller derived in Section 3, they have a significant increase in performance.

It would be interesting to extend this work by investigating how to derive a controller when the true performance is unknown or when the time-overhead needed to start virtual resources is unknown. Moreover, it would be interesting to investigate how to control a forwarding graph that has forks and joins, i.e., a graph structure rather than just a chain.

Acknowledgments The authors would like to thank Karl-Erik Årzén and Joao Monteiro Soares for the useful comments on early versions of this paper.

Source code The source code for the simulation in Section 4 can be found on Github at https://github.com/vmillnert/ICC17simulation.

References

1. ETSI (2012) Network Functions Virtualization (NFV), https://portal.etsi.org/nfv/nfv_white_paper.pdf
2. Leitner P, Cito J (2016) Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. ACM Trans Internet Technol 16(3):15
3. Shen W, Yoshida M, Kawabata T, Minato K, Imajuku W (2014) vconductor: An nfv management solution for realizing end-to-end virtual network services. In: Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific. IEEE, pp 1–6
4. Moens H, De Turck F (2014) Vnf-p: A model for efficient placement of virtualized network functions. In: 10th International Conference on Network and Service Management (CNSM) and Workshop. IEEE, pp 418–423

5. Mehraghdam S, Keller M, Karl H (2014) Specifying and placing chains of virtual network functions. In: 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet). IEEE, pp 7–13
6. Mao M, Li J, Humphrey M (2010) Cloud auto-scaling with deadline and budget constraints. In: 2010 11th IEEE/ACM International Conference on Grid Computing. IEEE, pp 41–48
7. Wang X, Wu C, Le F, Liu A, Li Z, Lau F (2016) Online vnf scaling in datacenters, arXiv preprint arXiv:1604.01136
8. Li Y, Phan L, Loo BT (2016) Network functions virtualization with soft real-time guarantees. In: IEEE International Conference on Computer Communications (INFOCOM)
9. Di Natale M, Stankovic JA (1994) Dynamic end-to-end guarantees in distributed real time systems. In: Proceedings of the 15-th IEEE Real-Time Systems Symposium, pp 215–227
10. Jiang S (2006) A decoupled scheduling approach for distributed real-time embedded automotive systems. In: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 191–198
11. Serreli N, Lipari G, Bini E (2009) Deadline assignment for component-based analysis of real-time transactions. In: 2nd Workshop on Compositional Real-Time Systems, Washington, DC, USA
12. Serreli N, Lipari G, Bini E (2010) The demand bound function interface of distributed sporadic pipelines of tasks scheduled by EDF. In: Proceedings of the 22-nd Euromicro Conference on Real-Time Systems, Bruxelles, Belgium
13. Hong S, Chantem T, Hu XS (2015) Local-deadline assignment for distributed real-time systems. IEEE Trans Comput 64(7):1983–1997
14. Jayachandran P, Abdelzaher T (2008) Delay composition algebra: A reduction-based schedulability algebra for distributed real-timesystems. In: Proceedings of the 29-th IEEE Real-Time Systems Symposium, Barcelona, Spain, pp 259–269
15. Henriksson D, Lu Y, Abdelzaher T (2004). In: Proceedings of the 16th Euromicro Conference on Real-Time Systems, pp 61–68
16. Bonafiglia R, Cerrato I, Ciaccia F, Nemirovsky M, Risso F (2015) Assessing the performance of virtualization technologies for nfv: a preliminary benchmarking. In: 2015 Fourth European Workshop on Software Defined Networks. IEEE, pp 67–72
17. Kapoor R, Porter G, Tewari M, Voelker GM, Vahdat A (2012) Chronos: Predictable low latency for data center applications. In: Proceedings of the Third ACM Symposium on Cloud Computing, ser. SoCC '12. New York, NY, USA: ACM, pp 9:1–9:14. [Online]. Available: http://doi.acm.org/10.1145/2391229.2391238
18. Cruz RL (1991) A calculus for network delay, part I: Network elements in isolation. IEEE Trans Inf Theory 37(1):114–131
19. Le Boudec J-Y, Thiran P Network Calculus: a theory of deterministic queuing systems for the internet, ser. Lecture Notes in Computer Science. Springer, 2001, vol. 2050
20. Bezanson J, Edelman A, Karpinski S, Shah VB (2014) Julia: A fresh approach to numerical computing, arXiv preprint arXiv:1411.1607
21. 2016, 10. [Online]. Available: https://aws.amazon.com/