SNC-Meister: Admitting More Tenants with Tail Latency SLOs

Timothy Zhu, Daniel S. Berger*, Mor Harchol-Balter

May 31, 2016
CMU-CS-16-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*University of Kaiserslautern

This work was supported by NSF-CMMI-1538204, NSF-CMMI-1334194, and NSF-CSR-1116282, by the Intel Pittsburgh ISTC-CC, and by a Google Faculty Research Award 2015/16.

Keywords: tail latency guarantees, stochastic network calculus, computer networks, quality of service

Abstract

Meeting tail latency Service Level Objectives (SLOs) in shared cloud networks is known to be an important and challenging problem. The main challenge is determining limits on the multi-tenancy such that SLOs are met. This requires calculating latency guarantees, which is a difficult problem, especially when tenants exhibit bursty behavior as is common in production environments. Nevertheless, recent papers in the past two years (Silo, QJump, and PriorityMeister) show techniques for calculating latency based on a branch of mathematical modeling called Deterministic Network Calculus (DNC). The DNC theory is designed for adversarial worst-case conditions, which is sometimes necessary, but is often overly conservative. Typical tenants do not require strict worst-case guarantees, but are only looking for SLOs at lower percentiles (e.g., 99th, 99.9th). This paper describes SNC-Meister, a new admission control system for tail latency SLOs. SNC-Meister improves upon the state-of-the-art DNC-based systems by using a new theory, Stochastic Network Calculus (SNC), which is designed for tail latency percentiles. Focusing on tail latency percentiles, rather than the adversarial worst-case DNC latency, allows SNC-Meister to pack together many more tenants: in experiments with production traces, SNC-Meister supports 75% more tenants than the state-of-the-art. We are the first to bring SNC to practice in a real computer system.

1 Introduction

Meeting tail latency Service Level Objectives (SLOs) in multi-tenant cloud environments is a challenging problem. A tail latency SLO such as a 99th percentile of 50ms (written $T99 < 50ms$) requires that 99% of requests complete within 50ms. Researchers and companies like Amazon and Google repeatedly stress the importance of achieving tail latency SLOs at the 99th and 99.9th percentiles [14, 52, 13, 39, 4, 47, 48, 26, 45, 23, 27, 55]. As demand for interactive services increases, the need for latency SLOs will become ever more important. Unfortunately, there is little support for specifying tail latency requirements in the cloud. Latency is much harder to guarantee since it is affected by the burstiness of each tenant, whereas bandwidth is much easier to divide between tenants. Tail latency is particularly affected by burstiness, and recent measurements show that the 99.9th latency percentile can vary tremendously and is typically an order of magnitude above the median [35].

1.1 The case for request latency SLOs

Throughout this paper, we measure *request latency* (a.k.a., flow completion time), which is defined as the time from when a tenant's application makes a request for data to the time until all the requested data is received by the tenant's application. This is in contrast to *packet latency*, which is the time it takes a packet to traverse through the network. Packet latency is the right metric when requests are small. However, as the amount of data used increases, request latency becomes the most relevant granularity (as argued in [54]).

1.2 Queueing is inevitable for request latency

The major cause for high request latency is almost always excessive queueing delay [23, 27]. Queueing is inevitable. In production environments, traffic is typically bursty as shown in Fig. 3(a). When these bursts happen simultaneously, the result is high queueing delays.

Queueing can occur both within the network (*in-network queueing*) and at the end-hosts (*end-host queueing*). Some works (e.g., Fastpass [39], HULL [3]) claim to eliminate or significantly reduce queueing. What they actually mean is that they eliminate *in-network queueing* by shifting the queueing to the end-hosts with rate limiting. This produces great benefits for packet latency, which does not include this end-host queueing time. However, these techniques do not solve the problem for request latency, which by definition captures the entire queueing time, both in-network queueing and end-host queueing. Queueing delay still comprises the biggest portion of request latency, particularly when looking at the tail percentiles [23].

1.3 Dual goals: meeting tail latency SLOs and achieving high multi-tenancy

The goal of this paper is two-fold: 1) We want to meet tail request latency SLOs and 2) we want to admit as many tenants as possible. Clearly, there's an obvious *tradeoff*. If we admit very few tenants, e.g., just 1 tenant, then it's very likely that we can meet that tenant's request latency SLO, because there will be very little queueing (queues will only be caused by that single tenant's bursts,

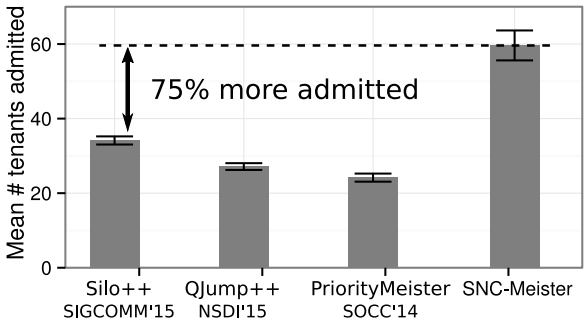


Figure 1: Admission numbers for state-of-the-art admission control systems and SNC-Meister in 100 randomized experiments. The state-of-the-art systems are Silo [27], QJump [23], and PriorityMeister [55], which have all been published in the last two years. In each experiment, 180 tenants, each submitting hundreds of thousands of requests, arrive in random order and seek a 99.9% SLO randomly drawn from {10ms, 20ms, 50ms, 100ms}. While all systems meet all SLOs, SNC-Meister is able to support on average 75% more tenants with tail latency SLOs than the next-best system.

not contention with other tenants). By contrast, if we admit many tenants, then there will be high contention across tenants, resulting in long queues and possibly violating SLOs. *Admission control* is the necessary component that limits the multi-tenancy so as to guarantee that we *only* admit tenants whose SLOs we can meet. The challenge in admission control is predicting what the request latency will be for each tenant.

1.4 The state of the art in admission control: worst-case bounds on the request latencies

Predicting request latency is not an easy task due to sharing the network with many tenants that send bursty traffic as is common in production environments. The burstiness makes it challenging to meet SLOs by only keeping system load below a "magic number", e.g., below 60%. For example, Fig. 2 illustrates that there can be violations of tail latency SLOs even at low loads. Furthermore, the number of SLO violations depends on the specific SLO latencies and percentiles, so even determining a magic load number is not straightforward. This shows that load alone is an insufficient criterion to determine admission decisions.

The state-of-the-art in admission control are Silo (SIGCOMM 2015 [27]), QJump (NSDI 2015 [23]), and PriorityMeister (SoCC 2014 [55]). These systems perform admission control by using Deterministic Network Calculus (DNC) to calculate upper bounds on the request latency. Typically DNC characterizes a tenant's request process based on the maximum arrival rate and burst size. DNC then uses each tenant's maximum rate/burst constraint to compute the worst-case latency within a shared network. If the worst-case latency for a tenant is higher than its SLO, the tenant is not admitted.

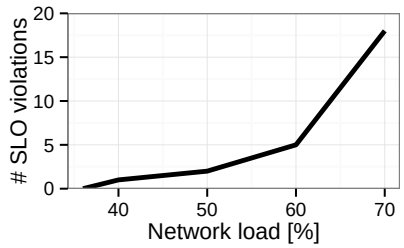


Figure 2: Without admission control, the number of SLO violations increases quickly as the load increases.

The above systems all use DNC, but in somewhat different ways. Silo uses DNC to calculate the amount of queueing within the network and performs admission control to ensure that network switch buffers do not overflow. QJump offers several classes with different latency-throughput trade-offs, for which latency guarantees are calculated with DNC. PriorityMeister considers different prioritizations of tenants. For each priority ordering, PriorityMeister uses DNC to bound the worst-case latency of each tenant. PriorityMeister aims to choose a priority ordering that maximizes the number of tenants that can meet their SLOs if admitted.

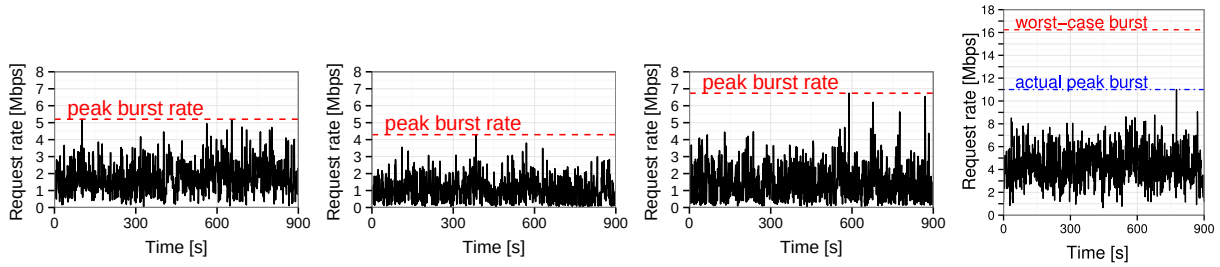
1.5 The limitations of DNC

While DNC is an excellent tool for latency analysis, it assumes that tenants behave adversarially where all the tenant's worst possible bursts happen simultaneously. While an adversarial worst-case assumption is suitable for some environments, it is typically too conservative in making admission decisions. As an example, we analyze three traces from a production server in Fig. 3(a) and show their aggregate behavior in Fig. 3(b). The peak burst in each trace is marked with a horizontal line. Note that bursts are short lived (on the order of seconds), and that these bursts are not caused by diurnal (hourly) trends. In fact, such short-term bursts occur during every hour of our traces and they are known to have a large impact on performance [25]. As DNC performs an adversarial worst-case analysis, it must consider the scenario where each of the peak bursts happen at the same time. But as shown in the aggregate trace, the actual peak is much lower than the adversarial sum of peaks. As a result, DNC's worst-case assumption limits the number of tenants that can be admitted into the system for any given SLOs.

1.6 The case for Stochastic Network Calculus (SNC)

Typical tenants do not seek strict worst-case guarantees. Instead, tenants target tail latency percentiles lower than the 100%, e.g., the 99.9th latency percentile [14]. DNC only supports the 100th percentile (i.e., adversarial worst-case), so given 99.9th percentile SLOs, DNC simply pretends they are 100th percentile SLOs, resulting in admission decisions which are far too conservative.

We therefore instead turn to an emerging branch of probabilistic theory called Stochastic Network Calculus (SNC). SNC provides request latency bounds for any tenant-specified latency



(a) Individual burstiness

(b) Aggregate burstiness

Figure 3: In computing worst-case analysis of the peak for an aggregate traffic trace, one assumes that all three individual traces have their worst peaks at the same time. This is overly conservative.

percentile, e.g., the 99th, 99.9th, or 99.99th latency percentile. By not assuming the adversarial worst-case, we will show that it is possible to admit many more tenants, even for high percentiles (several 9s).

1.7 Our SNC-based system: SNC-Meister

Our new system, SNC-Meister, uses SNC to upper bound request latency percentiles for multiple tenants sharing a network. Admission decisions are thus made for the specific percentile requested by each tenant. We implement and run SNC-Meister on a physical cluster, and our experiments with production traces show that SNC-Meister can support many more tenants than the state-of-the-art systems by considering 99.9th percentile SLOs (see Fig. 1).

This paper makes the following main contributions:

- **Bringing SNC to practice:** SNC is a new theory that has been developed in a theoretic context and has never been implemented in a computer system. Our primary contribution is identifying and overcoming multiple practical challenges in bringing SNC to practice (details in Sec. 4 and Appendix A). For example, it is an open problem how to effectively apply SNC to analyze tail latency in a network, in particular with respect to handling dependencies between tenants. Many approaches using SNC introduce artificial dependencies that make the analysis too conservative. SNC-Meister introduces a novel analysis, which minimizes artificial dependencies and also supports user-specified dependencies between tenants, which has not previously been investigated in the SNC literature. We prove the correctness of SNC-Meister's analysis (Appendix A) and show that SNC-Meister improves the tightness of SNC latency bounds by 2-4$\times$ (Sec. 4).
- **Extensive evaluation:** We implement SNC-Meister and evaluate it on an 18-node cluster running the widely-used memcached key-value store (setup shown in Fig. 4, details in Sec. 5). We compare against three state-of-the-art admission control systems, two of which we enhance to boost their performance[1]. Across 100 experiments each with 180 tenants represented by

[1]Silo++ admits 10% more tenants than a hand-tuned Silo baseline, and QJump++ admits 5$\times$ more tenants than a

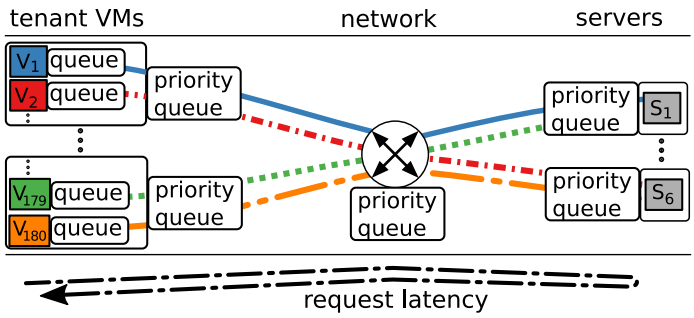


Figure 4: SNC-Meister meets tail latency SLOs on the complete request path for the network shown. The request latency spans the time from when a request is issued until it is fulfilled. Our evaluation experiments involve 180 tenant VMs (on 12 nodes), which replay recent production traces, and six servers running memcached.

recent production traces, SNC-Meister is able to support on average 75% more tenants than the enhanced state-of-the-art systems (Fig. 1) while meeting all SLOs. This improvement means that SNC-Meister allows tenants to transfer 88% more bytes in the median (Sec. 6.1). SNC-Meister is also within 7% of an empirical offline maximum, which we determined through trial-and-error experiments (Sec. 6.2).

