Achieving predictable and low end-to-end latency for a network of smart services

Millnert, Victor; Eker, Johan; Bini, Enrico

Published in:
2018 IEEE Global Communications Conference (GLOBECOM)

DOI:
10.1109/GLOCOM.2018.8647332

2019

Document Version:
Peer reviewed version (aka post-print)

Link to publication

Citation for published version (APA):
Millnert, V., Eker, J., & Bini, E. (2019). Achieving predictable and low end-to-end latency for a network of smart services. In *2018 IEEE Global Communications Conference (GLOBECOM)* https://doi.org/10.1109/GLOCOM.2018.8647332

Total number of authors:
3

Achieving predictable and low end-to-end latency for a network of smart services

Victor Millnert*, Johan Eker*†, Enrico Bini‡
*Lund University, Sweden
†Ericsson Research, Sweden
‡University of Turin, Italy

***Abstract*—To remain competitive in the field of manufacturing today, companies must constantly improve the automation loops within their production plants. This can be done by augmenting the automation applications with "smart services" such as supervisory-control applications or machine-learning inference algorithms. The downside is that these smart services are often hosted in a cloud infrastructure and the automation applications require a low and predictable end-to-end latency. However, with the 5G technology it will become possible to establish a low-latency connection to the cloud infrastructure and with proper control of the capacity of the smart services, it will become possible to achieve a low and predictable end-to-end latency for the augmented automation applications.**

In this work we address the challenge of controlling the capacity of the smart services in a way that achieves a low and predictable end-to-end latency. We do this by deriving a mathematical framework that models a network of smart services that is hosting several automation applications. We propose a generalized AutoSAC (automatic service- and admission controller) that builds on previous work by the authors [1], [2]. In the previous work the system was only capable of handling a single set of smart services, with a single application hosted on top of it. With the contributions of this paper it becomes possible to host multiple applications on top of a larger, more general network of smart services.

1. Introduction

To remain competitive in the field of manufacturing today, companies must continuously improve the automation loops within their production plants. This is typically done by augmenting the feedback-control loops with "smart services". Examples of such smart services could be supervisory-control applications or machine-learning inference algorithms. The downside is that these automation loops require a low and predictable end-to-end latency, making it very difficult to use smart services that reside in a cloud infrastructure, remote or local.

With the new and promising technology that 5G brings, it will become possible to establish a low-latency connection between the automation applications and the cloud-infrastructure hosting the smart services. This concept is illustrated in Figure 1, where a manufacturing plant is running an augmented feedback-loop that uses the network of smart services residing in the cloud. Evert smart service consists of a set of virtual resources, such as virtual machines (VMs) or containers, making it possible to automatically scale the capacity it. In the end, this enables us to control the end-to-end latency of the network of smart services and in turn the end-to-end latency of entire control-feedback loop.

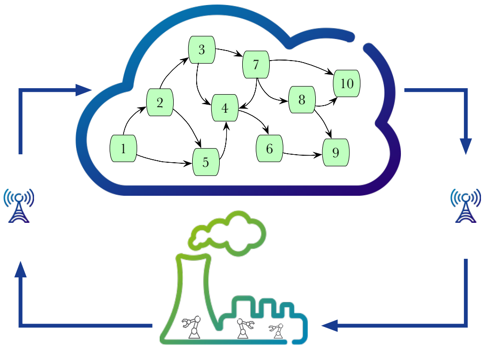

Figure 1: A simple illustration of how automation applications within a factory can make use of a network of smart services executing in the cloud. The automation applications that use these smart services require that the end-to-end deadline is very low and predictable. Since the smart services is built by a set of virtual resources, i.e., virtual machines, it is possible to scale the capacity of them, allowing us to control the latency required for passing through the network of smart services, and in turn the end-to-end latency of the automation applications.

In this work we address the challenge of controlling a network of smart services in a way that achieves a predictable and low end-to-end latency. This paper is a short version of the technical report [3], where we propose a generalized AutoSAC (automatic service- and admission control), that builds on previous work [1], [2]. In the early works the system was only capable of handling a single chain of cloud functions or smart services, a single packet-flow, and a single end-to-end deadline. The work presented in this paper is a generalization necessary to handle the new network structure and can be summed up by the following four parts:

a) *Input prediction:* A feedforward scheme between the smart services to improve the prediction of traffic flow.
b) *Service control:* A small theorem simplifying the strategy used when allocating resources to the smart services.
c) *Selection of node deadlines:* A new optimization problem that assigns intermediary deadlines to the cloud functions.
d) *Admission control:* An admission controller that enforce deadlines while having the highest possible throughput.

