Chameleon: Predictable Latency and High Utilization with Queue-Aware and Adaptive Source Routing

Amaury Van Bemten
Chair of Comm. Networks
Tech. Univ. of Munich

Nemanja Đerić
Chair of Comm. Networks
Tech. Univ. of Munich

Amir Varasteh
Chair of Comm. Networks
Tech. Univ. of Munich

Stefan Schmid
Fac. of Computer Science
University of Vienna

Carmen Mas-Machuca
Chair of Comm. Networks
Tech. Univ. of Munich

Andreas Blenk
Chair of Comm. Networks
Tech. Univ. of Munich

Wolfgang Kellerer
Chair of Comm. Networks
Tech. Univ. of Munich

ABSTRACT

This paper presents *Chameleon*, a cloud network providing both predictable latency and high utilization, typically two conflicting goals, especially in multi-tenant datacenters. *Chameleon* exploits routing flexibilities available in modern communication networks to dynamically adapt toward the demand, and uses network calculus principles along individual paths. More specifically, *Chameleon* employs source routing on the "queue-level topology", a network abstraction that accounts for the current states of the network queues and, hence, the different delays of different paths. *Chameleon* is based on a simple greedy algorithm and can be deployed at the edge; it does not require any modifications of network devices. We implement and extensively evaluate *Chameleon* in simulations and a real testbed. Compared to state-of-the-art, we find that *Chameleon* can admit and embed significantly, i.e., up to 15 times more flows, improving network utilization while meeting strict latency guarantees.

ACM Reference Format:
Amaury Van Bemten, Nemanja Đerić, Amir Varasteh, Stefan Schmid, Carmen Mas-Machuca, Andreas Blenk, and Wolfgang Kellerer. 2020. *Chameleon*: Predictable Latency and High Utilization with Queue-Aware and Adaptive Source Routing. In *Proceedings of CoNEXT '20*. ACM, New York, NY, USA, 15 pages. https://doi.org/TBA

1 INTRODUCTION

Datacenter networks have become a critical infrastructure of our digital society. With the popularity of data-centric applications (e.g., related to business, health, entertainment and social networking) and machine learning, the importance of realizing communication networks that meet stringent dependability requirements will likely increase further in the next years. Already today, the usefulness of many distributed cloud applications, such as web search and online retail [12, 30], critically depends on the performance of the underlying network [46], i.e., these applications are sensitive to both packet delay and available network bandwidth [31].

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN TBA…$TBA
https://doi.org/TBA

However, providing predictable network latency and throughput to cloud applications is challenging, especially in multi-tenant datacenters and under dynamic demands that come with uncertainty. In many scenarios, the predictability objective even seems to conflict with efficiency requirements, as the latter forbids conservative resource provisioning.

This paper is motivated by the unprecedented routing flexibilities provided in modern networks, which in principle allow networks to autonomously and dynamically re-evaluate resource allocation decisions, and hence enable novel opportunities navigate the trade-off between predictability, performance, and resource efficiency. In particular, these routing flexibilities enable networks to become *demand-aware*: network configurations can be adapted toward the workload they serve, potentially accounting for current delays along specific paths and exploiting currently underutilized links. The challenge, however, remains how to account for such information, and how to exploit routing flexibilities while maintaining predictability.

At the heart of our approach lies the idea to account for the network queues explicitly in the routing algorithm: rather than performing routing on the level of switches/routers, we propose to perform routing on the *queue-level topology*, a network abstraction accounting for queues. Indeed, while the same physical path may provide a very different performance to different flows, the queue-level topology shows such differences explicitly: queues reveal useful information about the current demand and network state, and hence allows for a more informed routing which can avoid delays and exploit available bandwidth resources. To cope with dynamic changes in the demand, networks can be reconfigured and routing decisions adapted dynamically, also leveraging priority queuing mechanisms. Source routing, e.g., based on tagging, provides an ideal framework to implement demand-awareness, requiring modifications on the hosts only.

A second key observation of our paper is that rendering networks more dynamic and adaptable does not have to contradict predictability. In particular, if done right, principles of network calculus can still be employed and the resulting performance guarantees along routing paths maintained. That is, networks can be adapted to flows arriving over time while still providing hard guarantees at all times.

Our main ***contribution*** is *Chameleon*, a demand-aware cloud network that combines adaptive source routing with priority queuing to meet both performance *and* resource efficiency objectives. *Chameleon* employs fine-granular routing to leverage path diversity, and relies on an enhanced network abstraction which accounts for

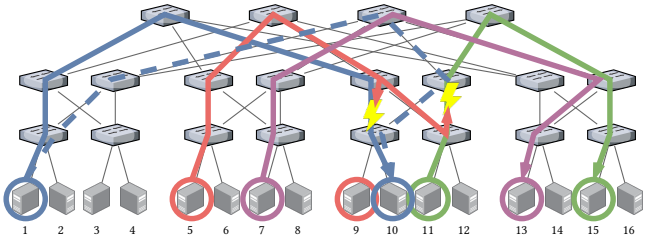


Figure 1: The blue (between virtual machines (VMs) 1 and 10), purple (between servers 7 and 13) and green (between servers 11 and 15) flows are already embedded. In this situation, the red flow (between servers 5 and 9) cannot be embedded by Silo. Rerouting (which Silo does not do) the blue flow on the dashed path would however make space for the new flow and allow to embed it.

queues: the queue-level topology. *Chameleon* dynamically reevaluates routing decisions, performing adjustments while maintaining network calculus invariants to ensure strict latency guarantees are provided and preserved.

In extensive experiments conducted in a testbed based on real data center topologies and using large-scale simulations, we find that *Chameleon* can significantly outperform the state-of-the-art (SoA) both in terms of runtime and achieved network utilization. By exploiting path diversity, priority queuing, and re-evaluations of routes, compared to Silo [31], *Chameleon* is able to admit significantly more flows, and hence increase network utilization and operator revenue, without sacrificing performance guarantees.

Our approach shows that there is an untapped potential for providing strict real-time guarantees when using off-the-shelf technologies. While we build upon several existing techniques, the main contribution of this paper lies in identifying the suitable building blocks and combining them in a clever way to design a complete end-to-end system. In addition, coping with hardware failures is also critical to ensure predictability and provide guarantees in data centers. While a complete discussion of mechanisms to handle failure scenarios is outside the scope of this article, we provide a few pointers on how *Chameleon* can be extended to cope with switch and/or link failures.

In order to facilitate future research and in order to ensure reproducibility of our results, we will make all our code and experimental results publicly available.

2 STATE-OF-THE-ART AND MOTIVATION

A main motivation for our work are the unexploited optimization opportunities available in current networks: SoA networks are operated in a fairly inflexible and demand-oblivious manner. We argue that this can lead to both suboptimal network performance and low predictability of performance (in terms of latency and throughput), which leads to unnecessarily low utilization. In the following, we identify and discuss such missed optimization opportunities. Later in this paper, we will show that it is indeed possible to exploit these opportunities and operate networks in a dynamic and demand-aware manner, without sacrificing predictability.

2.1 The Price of Static Allocation

State-of-the-art approaches for providing predictable network performance have the common feature that they are fairly static: embedding decisions (e.g., related to the route or per-flow rate), once taken, are usually not reevaluated nor adapted later: state-of-the-art solutions are not designed for reacting to flows arriving over time. In environments where networks need to provide guarantees and, hence, perform admission control, this can lead to unnecessarily high network flow rejection rates. For example, if the network configuration chosen earlier does not fit the characteristics of arriving flows, these flows need to be rejected. In contrast, in a dynamic and demand-aware network, it may still be possible to accept these flows, using reconfigurations. To be more concrete, let us consider the two main solutions providing predictable latency in the cloud: Silo [31] and QJump [22].

2.1.1 QJump. QJump [22] relies on information about application performance requirements, related to latency, rate and packet size, at network initialization time. This information is then used to compute the QJump formula: a maximum latency of $2nP/R + \epsilon$. Here, n is the number of applications using the system, P the packet size, R the links rate, and ϵ the cumulative processing time, which is guaranteed to all applications, assuming that they transmit at most one packet per each of these time periods, i.e., at a rate of at most $P/(2nP/R + \epsilon)$ [22]. While the ϵ and R parameters are constant and dependent on the physical topology only, the n and P parameters must be defined upfront, at network initialization time; this is necessary to be able to compute the maximum latency guarantee that the system will provide to flows.

Let us consider a $k = 4$ fat-tree topology with 10 Gbps links and with a cumulative end-to-end processing time of 4 μs. Here we have $R = 10$ Gbps and $\epsilon = 4$ μs. If the network operator decides to authorize 10 VMs per server and packets of at most 300 bytes, we have $n = 16 \times 10 = 160$ applications and $P = 300$ bytes. As a result, the QJump system guarantees a maximum latency of 80.8 μs at a rate of at most 29.7 Mbps for each VM.

While providing predictable performance, these static allocations can lead to unnecessary request rejections and as a result low utilization. For example, even if the network is unused, QJump would in this situation not admit a tenant request for a flow with a latency requirement of 50 μs. Similarly, a request for a 50 Mbps flow would be rejected unnecessarily. If an applications needs much less bandwidth than 29.7 Mbps, say 3.11 Mbps, and tolerates a higher latency guarantee than 80.8 μs, say 772 μs, the system will accept only 160 flows, while 1600 of these flows could have been accepted if the network operator initially decided to define $n = 1600$. Similar observations can be made for applications that send packets smaller or greater than 300 bytes.

This is a real concern, especially in cloud environments where tenant applications are typically unknown and requirements are hard to estimate.

2.1.2 Silo. Silo [31] also provides latency guarantees by leveraging admission control and relying on deterministic network calculus (DNC). At network startup, Silo assigns a delay D_i to each link in the network. Then, upon a new flow request, the admission control logic of Silo uses DNC to calculate the worst-case delays d_i of each link if the flow was to be accepted. The flow is rejected if $d_i > D_i$ for a link i. If a flow can be accepted, its latency guarantee is $\sum_i D_i$ for all links i traversed by the flow. As a result, the number of possible delay guarantees for a given application corresponds to the number

of different paths between the two VMs of the flow, i.e., $(k/2)^2 = 4$ for a fat-tree topology with $k = 4$.