- **Building a deployable system:** We design SNC-Meister to operate in existing infrastructures alongside best effort tenants without requiring kernel, OS, or application changes. To simplify user adoption, SNC-Meister only requires high-level user input (e.g., SLO) and automatically generates SNC models and corresponding configuration parameters. Our representation of SNC in code is simple and efficient, which results in the ideal linear scaling of computation time in terms of the number of tenants.

The rest of this paper is organized as follows. The SNC-Meister admission system is described in Sec. 2. Sec. 3 introduces background on SNC and Sec. 4 introduces the challenges solved by SNC-Meister in bringing SNC to practice. We describe our experimental setup, including the enhanced state-of-the-art admission systems, in Sec. 5 and present our main results in Sec. 6. We discuss our results in a brief conclusion in Sec. 8. Details of SNC-Meister's analysis algorithm and corresponding correctness proofs are given in Appendix A.

2 SNC-Meister's admission control process

This section describes SNC-Meister's process in determining admission. When a new tenant seeks admission, it provides its desired tail latency SLO (e.g., $T99 < 50ms$) and a trace of the tenant's requests that represents the burstiness and load added by the tenant. Traces consist of a sequence of request arrival times and sizes, and they can be extracted from historical logs or captured on the fly. Using traces simplifies the burden of having users specify many complex parameters to describe

hand-tuned QJump baseline.

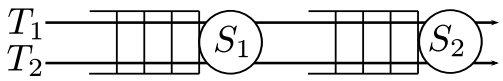


Figure 5: Example network with two tenants T_1 and T_2 flowing through two queues S_1 and S_2.

their traffic.

To understand the implications when selecting a representative trace, we first need to consider the differences between short-term burstiness and long-term load variations. Short-term burstiness denotes sub-second variations of a tenant's bandwidth requirements. Long-term load variation denotes trends over the course of hours, such as diurnal patterns. While both types of variation can affect latency, tail latency is mainly caused by transient network queues due to short-term burstiness. In fact, in our production traces (to be described in Sec. 5), short-term peaks have a rate that is $2\times$ to $6\times$ higher than the average rate, which is higher than the difference between day-hour rates to night-hour rates (less than $2\times$). This short-term burstiness leads to tail latency SLO violations even under low load: in our experiments, SLO violations occurred for network utilizations as low as 40%.

Due to the significance of short-term burstiness, SNC-Meister requires only a short trace segment. This trace segment can be taken from the peak hour of the previous day, or can be periodically updated throughout the day. In our experiments in Sec. 6, we find that 15min trace segments are sufficient to characterize a tenant's load and short-term burstiness.

After having received the tenant trace, SNC-Meister determines admission through the following three steps. First, SNC-Meister analyzes the tenant's trace to derive a statistical characterization understood by the SNC theory (see Sec. 4.3). Second, SNC-Meister assigns a priority to the tenant based on its SLO where the highest priorities are assigned to tenants with the tightest SLOs. We opt for this simple prioritization scheme since our experiments with a more complex prioritization scheme [55] show similar results. Third, SNC-Meister's SNC algorithm calculates the latency for each tenant using the SNC theory (see Sec. 4.1)). We then check if each tenant's predicted latency is less than its SLO. If the previously admitted tenants and the new tenant all meet their SLOs, then the new tenant is admitted at its priority level. Otherwise, the tenant is rejected and can only run at the lowest priority level as best-effort traffic.

SNC-Meister enforces its priorities both in switches and at the end-hosts. To enforce priority at the end-hosts, SNC-Meister configures the HTB queueing module in the Linux Traffic Control interface. To enforce priority at the network switch, we use the Differentiated Services Code Point (DSCP) field (aka TOS IP field) and mark priorities in each packet's header with the DSMARK module in the Linux Traffic Control interface. Our switches support 7 levels of priority for each port; using this functionality simply requires enabling DSCP support in our switches.

3 Stochastic Network Calculus background

At the heart of SNC-Meister is the Stochastic Network Calculus (SNC) calculator. SNC is a mathematical toolkit for calculating upper bounds on latency at any desired percentile (e.g., 99th percentile). This is in contrast to DNC which computes an upper bound on the worst-case latency

(i.e., 100th percentile). In Sec. 3.1, we explain the core concepts of SNC by way of example (Fig. 5). In Sec. 3.2, we explain the necessary mathematical details needed to implement SNC.

3.1 SNC core concepts

SNC is based on a set of operators that manipulate probabilistic distributions. We refer to these distributions as *arrival processes* (A_1 and A_2 for tenants T_1 and T_2 in Fig. 5) and *service processes* (S_1 and S_2 in Fig. 5). One of the main results from SNC is a *latency operator* for taking an arrival process (e.g., A_1), a service process (e.g., S_1), and a percentile (e.g., 0.99), and calculating a tail latency. We write this as $Latency(A_1, S_1, 0.99)$. The latency operator works for any arrival and service process. In our Fig. 5 example, A_1 does not experience a service process S_1 since there is congestion introduced by A_2. Rather, A_1 experiences the leftover (aka residual) service process after accounting for A_2. In SNC, this is handled by the *leftover operator*, $\ominus$. In our example, we get a new service process $S_1' = S_1 \ominus A_2$. Then we can apply the latency operator with S_1' (i.e., $Latency(A_1, S_1', 0.99)$) to get T_1's 99th percentile latency at the first queue.

Moving to the second queue in Fig. 5, we now need arrival processes at the second queue. This is precisely the output (aka departure) process from the first queue. In SNC, this is handled by the *output operator*, $\oslash$. In our example, we calculate A_1's output process A_1' as $A_1' = A_1 \oslash S_1'$ where S_1' is the service process that A_1 experiences at the first queue as defined above. A_2's output process A_2' is calculated similarly. We can then calculate T_1's latency at the second queue as $Latency(A_1', S_2 \ominus A_2', 0.99)$.

To calculate T_1's total latency, we can add up the latencies from each queue (i.e., $Latency(A_1, S_1', 0.99) + Latency(A_1', S_2 \ominus A_2', 0.99)$). However, this is not a 99th percentile latency anymore. To get a 99th percentile overall latency, we need to use higher percentiles for each queue (e.g., 99.5th percentile) [2]. There are in fact many options for percentiles at each queue (e.g., 99.5 & 99.5, 99.3 & 99.7, 99.1 & 99.9) for calculating an overall 99th percentile latency. Choosing the option that provides the best latency bound is time consuming, so SNC provides a *convolution operator*, $\otimes$, which avoids this problem by treating a series of queues as a single queue with a merged service process. In our example, we can apply the convolution operator as $S_1' \otimes (S_2 \ominus A_2')$. We then use this new service process to calculate the latency as $Latency(A_1, S_1' \otimes (S_2 \ominus A_2'), 0.99)$.

Lastly, SNC has an *aggregation operator*, $\oplus$, which calculates the multiplexed arrival process of two tenants. For example, the aggregate operator can be used to analyze the multiplexed behavior of T_1 and T_2 as $A_1 \oplus A_2$.

The SNC theory provides this set of operators along with proofs of correctness. For a recent in-depth introduction to the SNC operators, see the recent survey [18]. Unfortunately, however, SNC has rarely been used to analyze realistic network topologies. For example, almost all prior work in the SNC theory focuses on simple line networks [9, 16, 22, 33, 6, 10, 5, 18]. It remains currently an open question how to practically put these operators together to obtain an accurate latency analysis in complex networks. We will describe in Sec. 4 the challenges we face in bringing SNC to a typical cloud data center network and the approach taken by SNC-Meister to obtain an accurate latency analysis.

[2] This is formally known as the union bound.

Purpose	$\rho(\cdot)$	$\sigma(\cdot)$
Arrival process A for MMPP with transition matrix Q and diagonal matrix $E(\theta)$ of each state's MGF	$\rho_A(\theta) = sp(E(\theta)\ Q)$	$\sigma_A(\theta) = 0$
Service process S for network link with bandwidth R	$\rho_S(\theta) = -R$	$\sigma_S(\theta) = 0$
Leftover operator $\ominus$ for service process S and arrival process A	$\rho_{S \ominus A}(\theta) = \rho_A(\theta) + \rho_S(\theta)$	$\sigma_{S \ominus A}(\theta) = \sigma_A(\theta) + \sigma_S(\theta)$
Output operator $\oslash$ for service process S and arrival process A	$\rho_{A \oslash S}(\theta) = \rho_A(\theta)$	$\sigma_{A \oslash S}(\theta) = \sigma_A(\theta) + \sigma_S(\theta) - \frac{1}{\theta}\log\left(1 - e^{\theta(\rho_A(\theta)+\rho_S(\theta))}\right)$
Aggregate operator $\oplus$ for arrival process A_1 and arrival process A_2	$\rho_{A_1 \oplus A_2}(\theta) = \rho_{A_1}(\theta) + \rho_{A_2}(\theta)$	$\sigma_{A_1 \oplus A_2}(\theta) = \sigma_{A_1}(\theta) + \sigma_{A_2}(\theta)$
Convolution operator $\otimes$ for service process S_1 and service process S_2	$\rho_{S_1 \otimes S_2}(\theta) = \max\{\rho_{S_1}(\theta), \rho_{S_2}(\theta)\}$	$\sigma_{S_1 \otimes S_2}(\theta) = \sigma_{S_1}(\theta) + \sigma_{S_2}(\theta) - \frac{1}{\theta}\log\left(1 - e^{-\theta\lvert\rho_{S_1}(\theta)-\rho_{S_2}(\theta)\rvert}\right)$
Tail latency L for percentile p, arrival process A, and service process S	$L = \min_{\theta} \frac{1}{\theta\rho_S(\theta)} \log\Big((1-p) * \big(1 - \exp(\theta * (\rho_A(\theta) + \rho_S(\theta)))\big)\Big) - \frac{1}{\rho_S(\theta)}(\sigma_A(\theta) + \sigma_S(\theta))$	

Table 1: The SNC operators and equations used by SNC-Meister.

3.2 Mathematics behind SNC

In this section, we expand upon the high level description of the SNC concepts in Sec. 3.1 and describe the mathematics behind SNC. To begin, we need to represent arrival processes. We write the arrival process for tenant T_1 as $A_1(m, n)$, which represents the number of bytes added by T_1 between time m and n. As arrival processes are probabilistic in nature, SNC is based on moment generating functions (MGFs), which are an equivalent representation of distributions. Directly working with MGFs is unfortunately quite challenging mathematically, so SNC operates on an upper bound on the MGF, parameterized by two subcomponents $\rho(\theta)$ and $\sigma(\theta)$. For example, the MGF of $A_1(m, n)$, written $MGF_{A_1(m,n)}(\theta)$, is upper bounded by:

$$MGF_{A_1(m,n)}(\theta) \leq e^{\theta(\rho_{A_1}(\theta)(n-m)+\sigma_{A_1}(\theta))} \quad \forall\theta > 0$$

The parameter θ ensures that all moments of the distribution of $A_1(m, n)$ are covered. By using this standardized form, all arrival processes are defined by the two subcomponents $\rho(\theta)$ and $\sigma(\theta)$, and all SNC operators provide equations for these subcomponents (Tbl. 1).

To calculate the $\rho_{A_1}(\theta)$ and $\sigma_{A_1}(\theta)$ for T_1, we need to assume a stochastic process for T_1, such as a Markov Modulated Poisson Process (MMPP). An MMPP is useful for representing time-variant

(bursty) arrival rates (see Sec. 4.4). For example, a 2-MMPP switches between high-rate phases and low-rate phases using a Markov process. The MMPP's transition matrix is given by Q, which for a 2-MMPP has four entries:

$$Q = \begin{pmatrix} p_{hh} & p_{hl} \\ p_{lh} & p_{ll} \end{pmatrix}$$

where, e.g., p_{hl} indicates the probability that after a high-rate phase (h) we next switch to a low-rate phase (l). The distribution of the arrival rate and request size for each phase is captured in the matrix E, which is a diagonal matrix of the MGF for each phase:

$$E(\theta) = \begin{pmatrix} MGF_h(\theta) & 0 \\ 0 & MGF_l(\theta) \end{pmatrix}$$

We can now calculate the $\rho_{A_1}(\theta)$ and $\sigma_{A_1}(\theta)$ for T_1 as:

$$\rho_{A_1}(\theta) = sp(E(\theta) \cdot Q) \text{ and } \sigma_{A_1}(\theta) = 0$$

where $sp(\cdot)$ is the spectral radius of a matrix. This is proved in SNC, and we list this equation in Tbl. 1.

Service processes are defined similarly to arrival processes with the same two subcomponents $\rho(\theta)$ and $\sigma(\theta)$. Rather than working with lower bounds on the amount of service provided, SNC works with an upper bound:

$$MGF_{S_1(m,n)}(-\theta) \leq e^{\theta(\rho_{S_1}(\theta)(n-m)+\sigma_{S_1}(\theta))} \quad \forall \theta > 0$$

where the MGF has an extra negative sign on the θ parameter, which transforms a lower bound into an upper bound. For lossless networks, the $\rho_{S_1}(\theta)$ and $\sigma_{S_1}(\theta)$ have a simple form:

$$\rho_{S_1}(\theta) = -R \text{ and } \sigma_{S_1}(\theta) = 0$$

where R is the bandwidth of the network link.

Lastly, we need to deal with the arrival and service processes generated by the SNC operators described in Sec. 3.1. Fortunately, SNC has derived and proved equations for $\rho(\theta)$ and $\sigma(\theta)$ for each of the SNC operators, and they can be found in Tbl. 1. Formal definitions of the assumptions of SNC, of each operator, and of the latency equation can be found in the Appendix A.1.