Related works

There has been a number of works written on the topic of controlling resources in the cloud and the area of network function virtualization. The majority of them focus on orchestration, i.e. the problem of deciding where in the physical world the virtual resources should be allocated. A few works differ, however, in the way that they instead consider the problem of controlling the NFV graphs with respect to some end-to-end goals. For instance Lin et al. [4] do a static one-time orchestration with the right amount of resources to satisfy some end-to-end requests. Shen et al. [5] develop a management framework, vConductor, for realizing end-to-end virtual network services. However, they are not considering timing-sensitive applications with deadlines for the packets moving through the chain, which is done by Li et al [6] 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

Despite the dynamic nature of the traffic that the NFV graphs will encounter there is only a few works that consider it and aim at designing an elastic, dynamic resource controller to counter the problem. In [7] Mao et al. develop a mechanism for auto-scaling VNF resources to meet a user-specified performance goal. Another work that addresses the problem of meeting performance goals despite the dynamic traffic is [8]. They achieve it by doing load-balancing with a SDN controller between the VNFs. Another work also combining flow scheduling and resource allocation is [9] where they develop a neat mathematical model used as foundation for their synthesis. Other works focusing on developing a model of a VNF is [10], and [11].

The classic method to guarantee end-to-end deadlines of transaction is by holistic analysis [12], in which schedulability analysis at each node is iterated until the convergence of the response times of each transaction is reached. Pellizzoni and Lipari [13] improved the holystic analysis by using offset rather than jitter of tasks. Lorente et al. [14] extended the holysic analysis to the case with nodes running at a fraction of computing capacity (abstracted by a bounded-delay time partition with bandwidth and delay). Similarly, Ashjaei et al. [15] proposed resource reservation over each node along the path.

2. Model and problem formulation

The goal of this section is to derive a mathematical framework for modeling the network of smart services, cloud functions, or virtual network functions, as described in Section 1.

Network and packet flows. To model the network of cloud functions we start by describing the connectivity among them as a *directed graph* $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where

- $\mathcal{V}$ is the set of $n = |\mathcal{V}|$ VNF nodes. For convenience, we label the nodes with the integers from 1 to n, that is $\mathcal{V} = \{1, \dots, n\}$;
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges between these nodes. If $(i, i') \in \mathcal{E}$ then an edge from node $i \in \mathcal{V}$ to node $i' \in \mathcal{V}$ exists.

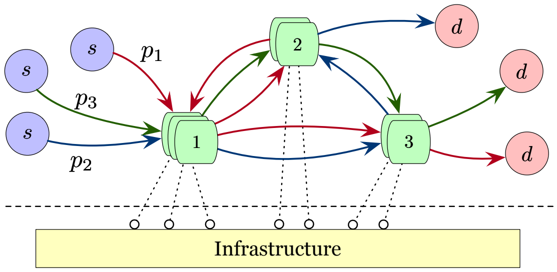

Figure 2: A simple network with three sources, three destinations, three cloud functions, and three packet flows $p_1 = \{1, 2, 1, 3\}$ (in red), $p_2 = \{1, 3, 2\}$ (in blue), and $p_3 = \{1, 2, 3\}$ (in green). Each cloud function consists of a number of virtual resources, e.g., virtual machines or containers, that are deployed on some commodity hardware infrastructure.

The network graph $\mathcal{G}$ will see a set of f different *flows* traversing the network. Each flow has an *end-to-end deadline* associated with it, which for the j-th flow is given by $\mathcal{D}_j$. Moreover, the packets belonging to the j-th flow will traverse the network and visit a specific set of nodes (where they will be processed) in a specific order. This is modeled by the sequence $p_j : \{1, \dots, \ell_j\} \to \mathcal{V}$, with $\ell_j \geq 1$ being the length of the path, such that

$$\forall k = 1, \dots, \ell_j - 1, \quad (p_j(k), p_j(k+1)) \in \mathcal{E}. \qquad (1)$$

The function p_j is therefore a mapping from the integers $1, \dots, \ell_j$ to the set of nodes given by $\mathcal{V}$. Hence, $p_j(k)$ is the k-th node of the j-th path. Naturally, equation (1) enforce the existence of an edge between two consecutive nodes in the path of a flow. One should note that this model is general and allow for flows to traverse a node more than once, thus allowing us to model the typical scenario of automation applications and feedback-control loops mentioned in Section 1. It is therefore useful to define $\delta_{j,i}$ as the number of times that flow j pass through node i.

An example of a network that we can model now is illustrated in Figure 2. Using our framework, it can be modeled by a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ with $\mathcal{V} = \{1, 2, 3\}$ and $\mathcal{E} = \{(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)\}$. The paths of three flows are modeled by $p_1 = \{1, 2, 1, 3\}$, $p_2 = \{1, 3, 2\}$, and $p_3 = \{1, 2, 3\}$, with end-to-end deadlines given by $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$ respectively. It should be noted that $\delta_{1,1} = 2$.