To give an example, we consider a fat-tree topology with $k = 4$. We allocate a delay D_i of $D_R = 20$ μs to rack links, $D_P = 50$ μs to intra-pod links and $D_A = 80$ μs to aggregation links. Without taking detours, there is only one possible delay between any pair of VMs. Between VMs in the same rack, the delay is $2D_R = 40$ μs. Between VMs in different racks but the same pod, the delay is $2(D_R + D_P) = 140$ μs. Between VMs in different racks and different pods, the delay is $2(D_R+D_P+D_A) = 300$ μs. The situation is then similar to QJump: if a tenant needs a latency guarantee lower than these values, say 30 μs, the flow will have to be rejected unnecessarily. Similarly, if tenant applications tolerate higher latency guarantees, say 10 ms, the admission control logic of Silo will start blocking flows to avoid reaching the predefined limits, even though guarantees would still be fulfilled. By increasing the allocated delays at each link, more flows could have been accepted.

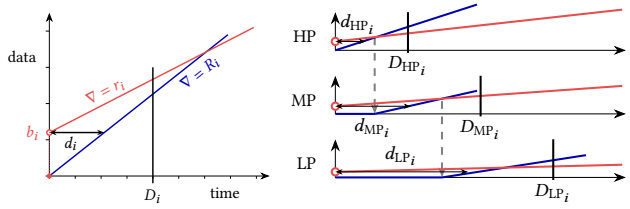
Once Silo embeds a flow, there is no reevaluation of its decision. This is illustrated in Fig. 1. Let us assume that the blue path (between servers 1 and 10), the purple path (between servers 7 and 13) and the green path (between servers 11 and 15) are already embedded. Furthermore, let us assume, for simplicity of the example, that these flows consume the entire capacity on their links. In this situation, the red flow (between servers 5 and 9) cannot be embedded: it is blocked by all the other flows. However, rerouting the blue flow on the dashed path, i.e., reevaluating a decision taken previously, would make space for the new flow and would actually make it possible to admit and embed it.

In ***conclusion***, similarly to QJump, Silo's predictability guarantees can come at the price of low utilization: resource allocation decisions related to link delays and flow embeddings are performed greedily, and never reevaluated again. If applications have requirements that do not match the defined link delays, Silo will reject them while they actually could have been accepted, as we will show. As a result, Silo's resource allocation and embedding approach leads to a bias in terms of the types of flows that can be accepted — and to unnecessary rejections and hence low utilization.

2.2 Unexploited Path Diversity

We see a great potential to exploit path diversity and more fine-grained routing to improve the efficiency and performance predictability of networks. In fact, even for the same physical path, multiple performance characteristics may be experienced: as the switches and routers along the physical route typically have multiple queues, the delay often depends not only on the specific router but also on the specific queue that is traversed. This motivates us in this paper to consider a finer granularity of routing: based on a "queue-level topology" rather than just a "router-level topology".

Surprisingly, SoA solutions do not exploit physical path diversity. For example, QJump's admission control algorithm does not account at all for the specific paths on which flows can be routed. As a consequence, QJump does not reserve network resources per switch or per link, but for the whole network, which can be inefficient. It implies that QJump assumes that two flows, even if they are disjoint, consume the same resources.



(a) Silo: per link. **(b) *Chameleon*: per queue.**

Figure 2: Silo and *Chameleon* modeling for access control. Silo models links while *Chameleon* goes one level lower and defines one model per priority queue. This offers higher delay diversity to applications.

Let us illustrate the problem again with a fat-tree topology with $k = 4$ as in Fig. 1. Taking the parameters as in §2.1, QJump will accept up to 160 flows with delay requirements of at most 80.8 μs and a rate of at most 29.7 Mbps. Let us consider that the 160 flows accepted by QJump are located in the two leftmost pods, which is possible with a simple first-come first-serve VM allocation algorithm. In this case, QJump will reject any new flow request because it reached the maximum of n resources. At the same time, half of the network is unused although it could actually accommodate more flows: the lack of routing knowledge leads to unnecessarily rejections. Similarly, while Silo reserves resources per link, there is no optimization of routes nor of priorities assignment in the network. By not optimizing nor accounting for the routes where flows are embedded, such approaches are not demand-aware, as the network state and performance characteristics depend on the specific route taken.

3 *CHAMELEON* SYSTEM DESIGN

Motivated by the above shortcomings and opportunities, we now describe the design of *Chameleon*, which combines adaptive source routing and priority queuing. The latency modeling of QJump is topology-agnostic and assumes the same traffic envelope for all the flows with deterministic requirements. These fundamental assumptions prevent it from being adapted to solve the above-mentioned shortcomings without a complete redesign of the system. However, similar to QJump, we exploit priority queuing capabilities of today's switches. Furthermore, we build up partly on components of Silo, as explained in this section. *Chameleon* is based on four building blocks: it does access control and delay computation using DNC similarly to Silo, leverages priority queuing like QJump, applies fine-grained source routing, and uses adaptive reconfigurations. We discuss these building blocks in turn and then provide an overview of the control plane.

3.1 Building Block 1: Silo

Like Silo, *Chameleon* leverages three basic components: the resource allocation that is run at network startup (§3.1.1), the access control logic that ensures that flows are embedded only if all delay requirements are satisfied (§3.1.2) and the resource reservation (§3.1.3) responsible for keeping track of resources usage at runtime.

3.1.1 Resource Allocation. Silo keeps track of resource consumption and per-link worst-case delays using DNC. Each link i is assigned a maximum delay D_i. We call this the *resource allocation*, as each link is allocated a delay budget. Then, Silo always makes sure that the DNC worst-case delay d_i of each link remains lower than

its maximum budget D_i, i.e., the admission control of Silo ensures that $d_i \leq D_i \quad \forall i \in \mathcal{G}$, where $\mathcal{G}$ represents the set of links in the network. This is illustrated in Fig. 2a for a particular link. Based on the token-bucket burst and rate parameters of each flow, Silo keeps track of the total burst b_i and rate r_i consumed at each link, forming the DNC arrival curve for this link. The DNC service curve of the link corresponds to the rate R_i of the link. The horizontal deviation $d_i = b_i/R_i$ between these two curves represents the worst-case delay at this link.

3.1.2 Access Control. A new flow request f consists of token-bucket parameters b^f and r^f, and of a maximum delay requirement d^f. At each link where the flow shall be embedded, Silo checks whether adding the token-bucket parameters b^f and r^f to the already used resources b_i and r_i would result in $d_i > D_i$ or . If this does not happen for any of the links supposed to be traversed by the flow and if the sum of all the delays D_i of these links is lower or equal to the delay requirement d^f of the flow, the flow is accepted. Otherwise, it is rejected.

3.1.3 Resource Reservation. When a flow is accepted, its token-bucket parameters are simply added to the b_i and r_i parameters of each link it traverses. Note that, per DNC, the burst b^f requirement of a flow increases at each hop. Indeed, at a switch, the burst of a flow increases for each new packet that arrives while previous packets are still queued. Formally, the burst increases by $r^f D_i$ at each hop; the maximum amount of data that can accumulate while packets are queued. This is taken into account by Silo when checking resource consumption at each link, as well as when reserving resources to account for the embedding of a new flow.

We note that Silo also incorporates a VM placement algorithm. In this paper, we focus on the embedding task and hence assume that a flow already has a source and destination server assigned.

3.2 Building Block 2: Priority Queuing

In order to increase the delay diversity offered to applications, i.e., to offer different service levels, *Chameleon* uses priority queuing. Each output link i now corresponds to n_i priority queues. Any number of queues can be supported by the system, though typical switches usually have up to 8 priority queues [60]. We discuss the impact of the number of queues in §3.3. In order to ease the parallelism with Silo, we present subsequently how the resource allocation, access control and resource reservation mechanisms are changed.

3.2.1 Resource Allocation. Delays D_q are assigned by a resource allocation algorithm to each queue q. The set of different delays that a physical path can offer is now multiplied by the number of combinations of priority levels at each hop.

3.2.2 Access Control. The process is illustrated in Fig. 2b for $n_i = 3$ priority queues. *Chameleon* keeps track of token-bucket resource consumption parameters for each individual priority queue. The service curves offered to each queue are governed by DNC. Non-preemptive priority queuing scheme requires high priorities to wait for at most one packet of a lower priority before being transmitted. As a result, the highest priority queue service curve is identical to the Silo case (i.e., it is the link rate) but is shifted towards the right by $l_{max}/R+\phi$, where l_{max} is the maximum packet size in lower-priority

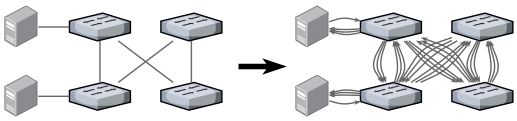


Figure 3: Modeling each individual priority queue leads to more embedding opportunities for a given application. This both increases the delay diversity offered to an application and the complexity of the routing procedure that has to select one particular embedding.

queues, R is the link rate and ϕ is a parameter for accounting for the overhead of the priority queuing implementation in the switch[1]. ϕ is typically in the order of microseconds [60]. The service curve of a low-priority queue then corresponds to the difference between the service curve of the higher-priority queue and the arrival curve of the traffic traversing the latter. This is shown in Fig. 2b.

When trying to embed a flow on a path (of queues), the access control must be slightly adapted to account for the presence of lower-priority queues. Indeed, adding a flow to a queue modifies the service curve offered to lower-priority queues, and could hence violate the $d_q \leq D_q$ constraint for these queues and hence lead to the violation of the guarantees provided to already embedded flows. When checking if a flow can be added to a particular queue, the delay of lower-priority queues also has to be checked. Additionally, the access control must check that the buffer capacity of each queue at the link is not violated. DNC provides a bound on the worst-case buffer occupancy at a system: it corresponds to the vertical deviation between the arrival and service curves. The access control simply checks that this deviation stays lower than the buffer capacity of each queue. In order to reduce the computed bounds, arrival curves are shaped by the rate of the input link where they are coming from.

3.2.3 Resource Reservation. When a flow is accepted, its token bucket parameters are simply added to the b_q and r_q parameters of each queue it traverses. Additionally, the service curves of the lower-priority queues also have to be updated (as described in §3.2.2) to reflect the change in the arrival curve of a higher-priority queue. The burst increase of flows is of course handled in the same way as for Silo.