In summary, the analysis of a network with SNC requires three conception steps: 1) create an arrival bound for each tenant, 2) calculate the service available for each tenant by chaining together the equations in Tbl. 1, and 3) calculate the tail latency for each tenant using the latency equation (last line in Tbl. 1). We will describe in Sec. 4.4 how we represent arrival and service processes in code, and how we evaluate the latency equation with the θ parameter.

4 Challenges in bringing SNC to practice

As we are the first to implement SNC in practice in a computer system, we face multiple practical challenges. Our primary contribution in this work is identifying these challenges and demonstrating how to overcome them.

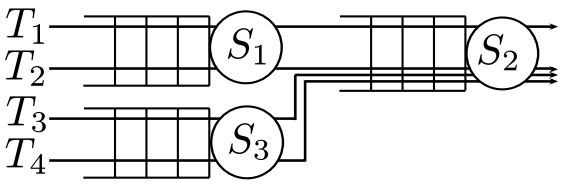


Figure 6: Extending Fig. 5's example with tenants T_3 and T_4 flowing through queues S_3 and S_2.

First, SNC is a new theory, and it is currently an open problem how to apply SNC effectively to a network. SNC theorists are primarily concerned with the theorems and proofs behind the SNC operators, but haven't studied how to practically put together the operators to analyze networks. Sec. 4.1 describes how we analyze networks.

Second, SNC theorists typically assume that all tenants are independent of each other, which can be overly optimistic. SNC-Meister allows users to specify dependencies between subsets of tenants. Sec. 4.2 discusses the effect of dependency on latency.

Third, real traffic exhibits bursty behavior, particularly at second/sub-second granularity, and we need to capture this behavior to properly characterize tail latency. SNC theorists, however, don't pay attention to modeling tenant behavior. Instead, they assume that tenant arrival processes have already been stochastically characterized and focus their attention on providing SNC operators that operate on these pre-specified distributions. However, to use SNC in practice, we need to figure out how to build stochastic characterizations that can represent the burstiness exhibited by tenants. In Sec. 4.3, we identify a reasonable model that is practical to compute using parameters that can efficiently be extracted from trace data.

Fourth, SNC works with full representations of probabilistic distributions to properly calculate tail latency. SNC theorists can easily write this down mathematically via equations operating on distributions, but to actually implement an SNC calculator, we need representations in code. Sec. 4.4 describes our simple symbolic representation and how we work with SNC in code.

4.1 Analyzing networks with SNC-Meister

The biggest challenge we face in bringing SNC to practice is building an algorithm for combining the SNC operators (described in Sec. 3.1) to analyze networks. Even with the simple example in Fig. 5, there are multiple ways to analyze the latency for T_1. For example, Sec. 3.1 describes how the latency can be analyzed one queue at a time (i.e., $Latency(A_1, S_1', 0.995) + Latency(A_1', S_2 \ominus A_2', 0.995)$) as well as through a convolution operator (i.e., $Latency(A_1, S_1' \otimes (S_2 \ominus A_2'), 0.99)$). Yet there is even another approach by first applying the convolution operator on S_1 and S_2 before accounting for the congestion from A_2 (i.e., $Latency(A_1, (S_1 \otimes S_2) \ominus A_2, 0.99)$). While each approach is correct as an upper bound on tail latency, they are not equally tight. One of our key findings is that some approaches can introduce artificial dependencies, which eliminate a lot of SNC's benefit. For example, in $Latency(A_1', S_2 \ominus A_2', 0.995)$, A_1' $(= A_1 \oslash (S_1 \ominus A_2))$ and A_2' $(= A_2 \oslash (S_1 \ominus A_1))$ are stochastically dependent because they are related by A_1, A_2, and S_1. Likewise, the convolution $S_1' \otimes (S_2 \ominus A_2')$ has an artificial dependency because S_1' $(= S_1 \ominus A_2)$ and A_2' both are related by

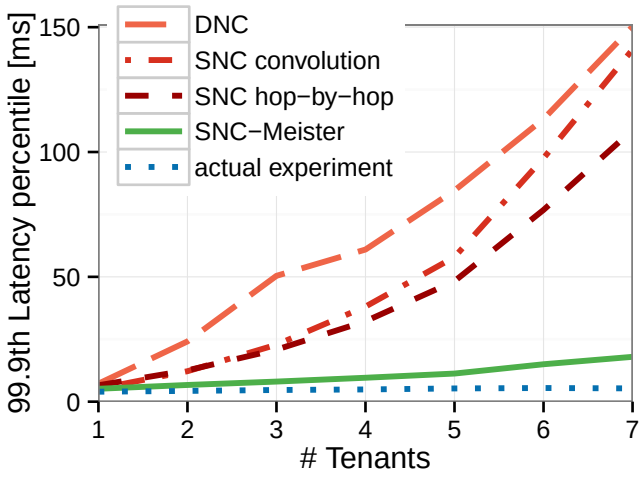


Figure 7: The tail latency calculated by DNC and multiple SNC methods, SNC convolution [16], SNC hop-by-hop [5], and SNC-Meister. In this micro-experiment, we vary the number of tenants connecting from a single client to a single server through two queues.

S_1 and A_2. In reality, there shouldn't be any dependencies between A_1, A_2, S_1, and S_2, but the ordering of SNC operators can introduce these artificial dependencies.

In our SNC-Meister SNC algorithm, we identify two key ideas that allow us to eliminate artificial dependencies.

Key idea 1. *When analyzing T_1, SNC-Meister performs the convolution operator before the leftover operator for any tenants sharing the same path as T_1. For example, $Latency(A_1, (S_1 \otimes S_2) \ominus A_2, 0.99)$.*

In our Fig. 5 example, this avoids the artificial dependencies at the second queue. However, this is not the only source of artificial dependencies. Fig. 6 shows a slightly more complex scenario with additional traffic from T_3 and T_4. To calculate T_1's latency, we now need to account for the effect of T_3 and T_4 at the second queue S_2. The straightforward approach is to apply the output operator on A_3 and A_4 to get arrival processes A'_3 $(= A_3 \oslash (S_3 \ominus A_4))$ and A'_4 $(= A_4 \oslash (S_3 \ominus A_3))$ at the second queue. However, this introduces a stochastic dependency between A'_3 and A'_4 because they are related by S_3, A_3, and A_4.

Key idea 2. *When handling competing traffic from the same source, SNC-Meister applies the aggregate operator before the output operator. For example, $(A_3 \oplus A_4) \oslash S_3$.*

This aggregate flow to the second queue now does not have any artificial dependencies. Combining these ideas for our Fig. 6 example, we can calculate the latency of T_1 as $Latency(A_1, (S_1 \otimes (S_2 \ominus ((A_3 \oplus A_4) \oslash S_3))) \ominus A_2, 0.99)$. Through these two ideas, SNC-Meister is able to produce much tighter bounds (see Fig. 7) than the two SNC analysis approaches used by prior SNC literature. The first prior approach is called SNC convolution and works by first calculating the

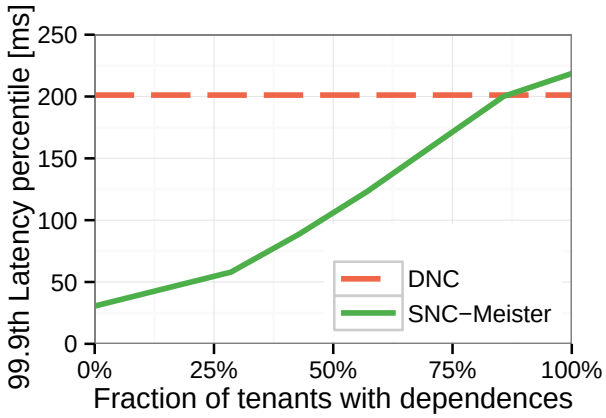


Figure 8: The tail latency calculated by DNC and SNC-Meister as we vary the fraction of tenants that are dependent on each other. In this micro-experiment, we have seven identical tenants connecting from a single client to a single server and indicate to SNC-Meister that a fraction of them are dependent on each other.

left-over service at each queue and then applies the convolution across the resulting leftover services [9, 16, 22, 33, 6, 10, 5, 18]. The second prior approach is called SNC hop-by-hop [5] because it analyzes one queue at a time. Examples and formal definitions of these approaches are given in Appendix A.3. Fig. 7 shows that the analysis used by SNC-Meister is close to the tail latency measured in an actual experiment, whereas SNC convolution and SNC hop-by-hop are $2-4\times$ less accurate.

A formal description of SNC-Meister's SNC algorithm and its proof of correctness can be found in Appendix A.4 and A.5.

4.2 Dependencies between tenants

Since not all tenants are necessarily independent, SNC-Meister also supports users specifying dependencies between tenants. This is useful in scenarios, for example, where multiple tenants are part of the same load balancing group. Unfortunately, generally assuming tenants are dependent when applying SNC leads to very conservative latencies, eliminating our benefit over DNC. We resolve this issue in SNC-Meister by tracking dependency information with arrival and service processes and only accounting for dependencies when SNC operators encounter dependent arrival/service processes.

Fig. 8 shows the effect of tenant dependency on latency. In this experiment, we take a fraction of the tenants and mark them as dependent on each other. As this fraction varies from 0% (i.e., all independent) to 100% (i.e., all dependent), we see the latency calculated by SNC-Meister increases. This is expected since it is more likely that a group of dependent tenants will be simultaneously bursty. Nevertheless, SNC-Meister's latency is almost always[3] under DNC since it always assumes

[3]SNC-Meister can generate higher latencies than DNC when nearly all tenants are dependent because the SNC

dependent behavior.

4.3 Modeling tenant burstiness

Properly characterizing tail latency entails representing the burstiness and load that each tenant contributes. In SNC-Meister, we find that a Markov Modulated Poisson Process (MMPP) is a flexible and efficient model for burstiness. An MMPP can be viewed as a set of phases with different arrival rates and a set of transition probabilities between the phases. A phase with high arrival rate can represent a bursty period, while a phase with low arrival rate represents a non-bursty period. The MMPP is efficient in that the number of phases can be increased to reflect additional levels of burstiness.

The MMPP parameters for each tenant are determined from its trace. The traces contain the arrival times of requests and their sizes, where the size of a request is the number of bytes being requested. In SNC-Meister, the trace analysis is automated by our Automated Trace Analysis (ATA) component. The ATA first determines the number of MMPP phases needed to represent the range of burstiness in the trace. We use an idea similar to [24] where each phase is associated with an arrival rate and covers a range of arrival rates plus or minus two standard deviations. The ATA then maps time periods in the trace to MMPP phases and empirically calculates transition probabilities between the MMPP phases.

While SNC-Meister adapts to the range of burstiness on a per-tenant basis using multiple MMPP phases, the specific number of phases is not critical. In our experimentation, we find a big difference going from a single phase (i.e., a standard Poisson Process) to two phases, but less of a difference with more than two phases. If computation speed is a limiting factor, it is possible to tune SNC-Meister to compute latency faster using fewer phases.

4.4 Representing SNC in code

The core building blocks in SNC are arrival and service processes. In this section, we'll demonstrate how SNC-Meister represents arrival and service processes as objects in code. We'll first show how to put together the SNC operators by walking through the example in Fig. 5 and then delve into details on how SNC operators are represented internally.

To analyze the example in Fig. 5, we start with two arrival processes for A_1 and A_2 and two service processes for S_1 and S_2:

```
ArrivalProcess* A1 = new MMPP(traceT1);
ArrivalProcess* A2 = new MMPP(traceT2);
ServiceProcess* S1 = new NetworkLink(bandwidth);
ServiceProcess* S2 = new NetworkLink(bandwidth);
```

We now proceed to calculate the latency of T_1, mathematically written $Latency(A_1, (S_1 \otimes S_2) \ominus A_2, 0.99)$. First, we'll need to create a service process for the convolution of S_1 and S_2 (i.e., $S_1 \otimes S_2$), which is yet another service process (named `S1x2`):

```
ServiceProcess* S1x2 = new Convolution(S1, S2);
```

equations are not tight upper bounds, whereas our DNC analysis is tight.

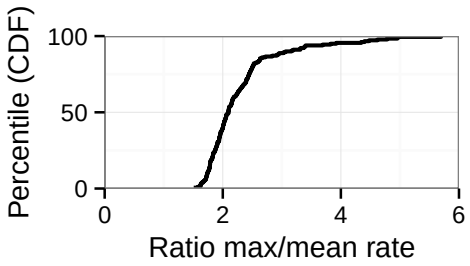


Figure 9: The ratio between maximum and mean request rate per second is high for many of our traces, which indicates high burstiness.

To get the final service process for A_1, we need to take `S1x2` and calculate the leftover service process after accounting for A_2 (i.e., $(S_1 \otimes S_2) \ominus A_2$):

```
ServiceProcess* S1x2_A2 = new Leftover(S1x2, A2);
```

Finally, we calculate the 99th percentile latency of T_1 by:

```
double L_A1 = calcLatency(A1, S1x2_A2, 0.99);
```

SNC-Meister is designed to allow the SNC operators to compose any algebraic expression (e.g., $(S_1 \otimes S_2) \ominus A_2$ is `new Leftover(new Convolution(S1, S2), A2)`). This is accomplished by having all of our operators as subclasses of the ArrivalProcess and ServiceProcess base classes, which have a standardized representation using the $\rho(\theta)$ and $\sigma(\theta)$ form (see Sec. 3.2). To symbolically represent these $\rho(\theta)$ and $\sigma(\theta)$ functions in code, the base classes define pure virtual functions for `rho` and `sigma` that every operator overrides with the equations in Tbl. 1.