Traffic flow. At time t, the i-th node of the network will see traffic arriving at a rate of $r_i(t) \in \mathbb{R}^+$ packets per second (pps). The arriving packets are either discarded, or admitted into the queue of the node at an *admission rate* of $a_i(t) \in [0, r_i(t)]$ pps. At time t the node will have a *queue size* of $q_i(t)$ packets. In order to process the packets in the queue, a number of $m_i(t) \in \mathbb{N}$ virtual machines will provide the node with a *maximum processing capacity* of $s_i^{\text{cap}}(t)$ packets per second. Naturally, the node might not always be able to process at this maximum capacity. For instance, the queue might be empty. The actual rate by which the node is processing packets with is therefore given by the *service rate* $s_i(t)$:

$$s_i(t) = \begin{cases} s_i^{\text{cap}}(t) & \text{if } q_i(t) > 0, \\ \min(s_i^{\text{cap}}(t),\, a_i(t)) & \text{else.} \end{cases} \qquad (2)$$

Machine model. In order to adapt the maximum processing capacity of the node to changes of the traffic rate, it is possible to control the number of virtual machines that are running in the node. This is done with the *control signal* $m_i^{\text{ref}}(t) \in \mathbb{N}$. To model the time necessary to start/stop a virtual machine, the number of VMs that are running is a delayed version of this control signal:

$$m_i(t) = m_i^{\text{ref}}(t - \Delta_i), \tag{3}$$

where $\Delta_i \in \mathbb{R}^+$ is the *time-delay* needed to start/stop a virtual machine.

Every machine instance running in the i-th node has an expected service capacity of $\bar{s}_i$ packets per second. However, the actual performance will often deviate from the expected performance and can depend on where it is deployed as well as on what other processes running on the physical server that the VM is hosted on, as shown in [16]. To model this, the maximum processing capacity of the node is given by:

$$s_i^{\text{cap}}(t) = m_i(t) \cdot (\bar{s}_i + \hat{\xi}_i(t)), \tag{4}$$

where $\hat{\xi}_i(t)$ is the *average machine uncertainty*. For increased readability we will neglect this uncertainty for the remainder of this section and Section 3. If interested, this uncertainty is treated thoroughly in the technical report [3]. It should be noted however, that when evaluating the performance in Section 4, the uncertainty will indeed be treated.

Node latency. It is useful to measure the time it takes a packet to pass through the i-th node. We denote this as the *node latency* $L_i(t)$:

$$L_i(t) = \inf\{\tau \geq 0 \,:\, A_i(t - \tau) \leq S_i(t)\}, \tag{5}$$

where $A_i(t) = \int_0^t a_i(x)\mathrm{d}x$ and $S_i(t) = \int_0^t s_i(x)\mathrm{d}x$.

Node deadline. For the scenario of augmented automation loops outlined in Section 1 it is assumed that there are many different packet flows going through the different smart services of the network. In order to reduce the complexity that this brings with it, we find it useful to introduce an intermediary *node deadline* D_i. This means that a packet that arrives to the i-th node will only be admitted into the node if it is possible to guarantee that the packet will be processed and exit the function within D_i seconds. This must hold, regardless of which flow the packet belongs to. In other words, every packet arriving to the i-th node will have the same node deadline D_i, even though they might belong to different flows with different *end-to-end deadlines*. Mathematically speaking, this means that when assigning node deadline one will be constrained by

$$\sum_{\forall i \in p_j} \delta_{j,i} \cdot D_i \leq \mathcal{D}_j \quad j = 1, \ldots, f, \tag{6}$$

which states that the sum of all the node deadlines over a path j must be less than the end-to-end deadline of that path.

2.1. Problem formulation

As mentioned in the introduction of this paper, the goal is to derive ways of controlling the resources allocated to the smart services, or the cloud functions, in order to have predictable and low end-to-end latencies for the augmented automation-applications using the network of smart services Informally, this goal can be described as trying to ensure that *"the end-to-end deadlines of the different packet-flows are met, while using as little resources and discarding as few packets as possible"*. As an aid when evaluating this goal we propose three formal metrics, defined in (7):

a) *availability* $U^{\mathsf{a}}(t)$ – is there a high throughput, or are many packets being discarded?
b) *efficiency* $U^{\mathsf{e}}(t)$ – are the nodes efficient, or are they wasting resources?
c) *utility* $U(t)$ – a combination of the availability and the efficiency.

The intuition behind the choice of these metrics is that it is typically easy to have either a high efficiency or a high availability but not both at the same time. One can for instance choose to overallocate resources in a node, something typically seen today, resulting in a high availability but a poor efficiency, or one can instead choose to have a high efficiency, forcing the node to discard many packets.

$$\begin{aligned} &U^{\mathsf{a}}(t) = \tfrac{1}{n}\sum_{i \in \mathcal{V}} u_i^{\mathsf{a}}(x), \quad U^{\mathsf{e}}(t) = \tfrac{1}{n}\sum_{i \in \mathcal{V}} u_i^{\mathsf{e}}(x), \\ &U(t) = \tfrac{1}{n}\sum_{i \in \mathcal{V}} u_i^{\mathsf{a}}(x) \cdot u_i^{\mathsf{e}}(t), \end{aligned} \tag{7}$$

with $u_i^{\mathsf{a}}(t)$ and $u_i^{\mathsf{e}}(t)$ given by

$$\begin{aligned} u_i^{\mathsf{a}}(t) &= \begin{cases} s_i(t)/r_i(t) & \text{if } L_i(t) \leq D_i \\ 0 & \text{if } L_i(t) > D_i \end{cases} \\ u_i^{\mathsf{e}}(t) &= \frac{s_i(t)}{s_i^{\text{cap}}(t)} \end{aligned} \tag{8}$$

3. AutoSAC for a network of cloud functions

The general idea behind AutoSAC (automatic service- and admission controller) is to create an abstraction of the parts included in a controller that can achieve the goal introduced in Section 1, which is to have a low and predictable end-to-end latency for a network of smart services. From our perspective these parts are illustrated in Figure 3 and include: *i)* input prediction, *ii)* service control, *iii)* selection of node deadlines, and *iv)* admission control.