3.3 Building Block 3: Routing

The introduction of priority queuing changes the path finding problem. Besides the physical path, also priority queues have to be selected. Effectively, this corresponds to finding a path in a topology where each physical link i is duplicated into n_i edges. This is illustrated in Fig. 3 for $n_i = 3$ priority levels. We call this a *queue-level topology*.

In this queue-level topology, each queue/edge q has been allocated a delay D_q by the resource allocation algorithm. Finding a path for a flow request then consists in finding a path $\mathcal{P}$ such that $\sum_{q \in \mathcal{P}} D_q \leq d^f$, and for which the access control allowed access to the flow. If we introduce a *cost function* that defines a metric value for each queue in the network, this corresponds to a delay-constrained least-cost (DCLC) routing problem, a problem for which numerous algorithms have been proposed [23]. The cost

[1] Note that, for computing *per-packet* delays, DNC requires service curves to be shifted down by the maximum packet size of the flow [40, 62]. While our implementation takes this into account, for simplicity, and because this is a very small value, we omit this in our description.

function must be defined in a way that makes the routing algorithm consume the least amount of resources for each flow and hence maximizes the probability of accepting future demands. We will come back to this in §3.4.

As the DCLC routing problem is NP-complete, optimal routing algorithms exhibit too high memory consumption and runtime to be used as online routing algorithms [23]. Hence, a sub-optimal, yet fast and complete algorithm has to be used, e.g., LARAC [33]. Two modifications have to be made: first, the routing algorithm implementation must be modified to check for access control at each queue/edge it visits and should not visit an edge for which it gets denied access; second, this access control depends on the total burst increase that the flow experienced (see §3.1.3), which in turn depends on the complete path followed by the flow up to the considered queue. Algorithmically, this impacts the completeness of the routing algorithm. Indeed, it has been shown that if an edge constraint (e.g., access control) in a routing problem depends on the complete path visited before reaching the subject edge, heuristics lose their completeness property [61]. As a result, the routing procedure is both sub-optimal *and* incomplete. We discuss the impact of this and how we cope with it in the next section.

Increasing the number of priority queues increases the number of edges in the topology. That gives the routing algorithms more options and a greater delay diversity (i.e., more possible end-to-end delay values, see §3.2) and hence increases the potential for accepting more flows. At the same time, routing algorithms scale with the number of nodes and edges in the network and more priorities hence translates to higher request processing time. While switches usually have up to 8 queues [60], *Chameleon* can deliberately decide to use a subset of the available queues to balance the tradeoff between runtime and delay diversity. We will show that in our scalability evaluations (§5.2).

In order to seamlessly cope with $n-1$ independent hardware failures, the routing procedure must be adapted to find n physically disjoint paths. The development of algorithms for finding disjoint routes is still an ongoing research topic [29, 35, 52], but the simplest solution is to run the DCLC routing algorithm n times and, after iteration, remove the links and nodes of the found path from the routing topology. On our queue-level topology, all the queues of the traversed physical links must be removed to ensure physical disjointness.

3.4 Building Block 4: Reconfigurations

When a flow is routed, it is the role of the cost function to direct the routing algorithm such that the least amount of resources is consumed. However, the cost function is not aware of upcoming requests and, as we highlighted in §2.1, a low-cost path for routing a flow f might happen to block a later flow g. Finding a cost function that is good for any network scenario is a challenging problem, as whether a choice now is good for later is only defined by the upcoming flows, which are unknown to the cost function. As we leave a more detailed study of cost functions for future work, we instead use a dummy cost function (e.g., least-delay) and when the routing procedure fails at embedding a new routing request, it can analyze the current network state, reroute already embedded flows to make space for the new flow and then embed the original flow.

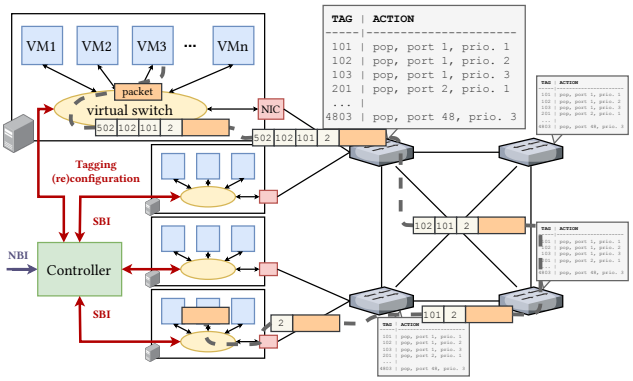


Figure 4: Example of *Chameleon* in operation: VM 3 on the first server sends a packet to VM 2 on the fourth server. Hypervisors in servers tag packets of the different VMs to define the path they take and their priority level at each hop. This enables easy reconfigurations and circumvents traditional issues in distributed network reconfigurations.

We will show in §4.1.4 that the least-delay cost function choice is actually wise and tries to minimize *(i)* the resources consumption of each flow, *(ii)* how often reconfigurations will be needed, and *(iii)* how efficient these reconfigurations will be.

However, reconfiguring running flows constitutes a big challenge. Algorithmically, consistent network updates is a complex task, especially in the presence of strict latency guarantees [19]. It has also been shown that the management interface exposed by existing programmable devices is not always predictable [60], the controller hence being unsure whether its desired configuration update is indeed implemented in the data plane. Furthermore, other measurement studies have shown that updating forwarding rules on programmable devices can lead to transient phases during which packets are forwarded on both paths [38], an unacceptable situation for predictability. Using technologies like MPLS that can smoothly migrate to a precomputed alternative path is not possible, as alternative paths with guarantees cannot be computed in advance without knowing the future network state.

To circumvent these issues, we propose to use source routing for configuring forwarding decisions, similarly to what Microsoft uses in their datacenters [17, 18]. This is illustrated in Fig. 4. Instead of forwarding based on a five-tuple matching, the forwarding elements in the network match on a particular tag in the packet to define their behavior. Each possible tag value corresponds to a port-queue combination. When sending packets from VMs, hypervisors push a stack of tags corresponding to the path the packet has to follow and the priority levels at which it should be enqueued. For example, if a priority level p and output link l correspond to tag $t = 100l + p$, a stack of tags 101, 503 means that the packet should be forwarded to port 1 of the first switch and with priority 1 and then to port 5 of the second switch with priority 3. The switches simply match on the tag to perform the corresponding action and then pop the outermost tag out of the stack to permit the next-hop switch to read the next tag.

This approach solves the above-mentioned issues. Indeed, the forwarding behavior of switches is configured once at startup and never has to be updated. This eliminates the unpredictability problem of the management interface of switches and the transient phase issue when updating flow tables. Further, routes are configured on end-hosts, which eliminates the problem of consistently

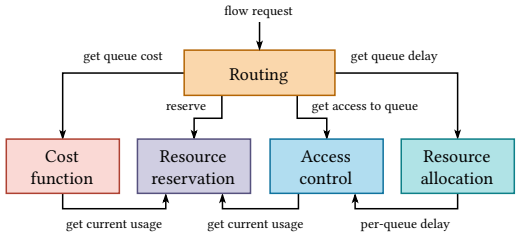


Figure 5: *Chameleon*'s control plane architecture. All these components run in a centralized controller. Flows are then configured as shown in Fig. 4.

updating the network configuration, as the network is configured centrally.

This however brings another important challenge. The tagging in the hypervisor virtual switch, and the updates of its tagging rules must be predictable, as this new component adds an additional delay to the packets. However, in contrast to blackbox forwarding devices based on closed implementation, the virtual switch is under our complete control. That allows us to specifically design it for satisfying these requirements. In particular, several recent technologies such as SmartNICs, P4, and data plane development kit (DPDK) offer the potential for achieving this predictability. We describe in detail in §4.2.2 how we implement the tagging component and confirm in §5.3 that this implementation is predictable and fast enough for predictable latency use cases.

We note that changing the route of existing traffic can lead to out-of-order packets, which can in turn lead to spurious retransmissions and throughput decrease for TCP-like transport protocols. While studying the impact of reconfigurations on TCP-like congestion control algorithms is outside the scope of this article and is part of our future work, we will see in §5.4 that our testbed evaluations (that include many reconfigurations of TCP flows) do not lead to any reordering of packets.

3.5 Control Plane Architecture

The architecture of *Chameleon*'s control plane logic is summarized in Fig. 5. First, delays are assigned to each queue by a *resource allocation* algorithm (§3.2.1). The *routing* module, which receives flow requests, uses these values as constraints for its DCLC problem (§3.3). The cost values are defined by a *cost function* based on the state of the network, i.e., based on which flows are embedded where. The routing module is also responsible for rerouting flows if that is necessary to embed the new flow (§3.4). The state of the network is managed by the *resource reservation* module, which updates the usage of the network when the routing module registers the embedding of a new flow (§3.2.3). To ensure that it only embeds flows that do not lead to any delay violations, the routing procedure relies on the *access control* module (§3.2.2). The latter accepts or denies access requests so that the per-queue delays assigned by the resource allocation algorithm are never violated. In the next section, we describe how all these elements are implemented.

4 *CHAMELEON* IMPLEMENTATION

We separate our description of the implementation of *Chameleon* into the control plane and the data plane parts.

4.1 Control Plane

The control plane is implemented as a multi-threaded set of Java 8 libraries implementing all the controller functionalities. The code consists of around 30k lines of code.