There is one more detail we need to take care of. The `calcLatency` function from above has an extra parameter called θ that we haven't mentioned. Each value of the θ parameter produces an upper bound on the latency percentile [18]. We can thus improve the accuracy of the latency prediction by optimizing θ to get the minimal upper bound on the latency. This search over θ values can only be executed after combining all SNC functions, so we add a symbolic θ (`theta`) parameter to the internal representation of all SNC functions, e.g., `calcLatency(A, S, 0.99, theta)`. After combining all SNC functions, SNC-Meister searches for the optimal θ starting at a coarse granularity (e.g., $\theta = 1, 2, 3, ..., 10$) and then progressively narrowing down to finer granularities (e.g., $\theta = 2.1, 2.2, ..., 2.9$).

5 Experimental setup

To demonstrate the effectiveness of SNC-Meister in a realistic environment, we evaluate our implementation of SNC-Meister and of three state-of-the-art systems in a physical testbed running memcached as an example application. This section describes the state-of-the-art systems (Sec. 5.1), our physical testbed (Sec. 5.2), our traces (Sec. 5.3), and the procedure of each experiment (Sec. 5.4).

5.1 State-of-the-art Admission Control Systems

The goal of an admission control system is to ensure that all tenants meet their request latency SLO, where the request latency includes both network queueing *and* end-host queueing. As all three state-of-the-art systems rely on some form of rate limiting, end-host queueing delay can be significant and has to be taken into account. Unfortunately, two of the three systems (Silo and QJump) do not account for end-host queueing, which is left to the tenant since the tenant is also responsible for selecting rate limits in these systems. As this is not practical for experimentation, we create enhanced versions (Silo++ and QJump++) that both 1) compute the effect on end-host queueing based on the rate limits and 2) automatically select rate limit parameters to attempt to maximize the number of admitted tenants.

Silo [27]: Silo offers tenants a worst-case packet latency guarantee under user-specified rate limits. Admission control is performed with DNC by checking that no switch queue in the network overflows. The maximum packet latency is calculated by adding up all maximum queue sizes along a packet's path.

A limitation with Silo is that the non-trivial problem of choosing a rate limit (i.e., bandwidth and maximum burst size) is left to the user. In the Silo experiments, the burst size is fixed to 1.5KB, but the bandwidth is chosen by trial and error. A small bandwidth (e.g., the mean rate of a tenant) entails a high end-system queueing delay due to being slowed down by the rate limiting. Because Silo focuses on packet latency and does not incorporate the end-system queueing delay into its admission decisions, selecting too small of a bandwidth leads to SLO violations for the total request latency. On the other hand, selecting too high a bandwidth causes very few tenants to be admitted.

Silo++: We extend Silo with an algorithm to automatically choose the minimal bandwidth so that each tenant's request latency SLO can be guaranteed. This is achieved by profiling each tenant's traffic requirements using the DNC effective bandwidth theory [31]. We then model the end-system queueing delay using DNC, add in Silo's packet latency guarantee, and check whether the tenant can meet its SLO.

QJump [23]: QJump offers multiple classes of service with different latency-throughput trade-offs. The first class receives the highest priority along with a worst-case latency guarantee based on a variant of DNC [37, 38], but is aggressively rate limited. For the other classes, tenants are allowed to send at higher rates, but at lower priorities and without any latency guarantee. There are two limitations in employing the original QJump proposal: 1) tenants do not know which class to pick because the respective latency guarantee is unknown in advance, and 2) tenants do not know the end-system queueing delay caused by the rate limiting of each class.

QJump++: We extend QJump with an algorithm to automatically assign tenants to a (near) optimal class. The algorithm iteratively increases the QJump level for tenants that do not meet their SLOs. To check if a tenant meets its SLO, we solve limitation 1 by calculating the latency guarantee offered by each class using DNC theory. We solve limitation 2 by automatically profiling a tenant trace and inferring the end-system delay using DNC theory.

Additionally, we find that instantiating the QJump classes using the QJump equation (Eq. (4) in [23]) severely limits the number of admitted tenants (5x fewer on average). By fixing a set of throughput values independent of the number of tenants, we significantly boost the number of admitted tenants for the QJump system.

PriorityMeister (PM) [55]: PriorityMeister uses DNC to offer each tenant a worst-case request latency guarantee based on rate limits that are automatically derived from a tenant's trace. PriorityMeister automatically configures tenant priorities to meet latency SLOs across both network and storage, and in this work, we tailor it to focus only on network latency.

5.2 Physical testbed

Our physical testbed comprises an otherwise idle, 18 node cluster of Dell PowerEdge 710 servers, configured with two Intel Xeon E5520 processors and 16GB of DRAM. We use the setup shown in Fig. 4. Six servers are dedicated as memcached servers running the most recent version (1.4.25) of memcached. Twelve servers run a set of tenant VM's using the standard kvm package (qemu-kvm-1.0) to provide virtualization support. Each tenant VM runs 64-bit Ubuntu 13.10 and replays a request trace using libmemcached. Each physical node runs 64-bit Ubuntu 12.04 and we use the distribution's default Linux kernel without modifications. The top-of-rack switch connecting the nodes is a Dell PowerConnect 6248 switch, providing 48 1Gbps ports and 2 10Gbps uplinks, with DSCP support for 7 levels of priority.

5.3 2015 production traces

Our evaluation uses 180 recent traces captured in 2015 from the datacenter of a large Internet company. The traces capture cache lookup requests issued by a diverse set of Internet applications (e.g., social networks, e-commerce, web, etc.). Each trace contains a list of anonymized requests parameterized by the arrival time and object size being requested, ranging from 1 Byte to 256 KBytes with a mean of 28 KBytes. Each trace is 30 minutes long and contains 100K to 600K requests, with a mean of 320K requests. We find that these traces exhibit significant short-term burstiness, and Fig. 9 shows that the CDF for the ratio of peak to mean request rates ranges from 2 to 6. We also perform standard statistical tests [36, 1] to verify the stationarity and mutual stochastic independence of our traces as required by SNC.

5.4 Experimental procedure

In our experiments, we run up to 180 tenants that replay memcached requests from each tenant's associated trace. For each experiment, tenants arrive to the system one by one in a random order with a 99.9% SLO drawn uniformly randomly from $\{$10ms, 20ms, 50ms, 100ms$\}$. Having different orders and SLOs for tenants leads to very different experiments where distinct subsets of the 180 tenants are being admitted in each experiment. When a tenant arrives, the admission system makes its decision based on the tenant's SLO and the first half of the tenant's trace (15 mins). After the admission decisions for all 180 tenants have been made, each tenant starts a dedicated VM (Sec. 5.2), which replays the second half of its request trace (15 mins). All tenants replay their traces in an open loop fashion, which properly captures the end-to-end latency and the effects of end-system queueing [43]. All admission systems meet the tenant SLOs, as verified by monitoring the total memcached request latency for every request (i.e., completion time - arrival time in the trace) and checking that the 99.9% latency across 3min time intervals for each tenant is less than

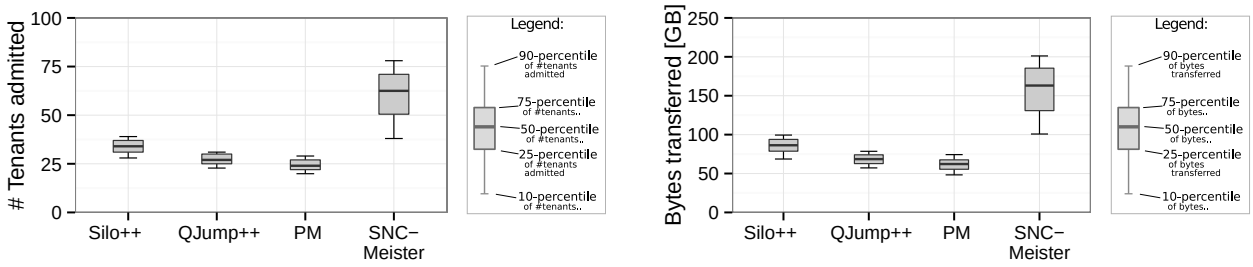


Figure 10: Comparison of three state-of-the-art admission control systems to SNC-Meister for 100 randomized experiments. In each experiment, 180 tenants, each submitting hundreds of thousands of requests, arrive in random order and seek a 99.9% SLO randomly drawn from {10ms, 20ms, 50ms, 100ms}. The left box plot shows that across the 100 experiments, all percentiles on the number of tenants admitted are higher for SNC-Meister than for any other system. The right plot shows that SNC-Meister achieves a similar improvement with respect to the volume of bytes transferred in each experiment.

its SLO. Thus, we evaluate the performance of the admission control systems under the following two metrics: 1) the number of tenants admitted by each system, and 2) the total volume of bytes transmitted by admitted tenants. Metric 1 indicates how many tenants with tail latency SLOs can be concurrently supported by each system. Metric 2 prevents a system from scoring high on metric 1 by admitting only small tenants, which could happen because the tenants have very different request volumes (Sec. 5.3).

6 Results

In this section, we experimentally evaluate the performance and practicality of SNC-Meister. Sec. 6.1 shows that SNC-Meister is able to support 75% more tail latency SLO tenants in the median than state-of-the-art systems across a large range of experiments. SNC-Meister also transfers 88% more bytes in the median, which shows that SNC-Meister supports a much higher network utilization. Sec. 6.2 shows that SNC-Meister's median performance is within 7% of an empirical offline solution. Sec. 6.3 demonstrates that SNC-Meister is able to support low-bandwidth tenants with very tight SLOs alongside high-bandwidth tenants. Sec. 6.4 investigates the sensitivity of the SNC latency prediction to the SLO percentile. Sec. 6.5 evaluates the scalability of SNC-Meister's SNC computation and shows that it scales linearly with the number of tenants.

6.1 SNC-Meister outperforms the state-of-the-art

This section compares SNC-Meister with enhanced versions of the state-of-the-art tail latency SLO systems (described in Sec.5.1). We run 100 experiments each with 180 tenants arriving in a random

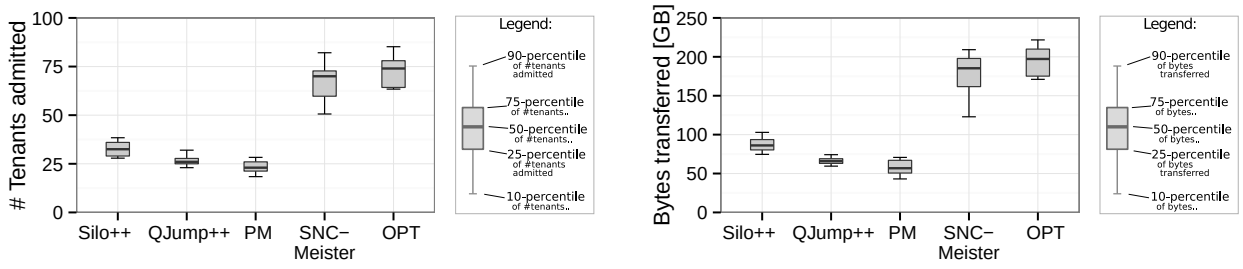


Figure 11: Comparison between state-of-the-art systems, SNC-Meister, and an empirical optimum (OPT) for 10 of the 100 experiments in Fig. 10. The left box plot shows that the number of tenants admitted by SNC-Meister is close to OPT, whereas the other systems admit less than half of OPT. The right plot shows that SNC-Meister is also close to OPT with respect to the volume of bytes transferred in each experiment.

order with random SLOs (described in Sec. 5.4). All four systems, including SNC-Meister, meet the latency SLOs for all tenants, but differ in how many tenants each system admits.

Fig. 10 shows a box plot of the number of admitted tenants and a box plot of the volume of transferred bytes. We see that the three state-of-the-art systems (Silo++, QJump++, PriorityMeister) perform roughly the same as they draw upon the same underlying DNC mathematics. SNC-Meister achieves a significant improvement over all three systems across all percentiles of admitted tenants and bytes transferred. At a more detailed look, Silo++ admits more than QJump++ and PriorityMeister, which is caused by the effective bandwidth enhancement of Silo++ (see Sec. 5.1). Nevertheless, SNC-Meister outperforms Silo++ by a large margin: of the 100 experiments, the 10-percentile of SNC-Meister is above the 75-percentile of Silo++ for both the number of admitted tenants and bytes transferred. The fact that SNC-Meister performs well for both metrics shows that SNC-Meister's improvement is not just due to admitting more small tenants, but actually allowing higher utilization.

6.2 Comparison to empirical optimum

To evaluate how well SNC-Meister compares to an empirical optimum, we determine the maximum number of tenants that can be admitted without SLO violations (labeled OPT) via trial and error experiments. In order to make finding OPT feasible, OPT only considers tenants in the order that they arrive and determines the maximum number of tenants we can admit until introducing SLO violations. Determining OPT via trial and error is time consuming and hence we only do this for a random subset[4] of 10 out the 100 experiments from Sec. 6.1.

Fig. 11 compares the state-of-the-art, SNC-Meister and OPT. We find that SNC-Meister is close to OPT across all percentiles of admitted tenants and bytes transferred. Specifically, SNC-Meister is

[4]Note that the results from the 10 experiments in Fig. 11 are representative because the state-of-the-art systems and SNC-Meister perform similarly to the 100 experiments in Fig. 10.

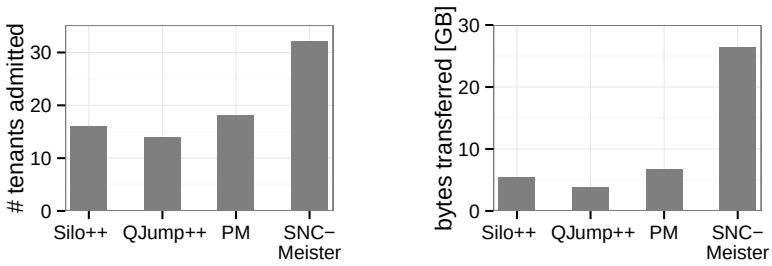


Figure 12: Number of admitted tenants (left) and bytes requested by admitted tenants (right) for two groups of tenants: a set of small-request low-latency (4ms) tenants and a set of large-request higher-latency (50ms) tenants. SNC-Meister again admits significantly more tenants and more than three times as many bytes as any of the state-of-the-art systems.

within 7% of OPT in the median, whereas the state-of-the-art admission systems achieve only half of OPT. Thus SNC-Meister captures most of the statistical multiplexing benefit without needing to run trial and error experiments.