In this section we will propose possible solutions for each of the four parts, but ultimately the goal is to stimulate research resulting in better versions, which in the end can replace what is proposed here. We will present a brief overview of the proposed solutions, followed with a slightly more detailed description in Sections 3.1–3.3. For a complete picture we refer the reader to the technical report [3]. Before diving into this however, it should be noted that the underlying timing-assumptions for this work, shown in Table 1, is the same as it has been for the previous work [1] and [2], on top of which this work builds.

Parameter	timing assumption
long-term trend change of the input	1min – 1h
time-overhead Δ_i	1s – 1min
end-to-end deadline $\mathcal{D}_j$	1µs – 100 ms

TABLE 1: Timing assumptions for the end-to-end deadline, the rate-of-change of the input traffic, and the time-overhead for starting/stopping instances. One should note that the difficulty of controlling this system stems from the fact that they are all on different orders of magnitude.

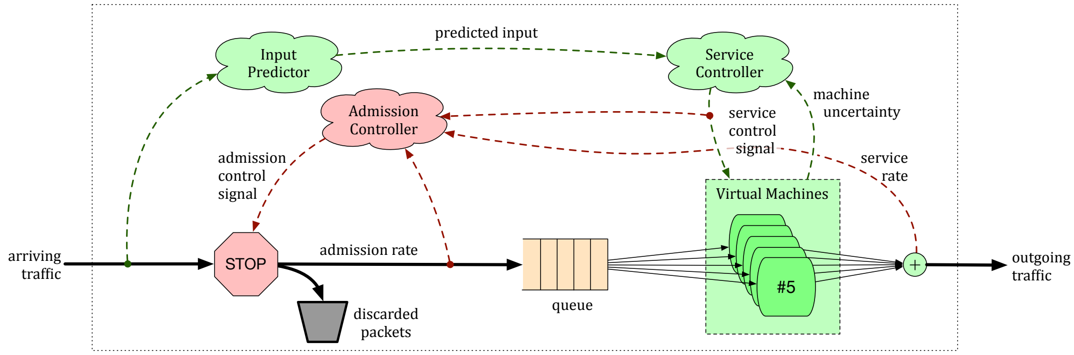

Figure 3: An abstract overview of the parts included in the controller of the smart services. Together, the input predictor, service controller, and admission controller can ensure that there is the right number of virtual machines running in the node, so that the latency for passing through the node is as low as it has to be. The colored arrows in the figure illustrate how information is passed between the controller-parts, i.e, from the input predictor to the service controller, and the black arrows indicate the traffic flow.

As a brief overview, the idea is that the *input predictor* is meant to predict the future arrival rate to the different nodes in the network. It should do this using only local information from the node, i.e., in a decentralized way. This information is then used by the *service controller* to compute the amount of allocated resources necessary to maximize the utility function (7). To do this, it is necessary to consider the variations of the performance, the machine uncertainty, of the virtual machines. To ensure that the end-to-end latency for the different flows in the network is low and predictable, we set up an optimization problem in *selection of node deadlines*. The goal is to split the end-to-end deadlines for the different flows into intermediary node deadline, thus reducing the complexity of the control problem. Finally, *admission controller* use information from the service controller when deciding how many packets that should be admitted into the node. Its goal is the admit as many packets as possible, while still ensuring that no packet violate the node deadline.

3.1. Input prediction

The importance of having a good prediction of arriving traffic comes form the fact that it takes Δ_i seconds for the i-th node to start/stop a VM. Recall that the number of VMs running is controlled through $m_i^{\mathrm{ref}}(t)$ and also that $m_i(t) = m_i^{\mathrm{ref}}(t-\Delta_i)$. Hence it takes Δ_i seconds for any changes of $m_i^{\mathrm{ref}}(t)$ to take effect. This means that at time t the node has to make a decision about how many VMs will be needed at time $t+\Delta_i$. For this decision to be good, to yield the highest possible utility, a good prediction of the arrival rate at time $t+\Delta_i$ is necessary.

The traffic that pass through a node originates from many different sources. Some traffic has passed through many other nodes within the network, while other traffic comes directly from an outside source. For this reason, we find it useful to introduce a distinction between these two scenarios. We therefore introduce the notion of *internal traffic* $r_i^{\mathrm{I}}(t)$ and *external traffic* $r_i^{\mathrm{E}}(t)$. By internal traffic we mean traffic that arrives to the node directly from another node within the network, and by external we mean traffic that arrive directly to the node *without* passing through another node in the network, i.e, directly from an outside source. Together, they form the total arrival rate for the node:

$$r_i(t) = r_i^{\mathrm{I}}(t) + r_i^{\mathrm{E}}(t). \tag{9}$$

The distinction between internal and external traffic is useful because it allows us to use different methods when predicting their future arrival rates. We can thus denote the *predicted arrival rate* to the node as $\hat{r}_i(t)$:

$$\hat{r}_i(t) = \hat{r}_i^{\mathrm{I}}(t) + \hat{r}_i^{\mathrm{E}}(t), \tag{10}$$

where $\hat{r}_i^{\mathrm{I}}(t)$ and $\hat{r}_i^{\mathrm{E}}(t)$ are the predictions of the internal and the external traffic respectively. Next, the strategies for predicting these two will be described in more detail.