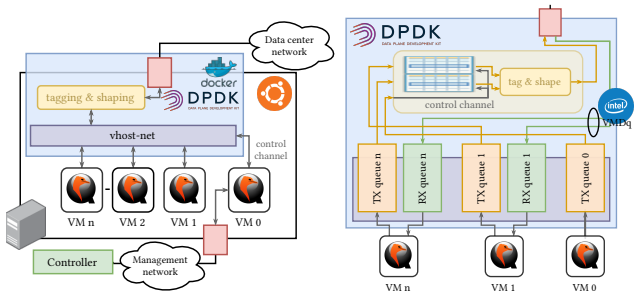
4.1.1 Interfaces. The controller implements a northbound interface (NBI) that exposes a representational state transfer (REST) application program interface (API) to users (Fig. 5). This API allows tenants, VMs, and flows to be created and deleted through hypertext transfer protocol (HTTP) *POST* requests. A tenant is a logical abstraction that supports users to create flows between VMs that they created, i.e., VMs of the same tenant. All created VMs are identical and allocated to a randomly selected physical server. VM placement is outside the scope of this work. The creation of a flow requires the specification of source and destination VMs, of burst, rate, and latency requirements, and of a five-tuple matching structure. A counterpart southbound interface (SBI) module implements the OpenFlow (OF) 1.0 protocol (Fig. 5). The module discovers the network topology at startup using link-layer discovery protocol (LLDP) packets and configures the static forwarding rules on switches (see §4.2). Upon a VM creation request, the SBI module triggers the actual creation on the chosen server via secure shell (SSH). Upon a flow embedding request, the NBI module forwards the request to the routing procedure. If the routing procedure returns an embedding, or if it requests the reconfiguration of a previous embedding, the SBI configures the corresponding tagging rules on the source server via SSH (§4.2). We do not aim at providing strict guarantees for request processing times. As a result, the communication between the SBI and the servers does not need latency guarantees and can happen over a traditional control network.

4.1.2 Resource Allocation and Reservation, and Access Control. The resource allocation simply assigns a maximum delay to each queue upon discovery of a new link (as described in 3.2.1). For 8-queue ports, it assigns the following delays: 0.1 ms, 0.5 ms, 1 ms, 1.5 ms, 3 ms, 6 ms, 12 ms, and 24 ms. For 4-queue ports, it assigns 0.1 ms, 1 ms, 6 ms, and 24 ms. For 2-queue ports, it assigns 0.1 ms and 6 ms. Host ports towards their access switches are assigned 0.5 ms. Choosing these parameters defines the sets of delays that a given physical path can offer, including in particular the minimum delay that can be achieved over that path. For example, with the above assignment, a 4-hop path will not be able to provide a delay guarantee lower than 0.4 ms. Also, these values impacts the number of flows that can be accommodated at a given queue, as flows will be rejected as soon as the delay they induce at a queue reaches its assigned maximum delay. Delay values for a given number of queues are chosen to be a superset of the values for lower number of queues in order to ensure that, for a given number of queues, we offer *at least* the same delay diversity as for lower number of queues. The somewhat arbitrary delay assignment above is chosen to be able to span delay requirements from sub-milliseconds to hundreds of milliseconds and try to maximize the number of flows that can be accepted in slower queues. A complete sensitivity analysis and an optimization of the delay assignment would be interesting but is outside the scope of this article. Access control and resource reservation are implemented as described in §3.2.2 and §3.2.3.

```
1:  function EMBEDDINGSTRATEGY(request)
2:      response ⟵ ROUTE(request)
3:      if response ≠ NULL then
4:          RESERVE(response), return response
5:      for each flowToReroute in LIM(SORT(GETFLOWSTOREROUTE(request))) do
6:          INCREASEGRAPHCOSTS(flowToReroute, request)
7:          reroutingResponse ⟵ ROUTE(flowToReroute)
8:          if reroutingResponse ≠ NULL then
9:              RESERVE(reroutingResponse)
10:             FREE(flowToReroute.originalPath)
11:             response ⟵ ROUTE(request)
12:             if response ≠ NULL then
13:                 RESERVE(response), return response
14:     return NULL
```

Figure 6: Pseudo code of the flow embedding and reconfiguration.

4.1.3 Routing and Rerouting Strategies. The routing procedure for finding a DCLC embedding is implemented using the LARAC algorithm [33] as described in §3.3. The complete routing and rerouting logic is shown in Fig. 6. First, the procedure tries to find a path for the flow request using a least-delay search (line 2). If it fails at finding a valid embedding (either because of its incompleteness – §3.3 – or because of previous flows poorly embedded – §2.1), the procedure tries to reroute already embedded flows to make space for the new one. First, in line 5, it selects a set of sorted candidate flows to be rerouted and iterates through them. In our implementation, it selects all the flows traversing at least one edge of any of the equal-length shortest paths (SPs) in the physical topology from the source server to the destination server of the new flow to embed. Those paths are found using Yen's algorithm [67]. Other flows not traversing these SPs are indeed not expected to prevent the new flow from being embedded. In our implementation, we sort flows according to the number of physical links they share with the SPs of equal length in the physical topology between the source and destination servers of the new flow to embed. This sorted list of candidate flows is then truncated to its first n elements to limit the maximum number of (re-)routing retries and hence to mitigate the runtime increase caused by the rerouting. In our implementation, we choose $n = 20$, as small benchmarks showed that most successful reroutings happen in the very first flows. Then, for a given candidate flow to reroute, based on the current state of the network, the procedure tries to re-embed the selected flow. In line 6, to direct the routing procedure toward a path that potentially allows the new flow to be embedded, we increase the cost (see §4.1.4) of the previous queues in which the flow was embedded (to move it somewhere else) and of all the queues of all the equal-length SPs between the source and destination servers of the new flow to embed (to prevent the rerouted flow to interfere with the new flow). The cost is increased by multiplying the original cost value by an arbitrary high value (30 000 in our implementation). If the flow cannot be re-embedded, the procedure continues to the next candidate flow. If the flow can be re-embedded, the procedure notifies the SBI to reserve the new embedding (line 9) and then to free the resources reserved for the previous embedding (line 10), and then retries to add the new flow. If that fails, the procedure continues to the next candidate flow to reroute. If that succeeds, the new flow is successfully embedded thanks to the reconfiguration of previous flows. If the truncated list of candidate flows is exhausted without any success, embedding failed and the new flow is rejected (line 14).



(a) Server. **(b) DPDK application.**

Figure 7: *Chameleon*'s end-host implementation (a) and zoom in the DPDK application. QEMU VMs are connected to the DPDK application using a *vhost-net/virtio-net* architecture and communicate through distinct receive and transmit queues. A control VM allows the configuration of the tagging/shaping rules.

Because failure handling is outside the scope of this work, we did not implement a disjoint routing strategy as described in Sec. 3.3. However, that would be a simple extension to the routing procedure.

4.1.4 Cost Function. As described in §3.4, the design of a good cost function is a very challenging task. In some sense, the proposed rerouting strategy is a way of adapting the cost function to future requests, as we reroute an old flow with the knowledge of the flows that were accepted later. Unfortunately, also the rerouting procedure needs a cost function. We decide to simply use the *delay* as cost, thereby effectively degenerating the DCLC problem into a simple least-delay routing problem solved by Dijkstra's algorithm [14]. The rationale behind this decision is that the burst increase of a flow at each hop is proportional to the delay of this hop (see §3.1.3). This cost function hence minimizes the resource (burst) consumption of flows and accordingly attempts to drive the routing algorithms towards clever decisions that minimize the number of costly reconfigurations that will be necessary later.

4.2 Data Plane

We consider 1 Gbps OF 1.0 devices: Dell S3048-ON and S4048-ON (four priority levels per port), and Pica P3297 (eight priority levels) switches. We use servers running Ubuntu 18.04 (4.15.0-66-generic kernel) with 64 (Dell servers) or 128 GB (Dell and Supermicro servers) of RAM, an Intel Xeon Silver 4114 @ 2.2 GHz (20 cores, Supermicro servers) or an Intel Xeon E5-2650 v4 @ 2.2 GHz (24 cores, Dell servers) as CPU, and an Intel X550 (Supermicro servers) or X540 (Dell servers) network interface card (NIC) towards the data network. We use virtual local area network (VLAN) tags to implement the tag stacks. While any other stackable tagging mechanism can be used (e.g., multi-protocol label switching (MPLS)), we use VLAN tags because of its low header overhead and its more widespread support. It has also been shown that matching on VLAN tags to output to a particular port and queue and popping the outermost VLAN tag can be done at line rate and with a predictable performance [60].

In the following, we describe the end-host implementation (Fig. 7), the cornerstone of our solution. This consists of a *tagging* part, responsible for pushing tag stacks to packets, and of a *shaping* module, responsible for ensuring that applications do not exceed their negotiated b^f and r^f token-bucket parameters. We implement the virtual switch of the VMs hypervisor as a DPDK 19.08 application running

in a privileged *docker* container. This approach of implementing packet processing NIC features in a DPDK software application is analogous to SoftNIC [25], as used for example by Google for implementing scalable traffic shaping at the edge [54]. The general architecture of the virtual switch is shown in Fig. 7a. The different VMs run in QEMU 2.11.1 with KVM. The VMs and the DPDK application are connected through *virtio* using a *vhost-net/virtio-net* para-virtualization architecture [3].

4.2.1 How to Ensure Predictability? The processing of the virtual switch has to be predictable, in terms of latency. To do so, we use mechanisms similar to those used by FairNIC to implement isolation on a SmartNIC [21]. The DPDK application is pinned to specific cores of the server and we prevent the kernel of the server to use these cores using the kernel *isolcpus* parameter. To avoid unpredictable performance variations, we further disable hyper-threading, turbo-boost, and power saving features of the central processing unit (CPU). This ensures that the DPDK application runs isolated on dedicated CPU cores that operate at a constant and stable speed. We use Intel's cache allocation technology (CAT) to allocate a specific portion of the CPU last level cache (LLC) to the cores used by the DPDK application. As level-one and level-two caches are per-core, this prevents other applications from interfering with DPDK through the memory caches. We use three cores for the application: one sending core, one receiving core, and one master core for the DPDK master process. Both sending and receiving cores process batches of packets for the different VMs in a round-robin fashion. Each VM is assigned a sending and a receiving queue (see Fig. 7b). The sending queues are filled by the VM *virtio* drivers and emptied by the sending core, which is then responsible for tagging and shaping before sending out the packets to the NIC. The receiving queues are filled by the receiving core. The destination VM of a packet is identified by its MAC destination address and VLAN tag. Doing this separation in software would prevent batch processing, a major enabler of the fast software processing performance of DPDK. Indeed, a series of packets received from the NIC is not necessarily entirely destined for the same VM. Hence, we use the virtual machine device queues (VMDQ) technology of Intel NICs. Packet separation is done in hardware and packets for the different VMs are automatically stored in separate physical queues that are then simply pulled by the receiving core and sent to the different VMs *virtio* drivers.