6.3 Small-request tenants

While we have focused on request latency, many related works focus on packet latency and the effects on small requests (i.e., single packet-sized requests).

As SNC-Meister supports prioritization (Sec. 2), we demonstrate that SNC-Meister can also support tenants with small requests and very tight SLOs. Fig. 12 shows the results from an experiment with a set of eleven tenants with single packet requests and tight SLOs (4ms) along with twenty-one other tenants with larger requests and higher SLOs (50ms). Like before, we see that SNC-Meister is able to admit many more tenants than the state-of-the-art systems. Here, PriorityMeister does better than Silo++ and QJump++ since it does not need to reserve a lot of bandwidth for the tight SLOs. Nevertheless, all three of these state-of-the-art systems suffer from the drawbacks of DNC and are unable to admit many of the large-request tenants once they've admitted the small-request tenants with tight SLOs. This can particularly be seen in the graph of the number of bytes transferred by admitted tenants. SNC-Meister tenants send a lot more traffic since SNC-Meister admits both the small-request tenants as well as many more large-request tenants. SNC-Meister is able to do so since, probabilistically, our small-request tenants don't have a large effect on the large-request tenants. The large-request tenants end up with a lower priority due to their higher SLOs and thus do not affect the small-request tenants.

6.4 Tail latency percentiles

One might wonder how SNC-Meister performs for latency SLOs other than the 99.9th percentile. We address this question by comparing SNC-Meister's latency prediction to the DNC latency

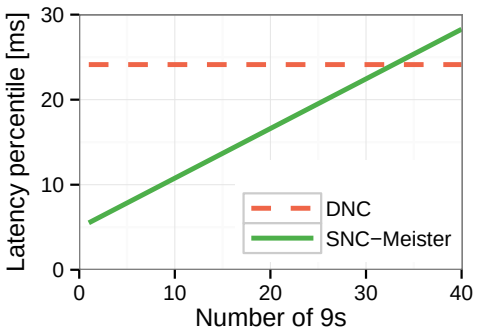


Figure 13: Comparison between the latency predictions of SNC-Meister and DNC for different SLO percentiles. Specifically, the x-axis denotes the number of 9s, where three 9s represents the 99.9th percentile. As expected, the latency using SNC increases with the SLO percentile, but is still superior to DNC even with thirty 9s.

prediction used by state-of-the-art systems. Lower and more accurate latency predictions allow SNC-Meister to admit more tenants than DNC. We consider how SNC-Meister's latency prediction varies with the SLO percentile.

Fig. 13 shows the latency prediction of SNC-Meister and DNC vs. the number of 9s in the SLO percentile, where three 9s represents the 99.9th percentile. SNC-Meister's latency increases with the SLO percentile, as expected, and only exceeds the DNC latency with thirty-three 9s. DNC's worst-case analysis is conservative in accounting for rare events that probabilistically should never occur. SNC gains most of its advantage by working with lower SLO percentiles that are unaffected by these rare, probabilistically improbable events.

6.5 Scalability of computation

In this section, we study the scalability of SNC-Meister's computation. In Fig. 14, we show the runtime for computing latency bounds as a function of the number of tenants. We see that SNC-Meister's runtime scales linearly with the number of tenants, which is ideal since each tenant's latency is calculated one by one. This is very promising, given that the computation is currently single threaded, and the analysis of each of the tenants can easily be parallelized.

7 Related work

SNC-Meister addresses tail latency SLOs, which is an active research area with a rich literature. The related work can be divided into four major lines of work, and is summarized in Tbl. 2. First, there is a body of work that ensures that tail latency SLOs are met using worst-case latency bounds; unfortunately these works are unable to achieve high degrees of multi-tenancy. To overcome these limitations, theoreticians have developed a second line of work that provides probabilistic tail latency

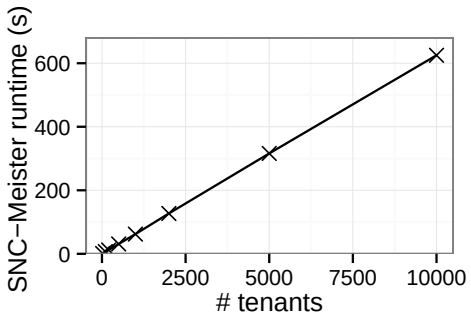


Figure 14: SNC-Meister's runtime scales linearly with the number of tenants.

bounds via Stochastic Network Calculus (SNC); unfortunately this work is entirely theoretical and has never been implemented in any computer system. Third, there is a body of work that proposes techniques for significantly reducing the tail latency; unfortunately, ensuring that request latency SLOs are met is not within the scope of that work. Fourth, there are some recent systems that try to meet SLOs based on measured latency; unfortunately, they aren't suited for admission control and don't cope well with bursty tenants.

Guaranteed Latency Systems

There are three recent state-of-the-art systems that provide SLO guarantees: QJump [23], Silo [27], and PriorityMeister [55], described in detail in Sec. 5.1. All three systems, however, are designed for worst-case latency guarantees. For tenants seeking a lower percentile tail guarantee (e.g., a guarantee on the $99.9th$ percentile of latency), these systems are overly conservative in their admission decisions: they admit less than half of the number of tenants as compared to SNC-Meister (see Sec. 6).

Besides these recent proposals, there has been a long history of DNC-based worst-case latency admission control algorithms in the context of Internet QoS [15, 34, 30, 46, 51]. These older proposals are not tailored to datacenter applications, and also suffer from the conservativeness of worst-case latency guarantees.

Stochastic Network Calculus (SNC)

The modern SNC theory evolved as an alternative to the DNC theory to capture statistical multiplexing gains and enable accurate guarantees for any latency percentile [18, 40, 5, 10, 33, 6, 22, 16, 9, 19, 8, 29, 7, 12, 41, 44, 53]. While the SNC community has made significant progress in building a theoretical framework for the analysis of latency tails, we are not aware of any implementations that use SNC in computer systems.

Almost all prior SNC analysis focuses on simple line network topologies [9, 16, 22, 33, 6, 10, 5, 18], in which the classical SNC convolution approach does not introduce artificial stochastic dependencies. This might be the reason that the problem of artificial dependencies, which is a major challenge in data center topologies, has rarely been considered before. In fact, only one work [5] points out the artifical-dependency problem, but does not propose a solution. The problem of user-specified dependencies and how to efficiently aggregate them has also not been considered in prior SNC literature (see Appendix A.3 for details).

			tail latency SLO	multi tenancy	commodity hardware	unmodified OS+apps	tenant parameters
guaranteeing tail latency	SNC-based	SNC-Meister	99.9th (e.g.)	high	yes	yes	automated
	worst-case admission control	QJump [23]	100th	low	yes	yes	manual
		SILO [27]	100th	low	yes	yes	manual
		PriorityMeister [55]	100th	low	yes	yes	automated
reducing tail latency	datacenter scheduling	pHost [20]	no	high	no	yes	manual
		Fastpass [39]	no	low	yes	no	manual
		pFabric [4]	no	high	no	no	manual
	congestion control	D2TCP [47]	no	high	yes	no	n/a
		DCTCP [2]	no	high	yes	no	n/a
	other	[13, 48, 26, 52, 45, 54]	no	high	yes	no	n/a

Table 2: Comparison of the features and deployability of prior approaches.

While there are no implementations of SNC in computer systems, there are a few works that use SNC in the modeling of critical infrastructures such as avionic networks [42] and the power grid [50, 21], which supports the robustness of SNC theory.

Reducing tail latencies

There are many systems that demonstrate how to significantly reduce tail latency. Datacenter schedulers, like pHost [20], Fastpass [39], and pFabric [4], improve the tail latency by bringing near-optimal schedulers (like earliest-deadline first) to the datacenter. These approaches can also shift queueing from within the network to the end-hosts, which greatly reduces tail packet latency and the latency of short messages. These approaches, however, are not designed to ensure tail latency SLO compliance.

Latency-aware congestion control algorithms, like D2TCP [47] and DCTCP [2], aggressively scale down sending rates and prioritize flows with deadlines. Unfortunately, these approaches can only react to congestion, which can lead to late decisions in face of bursty traffic [4, 27]. HULL [3] is an extension of DCTCP that keeps tail latencies low by controlling the network utilization. Unfortunately, recent experiments show that this does not prevent latency SLO violations [23].

Other techniques for reducing tail latency include issuing redundant requests [13, 48, 26], latency-adaptive machine selection [52, 45], and latency-adaptive load balancing [54]. All these proposals are orthogonal to our work: these techniques can significantly reduce the tail latency, but cannot give guarantees on the tail latency.

Measurement-based approaches

Several recent works measure the latency and adapt the system to try to meet tail latency SLOs [49, 32]. Unfortunately, these approaches aren't suited for admission control where measurements cannot be made dynamically, and prior work has shown that reactive approaches struggle with bursty tenants and often do not meet their SLOs [55].

The recent Cerebro [28] work uses measurements to characterize the latency of requests composed of multiple sub-requests. Unlike SNC-Meister, Cerebro is not designed to account for the

interaction between multiple tenants, which is a primary benefit of SNC.

8 Conclusion and discussion

SNC-Meister is a new system for meeting tail request latency SLOs while achieving much higher multi-tenancy than the state-of-the-art. In experiments with production traces, on a physical implementation testbed, we show that SNC-Meister can admit two to three times as many tenants as the state-of-the-art while meeting tail latency SLOs. SNC-Meister draws its power from being the first computer system to apply a new probabilistic theory called Stochastic Network Calculus (SNC) to calculate tail latencies, while prior systems used the worst-case Deterministic Network Calculus (DNC) theory.

As SNC is a very new theory, there are many challenges in bringing it to practice, and there is much room for further research. One challenge we identify is the important role of the order in which SNC's operators are applied – a fundamental problem that was not previously considered in SNC literature. Our novel algorithm for analyzing networks with SNC makes a significant step forward in making SNC a practical tool. We also add support in SNC-Meister for dependencies between subsets of tenants, which solves a practical issue that is generally ignored in SNC theory. Nevertheless, it is still an open question on how to better apply SNC techniques to get tighter bounds.

While this work focuses on the admission control problem, the ideas behind SNC-Meister and SNC are applicable to many applications beyond admission control. One such example is the datacenter provisioning problem. By being able to analyze tenant behavior and compute tail latency, SNC-Meister could be extended to deciding when (and how many) more resources are required for meeting tail latency SLOs. Similarly, these techniques could apply to tenant placement problems: SNC could be used to identify bottlenecks and make placement decisions in a tail latency aware fashion. We thus believe that the SNC theory can develop into a practical tool for working with tail latency. This first implementation of SNC, in SNC-Meister, is a first, but significant, step in this direction.

A SNC-Meister Analysis Algorithm and Correctness Proof

This section gives a detailed explanation of SNC-Meister's analysis technique and the corresponding proof of correctness. In order to state this proof, we first introduce basic SNC definitions and assumptions (Sec. A.1) and the SNC operators (Sec. A.2). We then give a detailed example explaining prior approaches to SNC network analysis and our approach in SNC-Meister (Sec. A.3). Finally, we describe the SNC-Meister analysis algorithm (Sec. A.4) and state the corresponding correctness proofs (Sec. A.5).

Secs. A.2 and A.3 describe material that is already known to the SNC community. Sects. A.4 and A.5 and parts of Sect. A.3 describe material that forms new contributions. These are new techniques that SNC-Meister develops to extend SNC both with respect to making it practical for real systems and also with respect to greatly improving the accuracy of latency bounds derived in SNC.

A.1 Basic SNC assumptions and definitions

Our SNC model is based on the "$\rho(\theta)$,$\sigma(\theta)$" notation developed by Chang [8] and the moment generating function framework by Fidler [16]. Note that we use the common discrete-time form, where the time step size is small enough to capture continuous-time effects. An excellent in-depth introduction of the discrete-time SNC building blocks and SNC operators can be found in a recent survey [18].

SNC is based on four definitions (the arrival process, the MGF-arrival bound, the service process, and the MGF-service bound), which are modified via the SNC operators (Sec. A.2).

We first formally define the *arrival process*, which captures the total work arriving from a tenant in any time interval.

Definition 1 (Arrival process)**.** *Let a_i $i \geq 1$ denote the work increments of a tenant. The cumulative work received between time m and n,*

$$A(m,n) := \sum_{i=1}^{n} a_i - \sum_{i=1}^{m} a_i$$

is called arrival process of this tenant.

Using this definition, we can formulate an upper bound on the distribution of the arrival process, using its MGF. Recall that the moment generating function (MGF) of a random variable X is defined as $\mathbb{E}[e^{\theta X}]$.

Definition 2 (MGF-arrival bound)**.** *Let $A(m,n)$ denote the arrival process of a tenant. Then, this tenant has the MGF-arrival bound $\rho_A(\theta), \sigma_A(\theta)$, if the moment-generating function of A exists and is bounded*

$$\mathbb{E}[e^{\theta A(m,n)}] \leq e^{\theta((n-m)\cdot\rho_A(\theta)+\sigma_A(\theta))} \quad \textit{for all } m \leq n \in \mathbb{N} \textit{ and } \theta > 0\,.$$

Note that the MGF-arrival bound captures both the arrival instants (time stamps) and each arrival's work requirement (the request size), and thus upper bounds the total work (in bytes) arriving in an interval. A typical MGF-arrival bound can be found in Sec. 3.2 in the form of the Markov-modulated process.