Predicting external traffic. When predicting the traffic arriving to the node directly from an external source, the timing assumptions of Table 1 imply that it is sufficient to use a linearization-method. The reason is that the rate-of-change of the external traffic is assumed to be on a different time-scale than the time-delay Δ_i. Using linearization, the prediction of the external traffic can be computed as

$$\hat{r}_i^{\mathrm{E}}(t+\Delta_i) = r_i(t) + \Delta_i \cdot \frac{\mathrm{d}r_i(t+\Delta_i)}{\mathrm{d}t}. \tag{11}$$

It should be noted that this method is the same as has been proposed in the previous works [1] and [2].

Prediction of internal traffic. Due to the network structure of the nodes it is possible to achieve a very good prediction of the internal traffic, i.e., the traffic flowing between two nodes in the network. In fact, this can be achieved by having every node that is sending traffic to the i-th node to also send its future predictions. Let us denote the prediction of traffic from node i' to node i as $\hat{r}_{(i',i)}^{\mathrm{I}}(t)$. The prediction of the total internal traffic arriving to node i can then be written as

$$\hat{r}_i^{\mathrm{I}}(t) = \sum_{i' \in \mathcal{V}} \hat{r}_{(i',i)}^{\mathrm{I}}(t). \tag{12}$$

The idea is therefore that it is the information about $\hat{r}_{(i',i)}^{\mathrm{I}}(t)$ that the i'-th node will send to node i. Node i can then simply sum these predictions up to form a good prediction about the internal traffic that will arrive in the future. Predicting $\hat{r}_{(i',i)}^{\mathrm{I}}(t)$ should be done within the i'-th node, using only local information, and can be decomposed into:

1) predict the future service-rate $\hat{s}_{i'}(t)$, and
2) predict the fraction of traffic routed to node i.

For a complete derivation how the internal traffic is predicted we refer to the technical report [3].

3.2. Service control

With good prediction of the arrival rate at time $t + \Delta_i$, it becomes possible to make a good decision about how much resources will be needed at that time, which in the end makes it possible to make a good decision about what $m_i^{\text{ref}}(t)$ should be at time t. The service controller we use was derived in the earlier works [1] and [2]. However, we will here present a simplification of the control-law when a large number of virtual machines is used. The service controller developed in the previous works is given by:

$$m_i^{\text{ref}}(t) = \begin{cases} \lfloor \kappa_i(t) \rfloor, & \text{if } \lfloor \kappa_i(t) \rfloor \lceil \kappa_i(t) \rceil \geq \kappa_i^2(t) \\ \\ \lceil \kappa_i(t) \rceil, & \text{else} \end{cases} \tag{13}$$

where $\kappa_i(t) = \hat{r}_i(t + \Delta_i)/\bar{s}_i$. The intuition behind this control-law comes from trying to maximize the utility function (7). Since the exact number of VMs required to match the incoming traffic is given by $\kappa_i(t)$ it becomes necessary to either select $m_i^{\text{ref}}(t) = \lfloor \kappa_i(t) \rfloor$ leading to slightly too little processing capacity (implying that packets will be discarded) or to instead select $m_i^{\text{ref}}(t) = \lceil \kappa_i(t) \rceil$ leading to slightly too much processing capacity (implying that resources will be wasted). The statement deciding which case to chose is meant to optimize the utility function, (7). For a complete derivation of (13) we refer to the earlier works. It should also be noted that hidden within this equation is a feedback-control law compensating for the variations of the performance of the virtual machines, i.e. the machine uncertainty.

As mentioned earlier, the addition made in this work is a small theorem showing that if one should have a large number of virtual machines, the control-law of (13) can be simplified into $m_i^{\text{ref}}(t) = \lfloor \kappa_i(t) \rceil$ as shown in Theorem 3.2 below.

Theorem 3.1. *When the node is having a large number of virtual machines running, $m_i^{\text{ref}}(t)$ can be computed as $m_i^{\text{ref}}(t) = \lfloor \kappa_i(t) \rceil$.*