4.2.2 Tagging. The sending core, after pulling a batch from a VM sending queue, is responsible for tagging the packets. The program maintains tagging rules with the following fields: *protocol*, *source IP*, *destination IP*, *source port*, *destination port*, *number of tags to push*, *tags to push*. The entries are stored in a two-dimensional array indexed by the VM ID and the rule ID for a given VM. The maximum number of VMs (64 in our implementation) and of rules per VM (3 in our implementation) is fixed and the array hence does not require dynamic memory allocation. Within a processed batch, for each packet, the core traverses the 3 rules of the VM it is currently serving. If a five-tuple match is found, the tags stored in the corresponding entry are directly copied between the Ethernet and Internet protocol (IP) headers of the packet. If no match is found, the packet is dropped. Once all packets of a batch are processed, the program sends the batch of tagged packets to the NIC. In order to handle failures, the tagger must be extended to duplicate each packet and tag the different physically disjoint paths selected by the routing procedure on the different copies of the packet. While TCP automatically removes duplicates at the receiver, a duplicate removal strategy would have to be implemented on the receiving core for other transport protocols.

4.2.3 Shaping. The sending core must ensure that flows do not exceed the token-bucket parameters that have been reserved for them. Indeed, a violation of these parameters invalidates all the DNC computations and can lead to delay guarantees violations. We add four fields to the tagging rules: *rate*, *burst*, *number of tokens*, and *last timestamp*. The two first fields store the token-bucket parameters of the entry, the third and fourth fields store the number of tokens in the token bucket when they were last computed and the corresponding timestamp. For each packet within a processed batch, the sending core computes the updated number of tokens based on the current timestamp and the rate parameter of the token bucket. The packet is dropped if there are not enough tokens for sending the packet. Otherwise, the number of tokens corresponding to the packet size are removed, the timestamp is updated and the packet is kept. Timestamps are obtained using the timestamp counter (TSC) register of the CPU. Having disabled the turbo-boost and power-saving features of the CPU ensures that this counter measures real time and not simply the number of instructions.

4.2.4 Configuration of Tagging/Shaping Rules. In order to communicate with the virtual switch without creating unpredictability and synchronization issues, we use an additional VM, the control VM (with ID 0). This VM is not allocated a receiving queue (see Fig. 7b). The packets sent by this VM are used to configure the rules stored in the sending core. When the sending core receives a control packet, the first two bytes of the packets are used to index the table – they correspond to the VM ID and rule ID to update. The next bytes in the packet are simply copied in the entry. The *Chameleon* controller connects to this control VM to update tagging/shaping entries and is hence responsible for sending the appropriate values in the correct order and endianness. The DPDK application then simply reinitializes the *number of tokens* and *last timestamp* fields of the modified entry.

5 EVALUATION

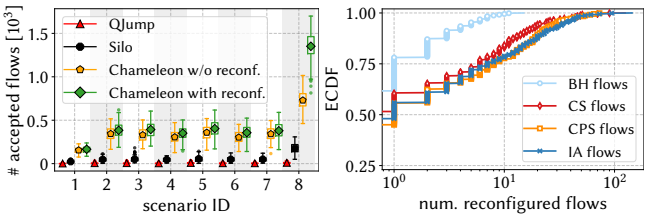
The goal of our evaluation is to show that *Chameleon* successfully provides latency guarantees, can reach higher network utilization than existing approaches, and scales to networks with tens of thousands of servers. First, §5.1 evaluates the utilization and number of flows our system can accommodate by running simulations of its admission control. For different types of traffic distributions, we show that *Chameleon* reaches higher network utilization and number of accepted flows than the SoA QJump [22] and Silo [31] systems. Second, in §5.2, we quantify the scalability of *Chameleon* by evaluating its performance for increasing network sizes. We show that, despite its higher complexity, *Chameleon* achieves better performance than its SoA counterparts for increasing network sizes, both in terms of number of accepted flows and runtime. Then, in §5.3, we perform a microbenchmark of our end-host tagging and shaping implementation. We show that our implementation is

Flow description	Rate	Burst	Deadline
Category 1: Industrial applications (IA) [1, 34]			
Database operations	[300, 550] Kbps	[100, 400] byte	[80, 120] ms
SCADA operations	[150, 550] Kbps	[100, 400] byte	[150, 200] ms
Production control	[100, 500] Kbps	[100, 400] byte	[10, 20] ms
Control and NTP	[1, 100] Kbps	[80, 120] byte	[10, 20] ms
Category 2: Clock synchronization (CS) [51]			
PTP	[1, 220] Kbps	[80, 300] byte	[2, 4] ms
Category 3: Control plane synchronization (CPS) [2, 55]			
Eventual consistency	[2, 4] Mbps	[80, 140] byte	[50, 200] ms
Strict consistency	[5, 8] Mbps	[1000, 3000] byte	[50, 200] ms
Adaptive consistency	[2, 4] Mbps	[80, 120] byte	[50, 200] ms
Category 4: Bandwidth-hungry applications (BH) [4, 5, 45, 65]			
Hadoop, data-mining	[100, 150] Mbps	[1000, 5000] byte	[10, 100] ms
Hadoop, data-mining	[100, 200] Mbps	[1000, 3000] byte	[10, 100] ms
Hadoop, data-mining	[80, 200] Mbps	[1000, 3000] byte	[50, 100] ms

Table 1: Considered flow types and their characteristics.

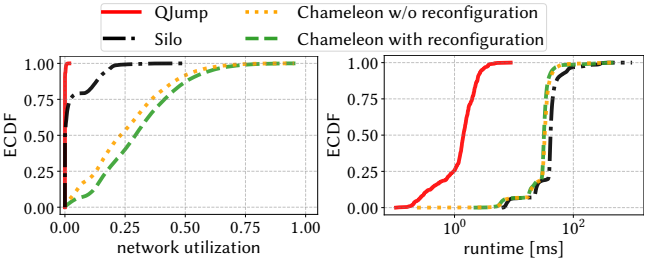
Scenario ID	Distribution (IA, CS, CPS, BH)	Scenario ID	Distribution (IA, CS, CPS, BH)
1	(0.25, 0.25, 0.25, 0.25)	2	(0.2, 0.2, 0.5, 0.1)
3	(0.2, 0.5, 0.2, 0.1)	4	(0.5, 0.2, 0.2, 0.1)
5	(0.1, 0.4, 0.4, 0.1)	6	(0.4, 0.1, 0.4, 0.1)
7	(0.4, 0.4, 0.1, 0.1)	8	(0.33, 0.33, 0.33, 0.01)

Table 2: Flow request distributions used in the simulation.



(a) Num. accepted flows. **(b) Num. reconfigured flows.**

Figure 8: Simulation results. (a) indicates the increased number of accepted flows in *Chameleon* compared to the SoA systems and (b) shows the number of reconfigured flows per flow type and the effect of the characteristics of flows on their reconfigurability. Whiskers show the 1% and 99% percentiles.



(a) Network utilization. **(b) Runtime.**

Figure 9: (a) Improved network utilization achieved by *Chameleon* compared to QJump and Silo. (b) Runtime of embedding one flow in the network.

accurate in shaping flows, can tag packets at high rates, and has a low memory footprint. Finally, in §5.4, we deploy the *Chameleon* system in a testbed composed of ten switches and eight servers. We show that *Chameleon* can improve the performance of applications that run on a shared infrastructure and that the guaranteed packet delays are indeed not violated.

5.1 Network Utilization

We conduct a comprehensive simulation study comparing *Chameleon* with the two main SoA approaches for predictable latency: QJump [22] and Silo [31]. We consider a $k = 4$ fat-tree topology with 1 Gbps physical links and 16 servers.

5.1.1 Configuration of the Systems. For *Chameleon*, at each physical port, we consider 8 queues each with 97 KB buffer size, according to results from our previous work reporting on the per-queue available buffer capacity for SoA switches [60]. For Silo, because it does not use priority queuing, we set a single queue with 590 KB of buffer size, still according to our previously published measurements [60]. We set the Silo per-link delay to 0.1 ms[2]. For QJump, we have $R = 1$ Gbps, $\epsilon = 4$ μs [22], we consider the maximum packet size $P = 1500$ byte, and we set[3] $n = 32$.

5.1.2 Simulation Setup. We define a set of flow requests as an input to evaluate the performance of the different systems. A flow request is defined by its source and destination nodes, and requested rate, burst, and deadline. We choose the source and destination of each flow request randomly from the hosts in the network. To specify the rate, burst, and deadline values, we define different types of application categories: industrial applications (IA), clock synchronization (CS), control plane synchronization (CPS), and bandwidth-hungry (BH) applications (see Tab. 1). Each category of application is defined by a set of distributions for rate, burst, and deadline values according to SoA references as reported in Tab. 1. This allows us to define a wide range of different scenarios and confirm that *Chameleon* performs well under any scenario. To randomly sample flow requests, we use a flow request distribution (a, b, c, d), where a, b, c, and d are the probabilities of a flow to belong to the IA, CS, CPS, and BH categories. For example, the flow distribution of scenario 1 in the Tab. 2 indicates that the probability of having a flow request from each category is the same and is equal to 0.25. After that, for a given flow category, we randomly select one of the distributions of this category and then randomly sample the rate, burst, and deadline values uniformly within the ranges defined in Tab. 1. We define eight different scenarios as shown in Tab. 1. For each scenario and system, we perform 100 runs for which we add flows until a rejection happens. The simulation was performed on a VM equipped with 48 cores and 320 GB RAM, running Arch Linux x64 (kernel version 5.4.15-arch1-1) hosted on a server with 500 GB RAM, a 48-core CPU Xeon CPU-E5-2697 v3 @ 2.6 GHz (2 sockets), and running Proxmox 6.1-7.

5.1.3 Results: Number of Accepted Flows. The comparison of number of accepted flows is shown in Fig. 8a for all the scenarios described in Tab. 2. We consider two cases for the *Chameleon* admission control system, with and without reconfiguring previously embedded flows. Yet, in the case without reconfigurations, *Chameleon* is able to accept between 2× and 10× more flow requests compared to the two SoA approaches. Additionally enabling reconfigurations allows to accept even more flow requests. The big performance difference between the SoA and *Chameleon* is due to

[2]We compared the performance of Silo with each of the delays we used for *Chameleon* (see §4.1.2) and selected the best performing value.
[3]Again, we evaluated QJump with different n values and we chose the best performing one.

the flexible and demand-awareness design of *Chameleon*, while SoA relies on static and greedy decisions (see §2). In particular, QJump is blocked to a maximum of $n = 32$ flows because of the necessity to define n beforehand.