Having bounded a tenant's arrivals, we next formalize the service model. We first formally define the *service process* assumption, which formalizes the relation between queue departures and the service process: if there are waiting arrivals, then the minimal number of finished requests (departures) is given by the service process. Note that the service process assumption is also known as the *dynamic server* assumption in the SNC literature [18]. We use $D(m, n)$ to describe the departures (the output in bytes) from a queue between time m and n, see [17] for more details about this definition.

Definition 3 (Service process (dynamic server))**.** *Let $S(m, n)$ describe the total work processed by a queue between time m and n, and let $D(m, n)$ denote the queue's departures. S is called a service process with departures $D(m, n)$, if S is positive and increasing in n and if for any tenant with arrival process $A(m, n)$ it holds that*

$$D(0, n) \geq \min_{0 \leq k \leq n} \{A(0, k) + S(k, n)\}$$

Note that service process assumption is fundamental for the correctness of SNC calculations and checking this assumption is a key step in the correctness proofs in Sec. A.5. Similar to the arrival process definition, $S(m, n)$ is measured in bytes.

Using the service process definition, we can formulate an upper bound on the distribution of the service process, using its MGF.

Definition 4 (MGF-service bound)**.** *Let S be a service process. Then, S has the MGF-service bound $\rho_S(\theta), \sigma_S(\theta)$, if the moment-generating function of S exists and is bounded*

$$\mathbb{E}[e^{-\theta S(m,n)}] \leq e^{\theta((n-m)\cdot\rho_S(\theta)+\sigma_S(\theta))} \quad \textit{for all } m \leq n \in \mathbb{N} \textit{ and } \theta > 0\ .$$

Note that the negative θ in the bound on the MGF actually makes this a lower bound on the service (the rate $\rho_S(\theta)$ is also negative).

A.2 Formal definition of the SNC operators

The concepts behind the SNC operators are described in Sec. 3. We recall that there are five SNC operators: the latency operator ($Latency$), the left-over operator ($\ominus$), the output operator ($\oslash$), the convolution operator ($\otimes$), and the aggregation operator ($\oplus$). While Tbl. 1 gives an overview over the most commonly used form of the operators, this section states the precise mathematical definition and assumptions and gives pointers to respective correctness proofs in the literature.

We start with the SNC latency bound. Recall that the SNC latency bound relies on a bound on a tenant's arrivals and the corresponding service process.

Theorem 1 (Latency Operator [16, 5])**.** *Let A be an arrival process and let S be a service process. Assume that A has MGF-arrival bound $\rho_A(\theta), \sigma_A(\theta)$, S has MGF-service bound $\rho_S(\theta), \sigma_S(\theta)$, and that $(-\rho_S(\theta)) > \rho_A(\theta)$.*

dependent case: *An upper bound on the tail latency L as a function of the percentile p is given by*

$$L(p) \leq \min_{\theta>0} \left\{ \frac{1}{\theta \rho_S(y\ \theta)} \log\left((1-p) \cdot \left(1 - e^{\theta \cdot (\rho_A(x\ \theta) + \rho_S(y\ \theta))}\right)\right) - \frac{1}{\rho_S(y\ \theta)} (\sigma_A(x\ \theta) + \sigma_S(y\ \theta)) \right\},$$

for any $x, y \in (1, \infty]$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

independent case: *If, A and S are stochastically independent, then the tail latency bound simplifies to*

$$L(p) \leq \min_{\theta>0} \left\{ \frac{1}{\theta \rho_S(\theta)} \log\left((1-p) \cdot \left(1 - e^{\theta \cdot (\rho_A(\theta) + \rho_S(\theta))}\right)\right) - \frac{1}{\rho_S(\theta)} (\sigma_A(\theta) + \sigma_S(\theta)) \right\},$$

for any $\theta > 0$.

Note that the assumption $(-\rho_S(\theta)) > \rho_A(\theta)$ is essentially a stability condition, as the time-dependent ρ_A component of the arrival process has to be lower than the time-dependent ρ_S component of the service process. Furthermore, note that the tail latency bound is valid for any fixed $\theta > 0$, and thus we can minimize this upper bound on the latency prediction by searching across θ's parameter range. This is done automatically by SNC-Meister as explained in Sec. 4.4.

Finally, note that the dependent case has additional parameters (x and y), besides θ. The latency bound is valid for any x and y (fulfilling $x, y \in (1, \infty]$ with $\frac{1}{x} + \frac{1}{y} = 1$), which requires an additional search for the minimal parameters. Additionally, we remark that the dependent case leads to significantly higher latency bounds because there is less multiplexing benefit. Mathematically, the lack of independence means that the dependent-case form relies on the Hoelder bound, which is "costly" and leads to a much higher latency prediction [16]. Sec. A.3 explains this further.

The next operator characterizes the left-over service for a tenant that shares a queue with a higher-or-equal priority tenant.

Theorem 2 (Left-Over Operator [16, 5])**.** *Assume that two tenants share a queue with service process S, for which the first tenant has higher or equal priority than the second. The tenant's arrival processes are A_1 and A_2, respectively. Then, the service offered by the queue to the second tenant is the service process $S \ominus A_1$.*

Assume that A_1 has MGF-arrival bound $\rho_{A_1}(\theta), \sigma_{A_1}(\theta)$, and that S has MGF-service bound $\rho_S(\theta), \sigma_S(\theta)$.

dependent case: *The service process $S \ominus A_1$ has MGF-service bound $\rho_{S \ominus A_1}, \sigma_{S \ominus A_1}$ with*

$$\begin{aligned} \rho_{S \ominus A_1} &= \rho_{A_1}(x\ \theta) + \rho_S(y\ \theta) \\ \sigma_{S \ominus A_1} &= \sigma_{A_1}(x\ \theta) + \sigma_S(y\ \theta)\ , \end{aligned}$$

for any $x, y \in (1, \infty]$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

independent case: *If, A_1 and S are stochastically independent, then the MGF-service bound simplifies to*

$$\begin{aligned} \rho_{S \ominus A_1} &= \rho_{A_1}(\theta) + \rho_S(\theta) \\ \sigma_{S \ominus A_1} &= \sigma_{A_1}(\theta) + \sigma_S(\theta)\ , \end{aligned}$$

for any $\theta > 0$.

Note that if the queue is shared between many tenants, this theorem can be repeatedly applied because the resulting $S \ominus A_1$ again fulfills the assumption of the theorem.

We also remark, that Theorem 2 is very conservative for the case when the two tenants have the same priority. For specific cases of scheduling policies, like FIFO scheduling, there are more accurate analysis techniques in the literature [11]. However, since switching fabrics do not strictly follow FIFO in practice, our analysis does not rely on assuming a specific scheduling policy (such as FIFO).

The next operator is the output operator, which is used to calculate a bound on the departures from a queue, which can then form the input (arrival process) to another queue in a network.

Theorem 3 (Output Operator [16, 5])**.** *A tenant with arrival process A traverses a queue with service process S. Assume that A has MGF-arrival bound $\rho_A(\theta), \sigma_A(\theta)$ and that S has MGF-service bound $\rho_S(\theta), \sigma_S(\theta)$.*

dependent case: *The departure process $A \oslash S$ ("output") has the MGF-arrival bound $\rho_{A\oslash S}(\theta), \sigma_{A\oslash S}(\theta)$ given by*

$$\begin{aligned}\rho_{A\oslash S}(\theta) &= \rho_A(x\ \theta)\\ \sigma_{A\oslash S}(\theta) &= \sigma_A(x\ \theta) + \sigma_S(y\ \theta) - \frac{1}{\theta}\log\left(1 - e^{\theta(\rho_A(x\ \theta)+\rho_S(y\ \theta))}\right)\ .\end{aligned}$$

for any $x, y \in (1, \infty]$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

independent case: *If A and S are stochastically independent, then the MGF-service bound simplifies to*

$$\begin{aligned}\rho_{A\oslash S}(\theta) &= \rho_A(\theta)\\ \sigma_{A\oslash S}(\theta) &= \sigma_A(\theta) + \sigma_S(\theta) - \frac{1}{\theta}\log\left(1 - e^{\theta(\rho_A(\theta)+\rho_S(\theta))}\right)\ ,\end{aligned}$$

for any $\theta > 0$.

The next operator is the convolution operator, which is used to "merge" two (or more) queues in sequence into a single mathematical representation.

Theorem 4 (Convolution Operator [16])**.** *Let S and T be two service processes, which have MGF-service bounds $\rho_S(\theta), \sigma_S(\theta)$ and $\rho_T(\theta), \sigma_T(\theta)$, respectively. If $\rho_S(\theta) \neq \rho_T(\theta)$, this network can be replaced by the service process $S \otimes T$.*

dependent case: *The MGF-service bound of $S \otimes T$ is given by $\sigma_{S\otimes T}, \rho_{S\otimes T}$ with*

$$\begin{aligned}\rho_{S\otimes T}(\theta) &= \max\left\{\rho_S(x\ \theta), \rho_T(y\ \theta)\right\}\\ \sigma_{S\otimes T}(\theta) &= \sigma_S(x\ \theta) + \sigma_T(y\ \theta) - \frac{1}{\theta}\log\left(1 - e^{-\theta|\rho_S(x\ \theta)-\rho_T(y\ \theta)|}\right)\ ,\end{aligned}$$

for any $x, y \in (1, \infty]$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

independent case: *If, S and T are stochastically independent, then the MGF-service bound simplifies to*

$$\rho_{S\otimes T}(\theta) = \max\{\rho_S(\theta), \rho_T(\theta)\}$$
$$\sigma_{S\otimes T}(\theta) = \sigma_S(\theta) + \sigma_T(\theta) - \frac{1}{\theta}\log\left(1 - e^{-\theta|\rho_S(\theta)-\rho_T(\theta)|}\right) ,$$

for any $\theta > 0$.

The idea behind the convolution theorem is that it can be repeatedly applied until each tenant's arrival process in a network traverses a single (convolution-type) service process [9, 16].

Note that the case $\rho_S(\theta) = \rho_T(\theta)$ is not covered by this theorem. The simplest way around this problem is to assume that one of the server is slightly slower than the other (e.g., scaling $\rho_S(\theta)$ by 0.99), which makes little difference numerically and allows us to always use this theorem.

The final SNC operator is used to merge two arrival processes into one, which is called aggregation.

Theorem 5 (Aggregation Operator)**.** *Assume two tenants with arrival processes A_1 and A_2, respectively. Assume that A_1 has MGF-arrival bound $\rho_{A_1}(\theta), \sigma_{A_1}(\theta)$ and that A_2 has MGF-arrival bound $\rho_{A_2}(\theta), \sigma_{A_2}(\theta)$*

dependent case: *The aggregated arrival process $A_1 \oplus A_2$ has MGF-arrival bound $\sigma_{A_1\oplus A_2}, \rho_{A_1\oplus A_2}$ with*

$$\rho_{A_1\oplus A_2}(\theta) = \rho_{A_1}(x\,\theta) + \rho_{A_2}(y\,\theta)\sigma_{A_1\oplus A_2}(\theta) \qquad = \sigma_{A_1}(x\,\theta) + \sigma_{A_2}(y\,\theta)$$

for any $x, y \in (1, \infty]$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

independent case: *If A_1 and A_2 are stochastically independent, then the MGF-service bound simplifies to*

$$\rho_{A_1\oplus A_2}(\theta) = \rho_{A_1}(\theta) + \rho_{A_2}(\theta)\sigma_{A_1\oplus A_2}(\theta) \qquad = \sigma_{A_1}(\theta) + \sigma_{A_2}(\theta)$$

for any $\theta > 0$.

This concludes the formal description of the five SNC operators.

A.3 Example: SNC convolution, hop-by-hop, and SNC-Meister analysis

Having formally introduced the SNC operators, we are now ready to give an example for the hop-by-hop analysis technique, convolution analysis technique, and SNC-Meister's analysis technique. All three techniques are based on the five SNC operators but differ in the order in which the operators are applied.

We consider a simple network, which showcases the challenge of working with stochastic dependencies. After discussing the simple network, we consider how the dependency problem compounds as more tenants are added to the network.

Recall the example network analyzed in Sec. 4.1, repeated here as Fig. 15. There are four tenants, with arrival processes A_{T_1}, A_{T_2}, A_{T_3}, and A_{T_4}. The four tenants traverse three queues, with

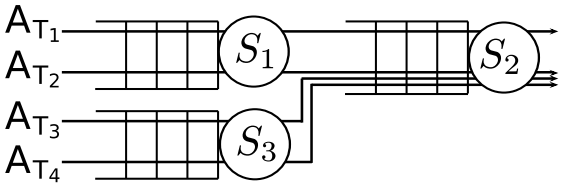


Figure 15: Example network with four tenants T_1 to T_4 flowing through three queues S_1 to S_3.

service processes S_1, S_2, and S_3. For the sake of simplicity, we assume that T_1 has a strictly lower priority than T_2 to T_4 on all queues and that all tenants are stochastically independent to start with. Sec. A.4 shows how to work with user-specified tenant dependencies. We furthermore require that all tenants have MGF-arrival bounds and all service processes have MGF-service bounds.

Recall that, besides the latency operator, there are five SNC operators, which modify arrival processes and service process: the left-over operator ($\ominus$), the output operator ($\oslash$), the convolution operator ($\otimes$), and the aggregation operator ($\oplus$).