Proof: A complete proof is shown in the technical report [3], but the idea is that one can rewrite $\kappa_i(t)$ as $\kappa_i(t) = \lfloor \kappa_i(t) \rfloor + \rho$, with $\rho \in [0, 1)$. This allows us to rewrite the control-law as

$$m_i^{\text{ref}}(t) = \begin{cases} \lfloor \kappa_i(t) \rfloor, & \text{if } \rho \leq \frac{1}{2} - \frac{\rho^2}{2\lfloor \kappa_i(t) \rfloor} \\ \lceil \kappa_i(t) \rceil & \text{else} \end{cases}$$

which for a large $\kappa_i(t)$, and thus a large $\lfloor \kappa_i(t) \rfloor$, becomes

$$m_i^{\text{ref}}(t) = \lfloor \kappa_i(t) \rceil .$$

□

3.3. Selection of node deadlines

To be able to ensure that the end-to-end latency for the different flows of the network are low and predictable there will be an end-to-end deadline associated with the each flow. The end-to-end deadline for the j-th flow is $\mathcal{D}_j$. Due to the variations of the performance of the virtual machines, as well as variations in the traffic load, it becomes necessary to sometimes discard packets. Doing this on a global scale, for each flow, becomes very complex so the proposition in this work is to instead split the end-to-end deadlines into smaller intermediary *node deadlines*. This means one deadline for every node in the network. In other words, every packet entering a node i will have a node deadline of D_i, regardless of which flow it belongs to. It will then be the task of the *admission controller* (presented in Section 3.4) to ensure that these node deadlines are met.

The ratio of Δ_i/D_i. How should one go about to split these end-to-end deadlines into smaller node deadlines then? To be able to address this, one must know how different choices of node deadlines affect the utility of the system. To gain some understanding of this, a thorough analysis was made in the technical report [3]. The intuition gained from it was that the ratio of Δ_i/D_i is a good metric to optimize for when selecting the node deadlines. This ratio gives insight in how difficult it is to control a node. A small ratio means that it is easy to quickly react to changes of the arrival rate, or to performance changes of the virtual machines. In fact, a ratio smaller than 1 implies that there is always time to react to such changes. As the ratio grows larger, it becomes impossible to react to such changes. Instead, it becomes necessary to have a proactive approach—to predict the future arrival rates. The larger the ratio, the longer into the future one must predict, hence the harder it becomes to control the node.

The optimization problem. We use the insights about Δ_i/D_i to set up an optimization problem that assigns node deadlines D_i to every node in a way so that no end-to-end deadline $\mathcal{D}_j$ is violated. The optimization problem is given as:

$$\begin{aligned} &\text{minimize} && \sum_{\forall i \in \mathcal{V}} \Delta_i / D_i \\ &\text{subject to} && \sum_{\forall i \in p_j} \delta_{j,i} \cdot D_i \leq \mathcal{D}_j \quad j = 1, \ldots, f \\ & && D_i \geq 0 \quad \forall i \in \mathcal{V} \end{aligned} \tag{14}$$

where $\delta_{j,i}$ indicates how many times path j goes through node i and Δ_i indicates the time-overhead necessary to change the number of VMs in the i-th node. As shown in [3], this optimization problem can either be solved using disciplined cone programming or by standard methods such as Lagrange multipliers.

3.4. Admission control

The goal of the admission controller is to allow as many packets as possible to pass through the node, without any of them to missing the node deadline. This is achieved by controlling the admission rate $a_i(t)$ according to the admission policy presented in Theorem 3.2. In the theorem we show that the policy is optimal and that it is capable of computing the admission rate in constant time. Furthermore, an overview of the admission controller is shown in the form of a block diagram in Figure 5. It highlights the fact that the computation required for this admission policy can be

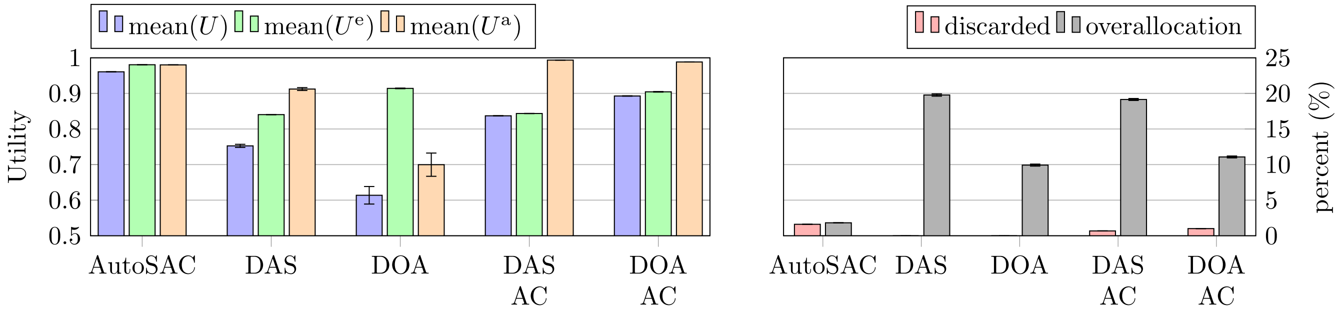

Figure 4: Through a large number of simulations of the network depicted in Figure 2 the performance of the automatic service- and admission controller (AutoSAC) developed in this paper was compared with what is currently being used in industry. The methods AutoSAC was compared against was dynamic auto-scaling (DAS) and dynamic overallocation (DOA). Neither DAS nor DOA has admission controller so they were augmented with the one developed in this paper. In the left figure one can see the result of the average utility, efficiency, and availability for each of the five methods, and in the right figure the fraction of the incoming packets that are discarded and the average amount of overallocation of resources.

computed instantly and in a continuously, something that is useful when implementing it in real-life. Moreover, it enables the user to continuously adapt the policy to changes of the node deadline. In other words, the admission policy derived in Theorem 3.2 allow one to dynamically change the node deadline L_i over time. This is not something that will be used in this paper, but investigated in future work.