We observe that the benefit provided by reconfigurations depends on the traffic distribution. For instance, scenario 8 in Fig. 8a benefits more from reconfigurations than scenario 1. In fact, as depicted in Fig. 8b, flow types appear to exhibit different levels of reconfigurability. In particular, Fig. 8b shows that flows from the BH category are reconfigured less than the other flow types. This is due to the fact that the rate and burst of BH flows are significantly higher than other types, hence having less chance to be reconfigured (especially in a highly utilized network, see Fig. 9a). However, in addition to rate and burst, flow deadline plays an important role in the reconfigurability of the flows. For example, in Fig. 8b, it can be seen that CS flows have been reconfigured less than IA and CPS, mostly due to their tight latency requirements.

It is worth to note that according to Fig. 8b, although the reconfiguration operations bring a great benefit in terms of number of accepted flows, we only reconfigure a few percent of the accepted flows (less than 100 flows on average).

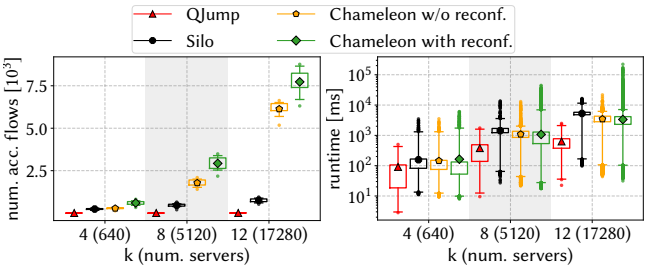
5.1.4 Results: Network Utilization. Fig. 9a shows the empirical cumulative distribution function (ECDF) of the network link utilizations achieved by the different systems for all the considered scenarios. Note that we excluded the host to top-of-rack switch links from the figure. It can be seen that *Chameleon* is able to significantly increase the network utilization compared to other approaches, to reach close to line rate utilization for some links. High utilization and predictable latency are hence not anymore exclusive objectives. For network operators, that means *Chameleon* has the potential of achieving greater revenue.

5.1.5 Results: Runtime. Fig. 9b depicts the comparison of runtime for embedding one flow request in the network for the different systems. We measure the time between a flow request and the reception of a response (whether positive or not). For *Chameleon*, this includes routing and reconfiguration operations. We observe that, despite the greater complexity in *Chameleon*'s logic, it achieves better runtime performance than Silo at the median. This is due to the fact that Silo runs a DCLC algorithm for finding the SP satisfying the delay requirement while *Chameleon* simply runs a Dijkstra least-delay search. Even at the tail, *Chameleon* runs faster than Silo. Because of its pre-assignment of all its decision parameters, QJump exhibits a much lower runtime.

5.2 Scalability

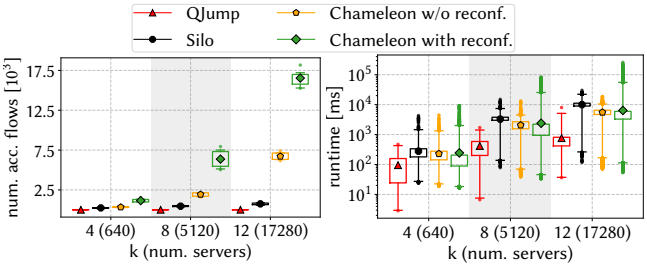
We extend our simulation study from §5.1 to assess the scalability of *Chameleon* compared to Silo and QJump in terms of number of accepted flows and runtime. Considering 40 servers per rack, we vary the k parameter of our fat-tree topology from $k = 4$ (640 servers) to $k = 12$ (17280 servers).

For *Chameleon*, we additionally vary the number of queues the system can use at each physical port. We want to show that even when many queues are available, *Chameleon* can be configured to use less queues in order to reduce runtime, but at the price of



(a) Num. accepted flows. **(b) Runtime.**

Figure 10: Scalability analysis with two queues. (a) Improved number of accepted flows achieved by *Chameleon* compared to QJump and Silo. (b) Runtime of embedding one flow in the network. Whiskers show the 1% and 99% percentiles.



(a) Num. accepted flows. **(b) Runtime.**

Figure 11: Scalability analysis with four queues. (a) Improved number of accepted flows achieved by *Chameleon* compared to QJump and Silo. (b) Runtime of embedding one flow in the network. Whiskers show the 1% and 99% percentiles.

reduced number of accepted flows. We use 4 and 2 queues with per-queue buffer sizes of 190 K and 356 KB respectively, still according to results from our previous work reporting on the per-queue available buffer capacity for Pica8 switches [60]. Values for Silo and QJump are the same as those used in §5.1.

The setup is identical to the one used in §5.1. We focus on scenario 8 from Tab. 1. For each scenario and system, we perform 10 runs for which we add flows until a rejection happens.

5.2.1 Results: Number of Accepted Flows. The comparison of number of accepted flows for different network sizes is shown for two queues in Fig. 10a and for four queues in Fig. 11a. Note that, because Silo and QJump do not exploit queues, their performance does not depend on the number of queues. We observe that *Chameleon* can accept up to 15× more flows requests than Silo, reaching up to 16000 flows for $k = 12$ and four queues. The benefit of *Chameleon* compared to the SoA increases with the network size and the number of queues. The bigger the network gets and the more queues *Chameleon* can use, the more it can optimize its routing decisions to fit in more flows compared to the SoA. Additionally, the benefit of reconfigurations also increases with the network size and the number of queues. For $k = 12$ and four queues, reconfigurations enable *Chameleon* to more than double the number of flows it can accommodate in the network.

5.2.2 Results: Runtime. Fig. 10b and Fig. 11b show the runtimes for embedding per flow request for the different network sizes. Despite the greater complexity of *Chameleon*, it achieves, both with and without reconfigurations, better runtime performance than Silo at both the median and the average. We observe that all approaches

scale exponentially with the network size. While Silo reaches up to 10 seconds on average and 30 seconds at the tail for $k = 12$, *Chameleon* needs up to 9 seconds in average, and 20 seconds at the tail without reconfigurations, or up to 3 minutes with reconfigurations. This shows the interesting tradeoff between the number of accepted flows (i.e., network utilization) and runtime that reconfigurations bring. Without reconfigurations, *Chameleon* still performs better than Silo, both in terms of runtime and number of accepted flows. Adding reconfigurations leads to a significant increase in the number of accepted flows, however, at the cost of an increased tail for the runtime. Limiting the number of reconfigurations allows to navigate this tradeoff. Note that while such runtime values can seem high for data center scenarios, we are here considering the embedding of typically long-lived flows (e.g., synchronization flows staying significantly longer in the network than the embedding time) with very strict delay requirements, in contrast to typical data center applications that run for very short amount of time and that have looser requirements. Additionally, an embedded flow can also actually consist of a long-lived aggregate of short micro-flows with identical latency requirements and that together respect a given traffic envelope over time. In summary, *Chameleon* performs better than Silo both in runtime and number of accepted flows and the difference in number of accepted flows can be further increased by allowing reconfigurations, which only impact the runtime of the system at the tail.

5.3 Tagger/Shaper Microbenchmark

5.3.1 Tagger. We connect two Dell servers (for specifications, see §4.2) directly using 10 Gbps interfaces. In the source server, we deploy a VM generating traffic using the *MoonGen* [15] traffic generator. This generated traffic is pulled in batches through the *virtio* virtual interface by the DPDK application. The combination of parameters outlined in Tab. 3 is used to create the evaluation scenarios. We measure the rate of traffic generated by the VM/MoonGen, pulled by the tagger, tagged and forwarded to the NIC. These values are obtained through simple packet counters in the DPDK application. The number of packets is converted into rate using the rate measured at the destination interface (not connected to DPDK) using *tcpdump*. Fig. 12a shows the generation, tagging and line rate for the different scenarios, ordered by packet size. We observe that the DPDK application is fast enough to tag every pulled packet from the VM, reaching up to 40 Gbps in some cases. All the tagged packets are successfully sent to the NIC. We see that the tagging rate is either bounded by the physical link rate (10 Gbps) or by the rate achieved by the traffic generator. Hence, the tagging implementation is never the bottleneck.

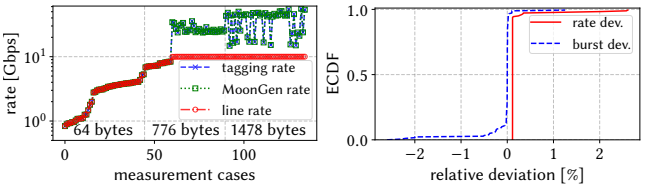
5.3.2 Shaper. We connect a Dell server directly to an Endace data acquisition and generation (DAG) 7.5G4 measurement card [42] through a 1 Gbps connection. We configure our DPDK application to pull packets one-by-one (batch size of one) and to add 6 tags to them. Using the parameters in Tab. 4, we deploy a number of VMs running *MoonGen* and generate traffic towards the DPDK application with the corresponding packet size. Based on the traces obtained by the measurement card, the actual shaped rate and token bucket size are determined. We calculate the rate (resp. burst) deviation as the relative deviation of the observed shaped rate (resp. burst) compared to the value defined in the shaping rule. As can be observed in Fig. 12b, *Chameleon*'s shaper implementation exhibits a precise performance, producing a maximum relative error of around 2%. Also, although not shown in the figure, we observe that shaping is more precise for a lower number of VMs. This is because the DPDK application pulls packet in a round-robin fashion from VMs: having less VMs leads to shorter pulling intervals.

Parameter	Values	Parameter	Values
Packet size [byte]	64, 776, 1478	Batch size	1, 16, 32
Num. flows	1, 2, 3	Num. tags	2, 4, 6, 8, 10

Table 3: Considered parameters for the tagger evaluation.