All network analysis approach have to first consider the departures from tenants T_3 and T_4 at S_3. A straightforward application of the left-over and output operators at S_3, would calculate their departures from S_3 as follows

$$A'_{T_3} = A_{T_3} \oslash (S_3 \ominus A_{T_4}), \text{ and}$$
$$A'_{T_4} = A_{T_4} \oslash (S_3 \ominus A_{T_3}) \text{ ,respectively.}$$

Then, when analyzing S_2 and subtracting A'_{T_3} and A'_{T_4} from S_2 (to calculate the service available to T), we would run into an artificial stochastic dependency (because A'_{T_3} and A'_{T_4} are not independent). In this example, it is easy to avoid this artificial dependency by aggregating A_{T_3} and A_{T_4} right from the start. This aggregation trick is a key part of SNC-Meister's analysis technique and will be discussed in more detail later.

We next state the explicit operator sequences to analyze the whole network based on the hop-by-bop approach, convolution approach, and SNC-Meister's approach.

The first approach is called hop-by-hop, because it separately applies the tail latency bound from Theorem 1 to each queue. Recent work [5] has shown that this technique can be used to analyze a broad set of queueing networks (feed-forward networks).

Analysis approach 1 (SNC hop-by-hop)**.** *We first derive the service S'_1 offered to T_1 at the first queue*

$$S'_1 = S_1 \ominus A_{T_2}$$

where we subtract the arrival processes of the tenant T_2. We can then calculate the tail latency T_1 at the first queue with

$$Latency(A_{T_1}, S'_1, 0.995) \; . \tag{1}$$

In order to analyze the latency at the second queue, we first derive the arrival process of T_1 at the second queue (which is the departure process from the first queue)

$$A'_{T_1} = A_{T_1} \oslash S'_1 \ ,$$

using the output operator. Similarly, we derive the departure process for T_2 as $A'_{T_2} = A_{T_2} \oslash S_1$. For tenants T_3 and T_4 we first aggregate them into a single arrival process and then calculate their departure process from S_3 as $A'_{T_{3/4}} = (A_{T_3} \oplus A_{T_4}) \oslash S_3$. The local service S'_2 offered to T_1 at the second queue is then derived as

$$S'_2 = S_2 \ominus A'_{T_2} \ominus A'_{T_{3/4}}$$

and we calculate the tail latency of T_1 at the second queue with

$$Latency(A'_{T_1}, S'_2, 0.995) \ . \tag{2}$$

Note that Eq. (2) *includes a stochastic dependency (S_1 occurs in both A'_{T_1} and S'_2), which means that the operators cannot use the simplified independent-case equation but need to use the more complex dependent-case equation.*

Observe that the stochastic dependency in the hop-by-hop analysis approach is inherent to the analysis and not due to actual dependencies between tenants (we assumed them to be initially stochastically independent). We therefore call such a stochastic dependency an *artificial dependency* as opposed to an actual (user-specified) dependency between tenants.

The second analysis approach is called SNC convolution and emerges when applying the line-network analysis technique [9, 16, 22, 33, 6, 10, 5, 18] to our network. The goal of the SNC convolution approach is to merge all queues into a single service process and to then apply Theorem 1 once to obtain the tail latency.

Analysis approach 2 (SNC convolution)**.** *We first apply the left-over operator to every queue to obtain the "local service process" offered to T_1. We use S'_1 to denote the local service process at the first queue, and we use S'_2 to denote the local service process at the second queue, and this follows exactly the same steps as in Approach 1.*

We next use the convolution operator to merge the two local service processes together into a global service process S

$$S = \left(S'_1 \otimes S'_2\right) \tag{3}$$

We can then calculate the tail latency of T_1 at both queues using

$$Latency(A_{T_1}, S, 0.99) \ .$$

Note that Eq (3) *includes an artificial stochastic dependency. The benefit of the convolution approach is that it only requires a single latency calculation.*

In contrast to both SNC hop-by-bop and SNC convolution, SNC-Meister uses a novel operator sequence, which minimizes the number of stochastic dependencies in the network analysis. For the particular example given here, it is easy to show that our analysis does not have any stochastic dependencies and always uses the more accurate stochastic independent-case of the operator equations.

Analysis approach 3 (SNC-Meister)**.** *The idea is similar to Approach 2, but changes the position of A_{T_2} in the operator sequence. Specifically, we exclude T_2 in the derivation of the local service processes for each queue, because it shares the whole path with T_1. We apply the left-over operator to T_2 and T_1 only after having merged the two local service processes into a global service process. This leads to the following operator sequence*

$$S = \Big(S_1 \otimes \big(S_2 \ominus ((A_{T_3} \oplus A_{T_4}) \oslash S_3) \big) \Big) \ominus A_{T_2} \ , \tag{4}$$

where in the inner-most parenthesis we aggregate T_3 and T_4 before calculating their departures from S_3 and then subtracting them from S_2. In the outer-most parenthesis, T_2 is subtracted after having merged S_1 and the left-overs from S_2 using the convolution operator. We calculate the tail latency of T_1 at both queues using

$$Latency(A_{T_1}, S, 0.99) \ .$$

Note that Eq. (4) *does not include any stochastic dependencies.*

Note that SNC-Meister's equation is simpler than the other approaches, but requires changing the order of several operators, which are not necessarily exchangeable. We therefore need to prove the correctness of this change in the operator sequence, which is done in Sec. A.5.

We remark that the aggregation step (described before the three approaches) play an important role in preventing stochastic dependencies. Unfortunately, as more and more tenants are added to a network aggregating departures becomes more complex.

Fig. 16 expands the network from Fig. 15 with two additional tenants traversing S_3. Specifically, we now consider four tenants at S_3, T_3 to T_6, of which only two, T_4 and T_6, traverse the queue S_2. The four tenants are ordered by strictly decreasing scheduling priority. As in the previous example, we are interested in analyzing S_2, which requires the departures from T_4 and T_6.

If we calculate the departures of T_4 and T_6 separately (i.e., without aggregation), we would introduce artificial stochastic dependencies. Therefore, it would be helpful to aggregate the departures. Unfortunately, T_4 and T_6 have different scheduling priorities, which prevents aggregating them into a single arrival process. SNC-Meister solves this problem by relaxing the priority of the higher-priority tenant, T_4, and then calculating the aggregated arrival process. While this seems at first counter-intuitive – as assuming T_4 has a lower priority makes the analysis conservative – this step is necessary to resolve artificial stochastic dependencies which would be introduced without aggregation. Changing these priorities (for the sake of analysis) and efficiently aggregating tenants are key parts of SNC-Meister's analysis algorithm.

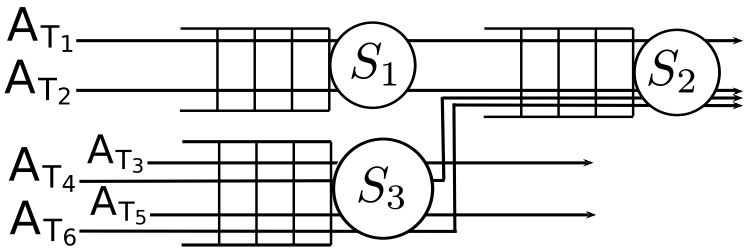


Figure 16: Reordering of tenant priorities can be necessary to allow for aggregation of the departures of tenants. In this figure, the tenant priorities are ordered $A_{T_3} > A_{T_4} > A_{T_5} > A_{T_6}$, which makes aggregating T_4 and T_6 impossible. In order to aggregate T_4 and T_6, they need to be in the same priority class, which means decreasing the priority of T_4 (for the sake of the analysis).

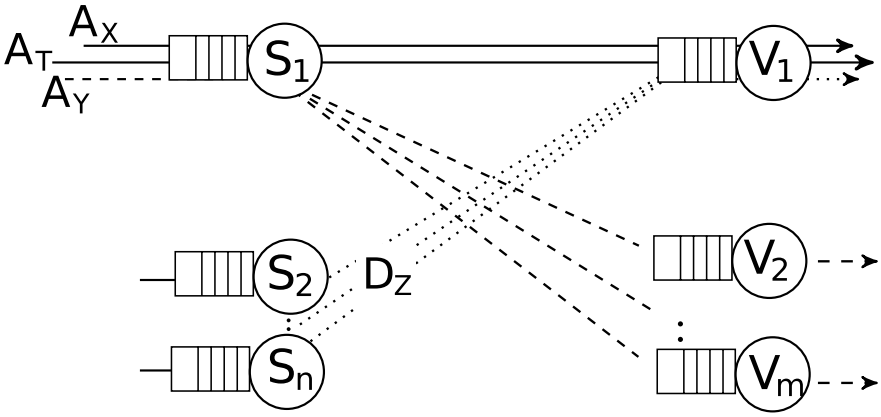


Figure 17: SNC-Meister calculates upper bounds for a bipartite graph of service processes $(S_i), i = 1..n$ (left-hand side) and $(V_j), j = 1..m$ (right-hand side). For example, in our experiments in Sec. 6 $n = 180$ tenants and $m = 6$ server nodes.

A.4 SNC-Meister's analysis algorithm

This section introduces the two parts of SNC-Meister's analysis algorithm: the arrival process aggregation algorithm and the actual network analysis based on the aggregation algorithm.

The first part, the arrival process aggregation algorithm, has the goal of aggregating many arrival processes, which might have inter-dependencies originating from user-specified dependencies. Specifically, the input to this algorithm is a list of arrival processes, and a graph of their dependencies. The dependency graph has an edge between arrival process A_i and A_j, if they are inter-dependent. In SNC-Meister we assume that user-specified dependencies are transitive (i.e., if i and j are dependent, and j and k are dependent, then also tenants i and k must be dependent). This means that the dependency graph consists of several cliques, which each represent one set of inter-dependent arrival processes.

The goal of the aggregation algorithm is to apply the minimal number of dependent-case SNC

operators to merge all arrival processes into a single arrival process. We will use this aggregation algorithm multiple times in the network analysis. The algorithm proceeds in four steps:

1. create a vector of groups G;
2. for each arrival process A, add A to the lowest numbered group in G that does not have a tenant with a dependency on A;
3. for each group g in G, aggregate the arrival processes in g, which are all independent by construction, and store the aggregate in G';
4. aggregate all the aggregates in G', which all are dependent by construction.

The output is a single arrival process.

The second part, SNC-Meister's network analysis algorithm, has the goal of deriving an upper bound on the tail latency for a given tenant. Specifically, this algorithm considers the network sketched in Fig. 17. The network of queues is a bipartite graph connecting two sets of service processes S_i ($i \in 1..n$) and V_j ($j \in 1..m$). Without loss of generality, we consider the tenant labeled T, which traverses S_1 and V_1. SNC-Meister's network analysis algorithm then proceeds in five steps:

1.
   - Find the set X of all tenants (excluding T) that traverse S_1 and V_1 and have a higher or equal priority than T.
   - Use the aggregation algorithm to merge the arrival processes of all tenants in X and denote the aggregated arrival process by A_X.
2.
   - Find the set Y of all tenants that traverse S_1 and V_j ($j > 1$) and have a higher or equal priority than T.
   - Use the aggregation algorithm to merge the arrival processes of all tenants in Y and denote the aggregated arrival process by A_Y.
3.
   - For each $i > 1$:
     - find all tenants Z_i, which traverse S_i and V_1 and have a higher or equal priority than T;
     - use the aggregation algorithm to merge all of Z_i's arrival processes into A_{Z_i};
     - find all tenants R_i, which traverse S_i, are not in Z_i, and have a higher or equal priority than the lowest priority tenant in Z_i;
     - use the aggregation algorithm to merge all of R_i's arrival processes into A_{R_i}; and
     - calculate the departure process for S_i: $D_{Z_i} = A_{Z_i} \oslash (S_i \ominus A_{R_i})$.
   - Use the aggregation algorithm to merge all departure processes into $D_Z = D_{Z_1} \oplus D_{Z_2} \oplus D_{Z_3} \oplus \ldots$.
4. Calculate the network service process for tenant T using the SNC operators and the following equation

$$S = \Big(\big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z) \big) \ominus A_X \Big) . \tag{5}$$

5. Use T's arrival process, S, and the SNC latency bound to derive the latency for T, for the percentile p via $Latency(A_T, S, p)$.

A.5 Correctness of SNC-Meister's analysis algorithm

This section describes the correctness proofs of SNC-Meister's analysis algorithm described in the previous section. Specifically, we prove three statements: a) that the aggregation algorithm does in fact lead to the minimal number of stochastic dependencies, b) that step 3 in the network analysis correctly calculates an output bound, and c) the correctness of the service process equation Eq. (5).

To prove statement a), we recall the dependency graph which is the input to the aggregation algorithm. As described above, this graph consists of several cliques, where each clique represents a set of inter-dependent arrival processes.

Theorem 6 (Optimality of aggregation algorithm)**.** *Let k be the maximum size of a clique in the dependency graph. The minimal number of applications of dependent-case SNC operators for any algorithm is $k-1$.*

Our aggregation algorithm requires $k-1$ applications of dependent-case SNC operators.

Proof. Note that the first statement is trivial because clearly each arrival process in the largest clique has to be aggregated with a dependency operation. Since all tenants in the largest clique are inter-dependent, we need at least $k-1$ aggregations with the dependent-case SNC operator.

We next prove that the aggregation algorithm needs at most $k-1$ dependent-case operations. Assume for the sake of contradiction that the aggregation algorithm requires k dependency operations. Step 4 in the aggregation algorithm requires $|G'|-1$ dependency operations. Thus, $k = |G'|-1 = |G|-1$, which implies $|G| = k+1$. This means that in step 2, there was some arrival process A^*, such that A^* was added to the $k+1$ group. This can only happen if A^* is dependent with some arrival process in all groups $1,\ldots,k$. Now by the transitive property of stochastic dependencies, we have a clique of size $k+1$ with A^* and the other arrival processes that it is dependent on in each group. This is a contradiction to k being the maximum size of a clique. □

To prove statement b), we show that decreasing the priority of a tenant (for the sake of analysis) leads to an upper bound on the tenant's departures.