Theorem 3.2. *The admission policy*

$$a_i(t) = \begin{cases} r_i(t) & \text{if } A_i(t) < S_i(t+D_i) \\ \min(r_i(t),\, s_i(t+D_i)) & \text{else.} \end{cases}$$

will admit as many packets as possible while still ensuring that the admitted packets meet the node deadline of L_i.

Proof: For a complete proof, and one that includes the machine uncertainty, please see the technical report [3]. The intuition is that incoming packets are guaranteed to meet their deadlines as long as $L_i(t) < D_i(t)$. From (5) it follows that this is equivalent to $S_i(t + D_i) > A_i(t)$. Hence, as long as this inequality holds, any incoming packet is guaranteed to meet its deadline. Should instead $L_i(t) = D_i(t)$, the largest possible admission rate becomes $a_i(t) \leq s_i(t + D_i)$. □

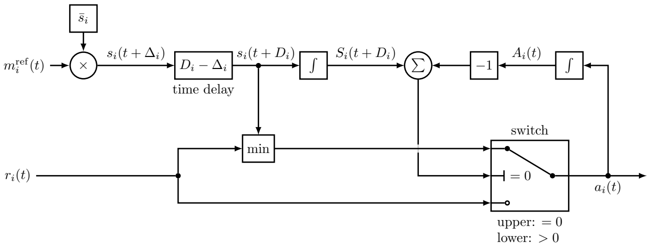

Figure 5: Block-diagram of the proposed admission controller. It uses feedback from the queue-size (computed using feedback from $m_i^{\mathrm{ref}}(t)$ and $a_i(t)$) in order to compute whether the incoming packets can be admitted or not. As shown in Theorem 3.2 the feedback-law (or admission policy) derived here is able to admit as many packets as possible, while still guaranteeing that all the admitted packets will meet the node deadline D_i.

4. Evaluation

To evaluate the performance of the new AutoSAC (automatic service- and admission controller) presented in Section 3, a large Monte Carlo simulation was performed. A complete description of the simulation method can be found in the technical report [3], but the basic idea was to randomly generate a network of smart services, similar to the one shown in Figure 2. For every simulation, the properties of the network such as the time-delay Δ_i for the different nodes, nominal service-rates of the VMs $\bar{s}_i(t)$, end-to-end deadlines $\mathcal{L}_j$ of the flows, etc., was randomly generated. To ensure that the traffic flowing through the network was realistic we used a real traffic-trace from the Swedish university network (SUNET). In Figure 6 we show an example of the traffic rate for three different flows in one of the simulations. One can see that it fluctuates quite a bit.

With this set-up, the generalized AutoSAC was compared against two common industry-methods: *dynamic auto scaling* (DAS) and *dynamic overallocation* (DOA). Since neither of these two methods use an admission controller, we augment both of them with the the one developed in this paper, resulting in two additional methods: (DAS+AC) and (DOA+AC). Each of these five methods was evaluated using a 1,000 MC simulations.

Dynamic auto-scaling (DAS). This is the auto-scaling method offered to customers of Amazon Web Services, [17]. It is a purely reactive method and is based by

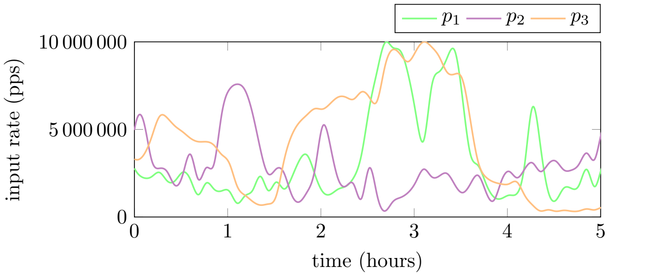

Figure 6: Input traffic for three packet flows, i.e., applications, for one of the many simulations.

having the users monitoring a specific metric (e.g., CPU utilization) of their VMs using CloudWatch. The user then specifies two thresholds on which the auto-scaling is based upon. For this evaluation the efficiency metric $u_i^{\mathsf{e}}(t)$ was used as the auto-scaling metric with the following thresholds:

$$\begin{cases} \text{add a VM} & \text{if } u_i^{\mathsf{e}}(t) > 0.9, \\ \text{remove a VM} & \text{if } u_i^{\mathsf{e}}(t) < 0.8. \end{cases}$$

Dynamic overallocation (DOA). A downside with DAS is that it only uses feedback to control the number of instances it needs. With a large ratio between the time-overhead and the local node deadline it becomes very difficult to control solely based on feedback. An alternative approach commonly used in industry is to instead use dynamic overallocation where one measures the input to each function and allocates virtual resources such that there is an expected overallocation of 10%.

Results. The result of the Monte Carlo simulation is shown in Figure 4. The left part show the comparison of the average utility, efficiency, and availability. The right part illustrate the same result, but by instead showing the fraction of packets that are discarded as well as how much overallocation each method cause.