# VMs	# Flows	Packet size [byte]	Rate [bps]	Burst [bits]
10	3	78	10^5	10^5
10	1	800	10^7	$10^4, 10^5, 10^6$
5	3	800, 1522	10^7	10^5
5	1	78	$10^3, 10^7$	$10^3, 10^4, 10^5$
1	3	800	$10^5, 10^7$	10^5
1	1	78, 800, 1522	10^7	10^6

Table 4: Measurement scenarios for the shaper evaluation.



(a) Tagging performance. **(b) Shaping precision.**

Figure 12: Performance evaluation of the tagger and shaper implementation of *Chameleon*.

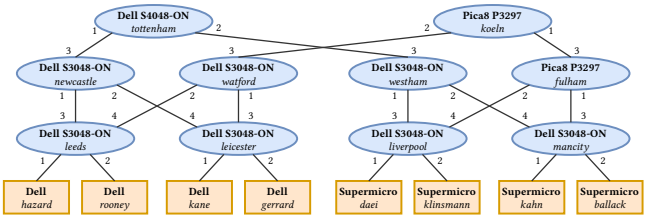
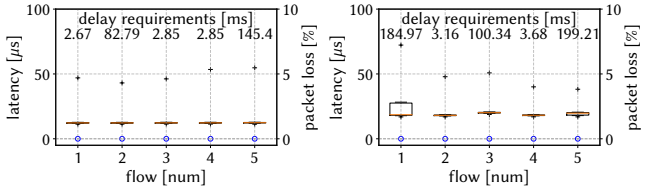


Figure 13: Testbed for our experiments.

5.3.3 Resources Consumption. Because the DPDK application runs three non-blocking processes pinned to three different cores (see §4.2.1), it consumes exactly three CPU cores. The application allocates most of the memory it needs at startup, and allocates a couple of additional buffers at each VM connection. We measured the memory consumption of our application using the *docker* API. The application consumes around 19.2 MB plus around 0.02 MB per connected VM. A couple of additional KB are necessary when transmitting batches, but those are directly freed. This is a very low memory footprint.

5.4 Testbed Measurements

We verify the *Chameleon* system in a $k = 4$ fat-tree testbed (Fig. 13). The *Chameleon* controller connects to the servers and switches through a management network not shown in the figure. In the first experiment (§5.4.1), we confirm that the delays guaranteed by *Chameleon* are indeed not violated throughout the whole lifetime of flows, even when flows are rerouted or new flows are embedded. In the second experiment (§5.4.2), we illustrate that *Chameleon*



(a) 4-server experiment. (b) 8-server experiment.

Figure 14: End-to-end latency measured for 5 different flows. The crosses depict the maximum observed latencies. Whiskers of the boxplots show the 10% and 90% percentiles.

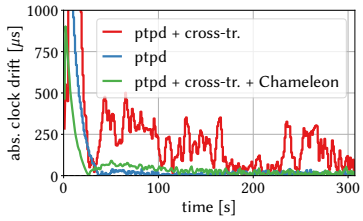


Figure 15: ptpd experiment. *Chameleon* resolves the interference introduced by sharing the network with bandwidth-hungry applications like Hadoop.

helps resolving network interference and can provide guarantees to applications even in presence of adversarial traffic.

5.4.1 Verification of E2E Latency Guarantees. In this part of our evaluation, we run the complete *Chameleon* system in the testbed depicted in Fig. 13. We consider two scenarios, in the first one we use the full testbed (with 8 servers), while in the second one, we use only the left pod of our topology (4 servers). To perform our experiments, we consider 31%, 31%, 31%, and 11% of, IA, CS, CPS, and BH applications and generate flow requests as in §5.1. These particular scenarios accepted a total of 298 and 218 flow requests. Using two network taps mirroring traffic to an Endace DAG 7.5G4 measurement card [42], we measure the packet delay experienced by five random accepted flows, while ensuring that at least one of these flows was reconfigured. During the various reconfigurations, no reordering of packets was observed. Fig. 14 presents the observed end-to-end packet delay of the selected flows in both scenarios. It can be seen that the required delays of flows are met and there is no packet loss occurring in the system. There is very little queuing happening: most packets experience the same delay (only due to processing in the switches) and only a few packets are delayed due to queuing. This shows that, to keep queues nearly empty, a very conservative approach like QJump (which allows to send at most one packet at the same time in the network) is not necessary and that a precise DNC modeling can achieve low and predictable latency while still achieving high utilization.

5.4.2 Resolving Network Interference. Precise clock synchronization is often a requirement of distributed systems [11]. In local area networks (LANs), the precision time protocol (PTP) is a master–slave protocol that is widely used for clock synchronization. It offers microsecond-granularity from a master server to other slave machines. In Fig. 15, we show the clock offset between a slave VM on the server *kane* and a master VM on the server *gerrard* in our testbed when both are running `ptpd` (version 2.3.1), an open-source implementation of PTP. The PTP application shares the network with two flows (one from a VM on server *rooney* and one from a VM on *kane*) that send Hadoop-like traffic to the VM on *gerrard* that runs `ptpd`: the traffic flows are competing for bandwidth with the PTP flows. The traffic is generated using *MoonGen* [15] on both VMs and emulates Hadoop traffic by sending bursts of line rate traffic at an average rate of around 480 Mbps. We observe in Fig. 15 that this cross-traffic causes `ptpd` to fall out of synchronization in the order of hundreds of microseconds, while the clock offset of `ptpd` on the same idle network remains in the order of tens of microseconds. When we introduce *Chameleon* to reserve network resources and route flows on appropriate queues through VLAN-based source routing, the interference is resolved and the `ptpd` synchronization offset remains as in the idle network scenario.

6 RELATED WORK

An overview of the most important existing works and their respective features in shown in Tab. 5 in the appendix.

Industrial applications have for a long time been a major use case for predictable latency. Proprietary solutions (e.g., Profibus or CAN) and Ethernet extensions have been developed for real-time industrial communications [13, 20, 56]. However, these solutions are either too expensive or demand changes within the protocol stack of forwarding devices.

In cloud networks, many efforts have been trying to provide bandwidth guarantees, work conservation, inter-tenant fairness and isolation, or a combination of these [8–10, 24, 28, 32, 39, 43, 44, 49, 50, 53, 58, 66]. While these approaches provide the scalability and quality of service (QoS) level needed for bandwidth-hungry data center applications, they do not provide strict buffer management as necessary for providing strict latency guarantees. Another category of works try to adapt the layer-4 (L4) protocol used in order to reduce and/or minimize (tail) latency and/or flow completion time (FCT) [4–7, 26, 27, 36, 41, 47, 59, 64, 69, 70]. However, these solutions do not provide packet latency guarantees.

A few recent efforts attempt to provide predictable latency and delay guarantees in shared network environments [48, 57, 63, 68]. These are centralized approaches relying on time-division multiple access (TDMA). However, such approaches either do not scale or rely on critical synchronization, hence, are too expensive and demand adaptations of the network infrastructure. Another approach falls back to physical isolation [68], which might, however, drastically waste physical resources. Avoiding end hosts synchronization, QJump [22] computes latency guarantees by ensuring that each flow has at most one packet in transit in the network at any given time. Unfortunately, this prevents applications from sending bursts of data. We saw in §5.1 that this leads to a high rejection rate and low network utilization. Silo [31], the closest related work, applies DNC to compute guarantees. Compared to Silo, *Chameleon* introduces priority queuing and exploits path diversity, also by reconfiguring flows at runtime; we have shown that this greatly increases the number of flows that can be accepted, i.e., network utilization. Furthermore, while Silo focuses on multi-rooted tree topologies, we introduce routing to accommodate (and leverage) any topological structure.

A few recent works focus on providing predictability and isolation at the end-host by efficiently sharing NIC resources among VMs [21, 37] but do not consider delays in the network fabric. PicNIC [37] employs congestion control mechanisms and hence does

not fit for traffic with strict latency requirements. FairNIC [21] provides predictability and isolation on a SmartNIC through principles very similar to those we used for our DPDK application (§4.2.1). Building on top of FairNIC to implement *Chameleon* on a SmartNIC is an interesting research direction.

7 CONCLUSION

This paper has shown that demand-aware and adaptive networks, leveraging source-routing and queuing flexibilities, introduce an opportunity to improve cloud network utilization while providing a predictable performance, in particular, latency. Our approach builds upon network calculus concepts while accounting for such flexibilities.

We understand our work as a first step and believe that our approach introduces several interesting avenues for future research. In particular, investigating the usage of a SmartNIC or a P4 NIC to tag packets deterministically in the data plane and reduce the footprint on host resources is an interesting research direction. More generally, while we have focused on datacenters, it will be interesting to explore opportunities of self-adapting networks, based on priority reconfigurations, in wide-area networks as well. We believe that the self-adapting approaches considered in this paper can also serve as a stepping stone toward self-driving networks [16] envisioned by the networking community.

REFERENCES

[1] [n. d.]. EU-project VirtuWind, Deliverable D3.2, Detailed Intra-Domain SDN & NFV Architecture. http://www.virtuwind.eu/. ([n. d.]). Accessed: 2020-01-30.

[2] [n. d.]. Use Cases IEC/IEEE 60802 v1.3. http://www.ieee802.org/1/files/public/docs2018/60802-industrial-use-cases-0918-v13.pdf. ([n. d.]). Accessed: 2020-01-30.

[3] Ariel Adam, Amnon Ilan, and Thomas Nadeau. [n. d.]. Introduction to virtio-networking and vhost-net (Red Hat Blog). https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net. ([n. d.]). Accessed: 2020-02-02.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data center tcp (dctcp). *ACM SIGCOMM computer communication review* 41, 4 (2011), 63–74.

[5] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 19–19.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 435–446.

[7] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-Service Multi-Queue Data Centers.. In *NSDI*. 537–549.

[8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *ACM SIGCOMM computer communication review*, Vol. 41. ACM, 242–253.

[9] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. 2013. Chatty Tenants and the Cloud Network Sharing Problem.. In *Nsdi*, Vol. 13. 171–184.

[10] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands.. In *NSDI*. 407–424.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.

[13] J-D Decotignie. 2005. Ethernet-based real-time and industrial communications. *Proc. IEEE* 93, 6 (2005), 1102–1117.