Lemma 1 (Aggregation with changed priorities)**.** *Assume that a set of tenants traverses a queue with service process S. Let Z denote a subset for which we are interested in a bound on the aggregated departures. Let R denote all tenants with an equal or higher priority than the lowest-priority tenant in Z. Assume that the aggregated arrival processes from Z and R have MGF-bounds A_Z and A_R, respectively.*

Then, the departure process of all Z tenants, D_Z, is upper bounded

$$D_Z \leq A_Z \oslash (S \ominus A_R) \,.$$

Proof. It is sufficient to show that for every tenant, calculating the departure process by decreasing the tenant's priority is an upper bound on the departure process with the original priority. To this end, let A denote any fixed tenant and let S denote the local service process of A at the queue. Let D denote A's departures. According to the departure theorem [31], it holds that

$$D(m,n) \leq \max_{0\leq k\leq m} \{A(k,n) - S(k,m)\} \,. \tag{6}$$

Decreasing the priority of A results in a service process $S'(m,n) \leq S(m,n)$ (for all $m \leq n \in \mathbb{N}$). Therefore, $\max_{0\leq k\leq m}\{A(k,n) - S'(k,m)\}$ gives an upper bound on the right-hand side of Eq.(6). This extends to the stochastic case as $\sigma_{A\oslash S}(\theta)$ in Theorem 3 is monotonically increasing in $\sigma_S(\theta)$. □

To prove statement c) (Eq. (5)), we need to verify the assumptions of the SNC latency bound, which is that S is a service process (Definition 3).

The following theorem formally describes this scenario and SNC-Meister's operator sequence together with a full proof of correctness.

Theorem 7 (Independent Network Analysis)**.** *Assume the scenario shown in Fig. 17:*

- *a bipartite graph connecting two sets of service processes S_i ($i \in 1..n$) and V_j ($j \in 1..m$)*
- *a set of tenants X traverses S_1 and V_1, and the aggregate arrivals are bounded by A_X;*
- *a set of tenants Y traverses S_1 and V_j ($j > 1$), and the aggregate arrivals are bounded by A_Y;*
- *a set of tenants Z originates as departures from S_i ($i > 1$) and traverses V_1, and the aggregate departures from these tenants are bounded by D_Z .*

We are interested in the tenant T, which has a lower or equal priority to X, Y, Z and traverses S_1 and V_1. The order of priorities between X, Y, Z can be arbitrary.

Then, the tail latency can be calculated using Theorem 1 using A_T and the service process S

$$S(m,n) = \Big(\big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z)\big) \ominus A_X\Big)\big(m,n\big) \ ,$$

whose MGF-service bound is calculated using the standard SNC operators.

Note that because the correctness of individual operators has already been proven, it remains to be shown that changing the operator sequence satisfies the service process requirement from Definition 3.

Proof. We explicitly show that the theorem's operator sequence satisfies the service process condition.

Let D_X, D_Y, D_T, D_Z denote the departures from a server in the first column, and let D^*_X, D^*_T, D^*_Z denote the departures from a server in the second column.

Let $n \geq 0$ be arbitrary, and choose $m \leq n$ maximal such that $D^*_X(0,m) = D_X(0,m)$, $D^*_T(0,m) = D_T(0,m)$, $D^*_Z(0,m) = D_Z(0,m)$ (i.e., m is the start of the current V_1-server busy period, if it exists). Then, by definition of V_1, it holds that

$$D^*_T(0,n) - D^*_T(0,m) = T(m,n) - (D^*_X(0,n) - D^*_X(0,m)) - (D^*_Z(0,n) - D^*_Z(0,m)) \quad . \quad (7)$$

By definition of m, and by Theorem 2 (applied to the left-over service of S_1), it also holds that

$$D^*_T(0,m) = D_T(0,m) \geq A_T \otimes (S_1 \ominus A_X \ominus A_Y)(0,m) \ .$$

Next, we use two classical bounding steps (e.g.,[31, p.21]) to estimate D^*_X and D^*_Z

$$\begin{aligned} D^*_X(0,n) - D^*_X(0,m) &\leq D_X(0,n) - D_X(0,m) \leq A_X(0,n) - A_X(0,m) \\ D^*_Z(0,n) - D^*_Z(0,m) &\leq D_Z(0,n) - D_Z(0,m) \ . \end{aligned}$$

We conclude by plugging into Eq.(7) the previous three inequalities, and bringing $D_T(0,m)$ to the right side

$$\begin{aligned} D_T^*(0,n) \geq\ & A_T \otimes (S_1 \ominus A_X \ominus A_Y)(0,m) \\ & + T(m,n) - (A_X(0,n) - A_X(0,m)) \\ & - (D_Z(0,n) - D_Z(0,m)) \quad , \end{aligned}$$

which can be reformulated using the definition of m

$$\begin{aligned} D_T^*(0,n) \geq & \min_{0\leq m\leq n} \Big\{ A_T \otimes (S_1 \ominus A_X \ominus A_Y)(0,m) \\ & + T(m,n) - (A_X(0,n) - A_X(0,m)) \\ & - (D_Z(0,n) - D_Z(0,m)) \Big\} \\ = & \min_{0\leq m\leq n} \Big\{ \min_{0\leq k\leq m} \big\{ A_T(0,k) + S_1(k,m) \\ & \quad - (A_X(0,m) - A_x(0,k)) - (A_Y(0,m) - A_Y(0,k)) \big\} \\ & + T(m,n) - (A_X(0,n) - A_X(0,m)) \\ & - (D_Z(0,n) - D_Z(0,m)) \Big\} \\ = & \min_{0\leq k\leq m\leq n} \Big\{ A_T(0,k) + S_1(k,m) \\ & \quad - (A_X(0,m) - A_x(0,k)) - (A_Y(0,m) - A_Y(0,k)) \\ & + T(m,n) - (A_X(0,n) - A_X(0,m)) \\ & - (D_Z(0,n) - D_Z(0,m)) \Big\} \\ = & \min_{0\leq k\leq n} \Big\{ A_T(0,k) \\ & + \min_{k\leq m\leq n} \big\{ S_1(k,m) - (A_Y(0,m) - A_Y(0,k)) \\ & \quad + T(m,n) - (D_Z(0,n) - D_Z(0,m)) \big\} \\ & - (A_X(0,n) - A_X(0,k)) \Big\} \\ = & A_T \otimes \Big(\big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z) \big) \ominus A_X \Big) \quad . \end{aligned}$$

This concludes the proof because S is a service process. □

Observe that $S(m,n)$ preserves all stochastic independencies of T, X, Y, and Z. SNC-Meister's network analysis is thus optimal in the sense in that it does not introduce artificial stochastic dependencies.

References

[1] Alan Agresti. Building and applying logistic regression models. *Categorical Data Analysis, Second Edition*, pages 211–266, 2007.

[2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp DCTCP. In *ACM SIGCOMM*, pages 63–74, 2011.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, pages 19–19, 2012.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*, pages 435–446, 2013.

[5] Michael A Beck and Jens Schmitt. The disco stochastic network calculator version 1.0: when waiting comes to an end. In *Valuetools*, pages 282–285, 2013.

[6] Almut Burchard, Jörg Liebeherr, and Florin Ciucu. On superlinear scaling of network delays. *IEEE/ACM Transactions on Networking (TON)*, 19(4):1043–1056, 2011.

[7] Cheng-Shang Chang. Stability, queue length, and delay of deterministic and stochastic queueing networks. *IEEE Transactions on Automatic Control*, 39(5):913–931, 1994.

[8] Cheng-Shang Chang. *Performance guarantees in communication networks*. Springer Science & Business Media, 2000.

[9] Florin Ciucu, Almut Burchard, and Jörg Liebeherr. A network service curve approach for the stochastic analysis of networks. In *ACM SIGMETRICS*, pages 279–290, 2005.

[10] Florin Ciucu and Jens Schmitt. Perspectives on network calculus: No free lunch, but still good value. In *ACM SIGCOMM*, pages 311–322, 2012.

[11] Rene L Cruz. Sced+: Efficient management of quality of service guarantees. In *IEEE INFOCOM*, volume 2, pages 625–634, 1998.

[12] RL Cruz. Quality of service management in integrated services networks. In *Proceedings of the 1st Semi-Annual Research Review, CWC*, 1996.

[13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SOSP*, pages 205–220, 2007.

[15] Domenico Ferrari and Dinesh C Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE JSAC*, 8(3):368–379, 1990.

[16] Markus Fidler. An end-to-end probabilistic network calculus with moment generating functions. In *IEEE International Workshop on Quality of Service (IWQoS)*, pages 261–270, 2006.

[17] Markus Fidler. Survey of deterministic and stochastic service curve models in the network calculus. *IEEE Communications Surveys & Tutorials*, 12(1):59–86, 2010.

[18] Markus Fidler and Amr Rizk. A guide to the stochastic network calculus. *IEEE Communications Surveys & Tutorials*, 17(1):92–105, 2015.

[19] Victor Firoiu, Jean-Yves Le Boudec, Don Towsley, and Zhi-Li Zhang. Theories and models for internet quality of service. *Proceedings of the IEEE*, 90(9):1565–1591, 2002.

[20] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT*, 2015.

[21] Yashar Ghiassi-Farrokhfal, Srinivasan Keshav, and Catherine Rosenberg. Toward a realistic performance analysis of storage systems in smart grids. *IEEE Transactions on Smart Grid*, 6(1):402–410, 2015.

[22] Yashar Ghiassi-Farrokhfal and Jörg Liebeherr. Output characterization of constant bit rate traffic in fifo networks. *IEEE Communications Letters*, 13(8):618–620, 2009.

[23] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *USENIX NSDI*, 2015.

[24] Daniel P Heyman and David Lucantoni. Modeling multiple ip traffic streams with rate limits. *Networking, IEEE/ACM Transactions on*, 11(6):948–958, 2003.

[25] Sadeka Islam, Srikumar Venugopal, and Anna Liu. Evaluating the impact of fine-scale burstiness on cloud elasticity. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 250–261, New York, NY, USA, 2015. ACM.

[26] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, pages 219–230, 2013.

[27] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *ACM SIGCOMM*, pages 435–448. ACM, 2015.

[28] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Response time service level agreements for cloud-hosted web applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 315–328, New York, NY, USA, 2015. ACM.

[29] Jim Kurose. On computing per-session performance bounds in high-speed multi-hop computer networks. In *ACM SIGMETRICS*, 1992.

[30] Jean-Yves Le Boudec. Application of network calculus to guaranteed service networks. *IEEE Transactions on Information Theory*, 44(3):1087–1096, 1998.

[31] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

[32] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. Pslo: Enforcing the xth percentile latency and throughput slos for consolidated vm storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 28:1–28:14, New York, NY, USA, 2016. ACM.

[33] Jörg Liebeherr, Yashar Ghiassi-Farrokhfal, and Almut Burchard. On the impact of link scheduling on end-to-end delays in large networks. *IEEE JSAC*, 29(5):1009–1020, 2011.

[34] Jörg Liebeherr, Dallas E Wrege, and Domenico Ferrari. Exact admission control for networks with a bounded delay service. *IEEE/ACM Transactions on Networking (TON)*, 4(6):885–901, 1996.

[35] Jeffrey C Mogul and Ramana Rao Kompella. Inferring the network latency requirements of cloud tenants. In *Usenix HotOS XV*, 2015.

[36] Guy Nason. A test for second-order stationarity and approximate confidence intervals for localized autocovariances for locally stationary time series. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 75(5):879–904, 2013.

[37] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[38] Abhay K Parekh and Robert G Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994.

[39] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*, pages 307–318, 2014.

[40] Felix Poloczek and Florin Ciucu. Scheduling analysis with martingales. *Performance Evaluation*, 79:56–72, 2014.

[41] Jing-yu Qiu and Edward W Knightly. Inter-class resource sharing using statistical service envelopes. In *IEEE INFOCOM*, volume 3, pages 1404–1411, 1999.

[42] Jean-Luc Scharbarg, Frédéric Ridouard, and Christian Fraboul. A probabilistic analysis of end-to-end delays on an afdx avionic network. *IEEE Transactions on Industrial Informatics*, 5(1):38–49, 2009.

[43] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[44] David Starobinski and Moshe Sidi. Stochastically bounded burstiness for communication networks. In *IEEE INFOCOM*, volume 1, pages 36–42, 1999.

[45] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, 2015.

[46] Guillaume Urvoy-Keller, Gérard Hébuterne, and Yves Dallery. Traffic engineering in a multipoint-to-point network. *IEEE JSAC*, 20(4):834–849, 2002.

[47] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *ACM SIGCOMM*, pages 115–126, 2012.

[48] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *ACM CoNEXT*, pages 283–294, 2013.

[49] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level slos on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 14:1–14:14, New York, NY, USA, 2012. ACM.

[50] Kai Wang, Florin Ciucu, Chuang Lin, and Steven H Low. A stochastic power network calculus for integrating renewable energy sources into the power grid. *IEEE JSAC*, 30(6):1037–1048, 2012.

[51] Shengquan Wang, Dong Xuan, Riccardo Bettati, and Wei Zhao. Providing absolute differentiated services for real-time applications in static-priority scheduling networks. *IEEE/ACM Transactions on Networking*, 12(2):326–339, 2004.

[52] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *USENIX NSDI*, pages 329–342, 2013.

[53] Opher Yaron and Moshe Sidi. Performance and stability of communication networks via robust exponential bounds. *IEEE/ACM Transactions on Networking*, 1(3):372–385, 1993.

[54] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM*, pages 139–150, 2012.

[55] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *ACM SOCC*, pages 29:1–29:14, New York, NY, USA, 2014. ACM.