One can see that AutoSAC is close to optimal in the average utility, availability, and efficiency. As expected, DAS has an efficiency in the range of 0.8–0.9, since those are the thresholds by which it bases its auto-scaling on. It should be noted that increasing this band of thresholds did not improve the efficiency (the current range yielded the best performance). DOA achieves an efficiency of around 0.85 which is also expected. The reason for the poor average utility for these two methods, however, is due to the lack of a admission controller. Without one a queue will build up every time there is a lack of processing capacity. This in turn increases the latency, causing packets to miss their deadlines resulting in a low availability. By augmenting DAS and DOA with the admission controller developed in Section 3.4 the availability is significantly increased by allowing it to drop some of the packets in a strategic way. By looking at the right part of Figure 4 one realize that it is only a very small fraction of the packets that are actually discarded, so it is a sacrifice well worth making.

5. Summary

In this paper we derive a general mathematical model for network of smart services that can be used by a set of automation applications requiring a very low and predictable end-to-end latency. The model is used to derive control-laws for controlling the resources allocated of the smart services in a way that achieves the required end-to-end latency. The proposed control-strategies are evaluated and compared against other methods, commonly used in the cloud industry today. The evaluation, based on a large Monte Carlo simulation, shows that the automatic service- and admission controller (AutoSAC) proposed in this work performs very well. The evaluation also shows that the optimal admission controller proposed in this work can significantly increase the performance of existing industry methods.

Source code Traffic data and code for simulations: https://github.com/vmillnert/GLOBECOM18simulation.

References

[1] V. Millnert, J. Eker, and E. Bini, “AutoSAC: automatic scaling and admission control of forwarding graphs,” *Annals of Telecommunications*, vol. 16, no. 3, pp. 15–12, Aug. 2017.

[2] ——, “Dynamic control of NFV forwarding graphs with end-to-end deadline constraints,” in *ICC 2017 - 2017 IEEE International Conference on Communications*. IEEE, 2017, pp. 1–7.

[3] ——, “Achieving predictable and low end-to-end latency for a cloud-robotics network,” 4 2018. [Online]. Available: http://lup.lub.lu.se/record/fe4a3a04-ee6d-49e2-b634-7f9917340641

[4] T. Lin, Z. Zhou, M. Tornatore, and B. Mukherjee, “Optimal Network Function Virtualization Realizing End-to-End Requests,” in *GLOBECOM 2015 - 2015 IEEE Global Communications Conference*. IEEE, 2014, pp. 1–6.

[5] W. Shen, M. Yoshida, T. Kawabata, K. Minato, and W. Imajuku, “vConductor: An NFV management solution for realizing end-to-end virtual network services,” in *2014 16th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2014, pp. 1–6.

[6] Y. Li, L. T. X. Phan, and B. T. Loo, “Network functions virtualization with soft real-time guarantees,” in *IEEE INFOCOM 2016 - IEEE Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[7] M. Mao, J. Li, and M. Humphrey, “Cloud auto-scaling with deadline and budget constraints,” in *2010 11th IEEE/ACM International Conference on Grid Computing (GRID)*. IEEE, 2010, pp. 41–48.

[8] A. Leivadeas, M. Falkner, I. Lambadaris, and G. Kesidis, “Resource Management and Orchestration for a Dynamic Service Chain Steering Model,” in *GLOBECOM 2016 - 2016 IEEE Global Communications Conference*. IEEE, 2016, pp. 1–6.

[9] H. Feng, J. Llorca, A. M. Tulino, and A. F. Molisch, “Dynamic network service optimization in distributed cloud networks,” in *IEEE INFOCOM 2016 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2016, pp. 300–305.

[10] G. Faraci, A. Lombardo, and G. Schembra, “A building block to model an SDN/NFV network,” in *ICC 2017 - 2017 IEEE International Conference on Communications*. IEEE, 2017, pp. 1–7.

[11] Y. Ren, T. Phung-Duc, J.-C. Chen, and Z.-W. Yu, “Dynamic Auto Scaling Algorithm (DASA) for 5G Mobile Networks,” in *GLOBECOM 2016 - 2016 IEEE Global Communications Conference*. IEEE, 2016, pp. 1–6.

[12] K. Tindell and J. Clark, “Holistic schedulability analysis for distributed hard real-time systems,” *Microprocessing and Microprogramming*, vol. 50, pp. 117–134, Apr. 1994.

[13] R. Pellizzoni and G. Lipari, “Feasibility analysis of real-time periodic tasks with offsets,” *Real-Time Systems*, vol. 30, no. 1–2, pp. 105–128, 2005.

[14] J. L. Lorente, G. Lipari, and E. Bini, “A hierarchical scheduling model for component-based real-time systems,” in *Proceedings of the 20-th International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, Apr. 2006.

[15] M. Ashjaei, S. Mubeen, M. Behnam, L. Almeida, and T. Nolte, “End-to-end resource reservations in distributed embedded systems,” in *22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug. 2016, pp. 1–11.

[16] P. Leitner and J. Cito, “Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds,” *ACM Transactions on Internet Technology*, vol. 16, no. 3, pp. 1–23, Aug. 2016.

[17] (2016, 10). [Online]. Available: https://aws.amazon.com/documentation/autoscaling/