[14] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[15] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*. 275–287.

[16] Nick Feamster and Jennifer Rexford. 2017. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583* (2017).

[17] Daniel Firestone. 2017. {VFP}: A Virtual Switch Platform for Host {SDN} in the Public Cloud. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 315–328.

[18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.

[19] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2018. Survey of consistent software-defined network updates. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1435–1461.

[20] Piotr Gaj, Jurgen Jasperneite, and Max Felser. 2013. Computer communication within industrial distributed environment - A survey. In *IEEE Transactions on Industrial Informatics*, Vol. 9. IEEE, 182–189.

[21] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. 2020. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 681–693.

[22] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. 2015. Queues don't matter when you can jump them!. In *NSDI*. 1–14.

[23] Jochen W Guck, Amaury Van Bemten, Martin Reisslein, and Wolfgang Kellerer. 2017. Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation. *IEEE Communications Surveys & Tutorials* 20, 1 (2017), 388–415.

[24] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*. ACM, 15.

[25] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155* (2015).

[26] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 29–42.

[27] Chi-Yao Hong, Matthew Caesar, and P Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 127–138.

[28] Shuihai Hu, Wei Bai, Kai Chen, Chen Tian, Ying Zhang, and Haitao Wu. 2016. Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 1–9.

[29] Takeru Inoue. 2018. Reliability analysis for disjoint paths. *IEEE Transactions on Reliability* 68, 3 (2018), 985–998.

[30] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 219–230.

[31] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 435–448.

[32] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical network performance isolation at the edge. *REM* 1005, A1 (2013), A2.

[33] Alpar Jüttner, Balazs Szviatovski, Ildikó Mécs, and Zsolt Rajkó. 2001. Lagrange relaxation based method for the QoS routing problem. In *Proc. IEEE INFOCOM*, Vol. 2. 859–868.

[34] Sotirios Katsikeas, Konstantinos Fysarakis, Andreas Miaoudakis, Amaury Van Bemten, Ioannis Askoxylakis, Ioannis Papaefstathiou, and Anargyros Plemenos. 2017. Lightweight & secure industrial IoT communications via the MQ telemetry transport protocol. In *2017 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1193–1200.

[35] Fernando A. Kuipers. 2012. An overview of algorithms for network survivability. *International Scholarly Research Notices* 2012 (2012).

[36] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies,*

architectures, and protocols for computer communication. 514–528.
[37] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, et al. 2019. PicNIC: predictable virtualized NIC. In *Proceedings of the ACM Special Interest Group on Data Communication*. 351–366.
[38] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. 2014. *What you need to know about SDN control and data planes*. Technical Report.
[39] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. 2012. Netshare and stochastic netshare: predictable bandwidth allocation for data centers. *ACM SIGCOMM Computer Communication Review* 42, 3 (2012), 5–11.
[40] Jean-Yves Le Boudec and Patrick Thiran. 2012. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer.
[41] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.
[42] Endace Technology Limited. 2016. Endace DAG 7.5G4 Datasheet". https://www.endace.com/dag-7.5g4-datasheet.pdf. (2016). Accessed: 2018-10-26.
[43] Fangming Liu, Jian Guo, Xiaomeng Huang, and John CS Lui. 2016. eBA: Efficient bandwidth guarantee under traffic variability in datacenters. *IEEE/ACM Transactions on Networking* 25, 1 (2016), 506–519.
[44] Zhuotao Liu, Kai Chen, Haitao Wu, Shuihai Hu, Yih-Chun Hut, Yi Wang, and Gong Zhang. 2018. Enabling Work-Conserving Bandwidth Guarantees for Multi-Tenant Datacenters via Dynamic Tenant-Queue Binding. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 1–9.
[45] William M Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C Snoeren, and George Porter. 2019. Expanding across time to deliver bandwidth efficiency and low latency. *arXiv preprint arXiv:1903.12307* (2019).
[46] Jeffrey C Mogul and Lucian Popa. 2012. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review* 42, 5 (2012), 44–48.
[47] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 221–235.
[48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. 307–318.
[49] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: sharing the network in cloud computing. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 187–198.
[50] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. 2013. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 351–362.
[51] Diana Andreea Popescu. 2019. *Latency-driven performance in data centres*. Ph.D. Dissertation. University of Cambridge.
[52] Neil Robertson and Paul D Seymour. 1995. Graph minors. XIII. The disjoint paths problem. *Journal of combinatorial theory, Series B* 63, 1 (1995), 65–110.
[53] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival O Guedes. 2011. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. *WIOV* 1, 3 (2011), 784–789.
[54] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 404–417.
[55] Ermin Sakic and Wolfgang Kellerer. 2018. Impact of adaptive consistency on distributed sdn applications: An empirical study. *IEEE Journal on Selected Areas in Communications* 36, 12 (2018), 2702–2715.
[56] Thilo Sauter. 2010. The three generations of field-level networks - evolution and compatibility issues. In *IEEE Transactions on Industrial Electronics*, Vol. 57. IEEE, 3585–3595.
[57] Eike Schweissguth, Peter Danielis, Christoph Niemann, and Dirk Timmermann. 2016. Application-aware industrial ethernet based on an SDN-supported TDMA approach. In *World Conference on Factory Communication Systems (WFCS)*. IEEE, 1–8.
[58] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network.. In *NSDI*, Vol. 11. 23–23.
[59] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. 2012. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 115–126.
[60] Amaury Van Bemten, Nemanja Đerić, Amir Varasteh, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. 2019. Empirical predictability study of SDN switches. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 1–13.
[61] Amaury Van Bemten, Jochen W Guck, Carmen Mas Machuca, and Wolfgang Kellerer. 2018. Routing metrics depending on previous edges: The Mn taxonomy and its corresponding solutions. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.
[62] Amaury Van Bemten and Wolfgang Kellerer. 2016. Network Calculus: A Comprehensive Guide. *Technical University of Munich, Chair of Communication Networks, Technical Report No. 201603* (October 2016).
[63] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C Snoeren. 2012. Practical TDMA for datacenter ethernet. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 225–238.
[64] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 50–61.
[65] Jackson Woodruff, Andrew W Moore, and Noa Zilberman. 2019. Measuring Burstiness in Data Center Applications. (2019).
[66] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. 2012. The only constant is change: incorporating time-varying network reservations in data centers. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 199–210.
[67] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *Management Science* 17, 11 (1971), 712–716.
[68] Eitan Zahavi, Alexander Shpiner, Ori Rottenstreich, Avinoam Kolodny, and Isaac Keslassy. 2019. Links as a Service (LaaS): Guaranteed tenant isolation in the shared cloud. *IEEE Journal on Selected Areas in Communications* 37, 5 (2019), 1072–1084.
[69] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 139–150.
[70] Junxue Zhang, Wei Bai, and Kai Chen. 2019. Enabling ECN for datacenter networks with RTT variations. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 233–245.

Name	Guarantees				Constraints		
	Pkt. lat.	BW	Burst	WC	Switches req.	OS/App. changes	Other
SoA focusing on providing bandwidth (BW) guarantees and/or being work-conserving (WC).							
SecondNet [24]	✗	✓	✗	✗	PQ, MPLS	-	-
Oktopus [8]	✗	✓	✗	✗	PQ	-	-
Seawall [58]	✗	✗	✗	✓	-	-	-
Gatekeeper [53]	✗	✓	✗	✓	-	-	Non-congested core
Proteus [66]	✗	✓	✗	●	-	-	-
NetShare [39]	✗	✗	✗	✓	WFQ	-	-
FairCloud PS-L/PS-N [49]	✗	✗	✗	✓	WFQ	-	-
FairCloud PS-P [49]	✗	✓	✗	✓	WFQ	-	Tree topology
EyeQ [32]	✗	✓	✗	✓	ECN	-	Non-congested core
Elasticswitch [50]	✗	✓	✗	✓	-	-	-
Hadrian [9]	✗	✓	✗	✓	Custom protocol	-	-
Trinity [28]	✗	✓	✗	✓	PQ, ECN	-	-
HUG [10]	✗	✓	✗	✓	-	-	-
eBA [43]	✗	✓	✗	✓	Custom protocol	-	-
QShare [44]	✗	✓	✗	✓	WFQ	-	-
SoA optimizing the transport protocol.							
DCTCP [4]	✗	✗	✗	✓	ECN	OS	-
D^3 [64]	✗	✗	✗	✓	Custom protocol	OS, App.	-
PDQ [27]	✗	✗	✗	✓	Custom protocol	OS, App.	-
D^2TCP [59]	✗	✗	✗	✓	ECN	OS, App.	-
HULL [5]	✗	✗	✗	✓	Custom feature	OS	-
DeTail [69]	✗	✗	✗	✓	Custom protocol	OS, App.	-
pFabric [6]	✗	✗	✗	✓	Custom protocol	OS, App.	-
NDP [26]	✗	✗	✗	✓	Custom protocol	OS	-
Homa [47]	✗	✗	✗	✓	PQ	OS	-
HPCC [41]	✗	✗	✗	✓	Custom protocol	OS	-
Swift [36]	✗	✗	✗	✓	-	OS	-
SoA providing per-packet latency guarantees.							
TDMA Ethernet [63]	✓	✓	✓	✗	-	-	Millisecond timescale
Fastpass [48]	✓	✓	✓	✗	-	OS	End-hosts synchronization
QJump [22]	✓	✓	✗	✗	-	-	-
Silo [31]	✓	✓	✓	✗	-	-	Multi-rooted tree topology
LaaS [68]	✓	✓	✓	✗	-	-	Tenants on diff. phys. links
Chameleon (this article)	✓	✓	✓	✗	PQ, VLAN/MPLS or similar	-	-

Table 5: Overview of the related work. While many approaches focus on providing BW guarantees or optimize the transport protocol to reduce FCT and/or tail latency, only a few approaches provide strict latency guarantees, the focus of this article. These approaches suffer from limitations such as the need for synchronization or the support of only particular topologies. Our contributions in this article provide strict latency guarantees with precise BW and burst allowance in any general network without any particular requirements. Furthermore, we show in § 5.1 that *Chameleon* reaches higher network utilization than existing approaches.
Note that *App.* change does not include the usage of another transport library but only the requirement for providing more information to this library (e.g., deadline).