Distributed Resource Management Across Process Boundaries

Lalith Suresh[1]
VMware Research Group

Peter Bodik
Microsoft Research

Ishai Menache
Microsoft Research

Marco Canini
KAUST

Florin Ciucu
University of Warwick

ABSTRACT

Multi-tenant distributed systems composed of small services, such as Service-oriented Architectures (SOAs) and Micro-services, raise new challenges in attaining high performance and efficient resource utilization. In these systems, a request execution spans tens to thousands of processes, and the execution paths and resource demands on different services are generally not known when a request first enters the system. In this paper, we highlight the fundamental challenges of regulating load and scheduling in SOAs while meeting *end-to-end* performance objectives on metrics of concern to both tenants and operators. We design Wisp, a framework for building SOAs that transparently adapts rate limiters and request schedulers system-wide according to operator policies to satisfy end-to-end goals while responding to changing system conditions. In evaluations against production as well as synthetic workloads, Wisp successfully enforces a range of end-to-end performance objectives, such as reducing average latencies, meeting deadlines, providing fairness and isolation, and avoiding system overload.

CCS CONCEPTS

• **Computer systems organization** → *Cloud Computing*; *n-tier architectures*;

KEYWORDS

Microservices, Service-Oriented Architectures, Resource Management, Rate Limiting, Scheduling.

ACM Reference Format:
Lalith Suresh[1], Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed Resource Management Across Process Boundaries. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3132020

[1]Part of the work was done while at Microsoft Research.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3132020

1 INTRODUCTION

Many organizations including Netflix, Amazon, Uber, SoundCloud, Google and Spotify have adopted Service-oriented Architectures (SOAs) [25] and Micro-services [52] to build large-scale Web applications [8, 47, 48, 53, 64, 70] and infrastructure systems [3, 71]. SOAs comprise fine-grained, loosely coupled *services* that communicate via lightweight API calls over the network. Every service comprises multiple *service instances* or *processes*, each running inside a physical or virtual machine. For instance, Netflix has separate services for managing movie and user data, authentication, and recommendations [49]. Typically, these divisions align with developer team structures [53]. These systems are commonly shared by multiple *tenants*, where tenants may represent different external customers or consumers, but also internal product groups, applications, or system background tasks.

SOAs have three characteristics that complicate managing their end-to-end latency and throughput. First, request execution in SOAs spans tens to hundreds of services, forming a DAG across the service topology [37]. The exact structure of the DAG is often unknown when the request first enters the system, since it depends on multiple factors like the APIs invoked at each encountered service, the supplied arguments, the content of caches, as well as the use of load balancing along the service graph. Second, by design, individual services in SOAs lack end-to-end visibility into the service topology and by extension, the request execution graph; in fact, services view each other as black boxes. Third, requests from different tenants contend for *shared* resources within individual processes of a service such as threadpools, locks, blocking queues, and connection objects. Isolation mechanisms at the host OS or hypervisor fail here, as they lack visibility into the existence of multiple tenants as seen by individual processes.

The above characteristics lead to fundamental resource management challenges. The lack of end-to-end visibility and complex request execution structures make it challenging to regulate two key metrics in an SOA across multiple tenants (§3.2): (i) the end-to-end throughput (and thereby, the load at every process) and (ii) the end-to-end latency. To regulate system load, we must correctly attribute overload to a specific subset of tenants, and rate limit the entire chain of API invocations for only those tenants, with minimal impact on others. Meeting end-to-end latency goals requires local request scheduling decisions at every hop despite limited visibility into the full request execution graph. Adaptively rate limiting and scheduling requests is necessary to meet several performance objectives that tenants and operators care about, such as avoiding overload, guaranteeing throughput, sustaining fairness, meeting deadlines, ensuring priorities, and achieving low latencies.

Unfortunately, existing libraries [50, 69] for building SOAs are ill-equipped to deliver on the above performance objectives. These libraries require extensive tuning of static thresholds for rate limiters, circuit breakers [17], and timeouts to regulate both load and latency. Setting these thresholds manually in a complex distributed system is fragile and becomes out of date quickly as systems evolve and workloads change [39, 51]. Furthermore, several case studies highlight how complex interactions in SOAs not only lead to sharp degradation in performance (e.g., lower throughput and higher latencies), but also trigger cascading behaviors that result in wide-spread application outages [7, 32, 34, 68]. These challenges necessitate adaptive, end-to-end resource management for SOAs.

In this paper, we highlight the unique challenges involved in meeting the above performance objectives in multi-tenant SOAs, which are fundamentally different than typical network scenarios. Our key contribution is the design of novel adaptive techniques for SOAs that leverage existing building blocks (rate limiters and request schedulers) to meet end-to-end performance goals, *despite* the lack of global visibility into request execution DAGs and their load at every service. These techniques are embodied in Wisp, a framework for managing resources in SOAs with minimal operator intervention.

Wisp's design hinges on the observation that rate limiting and scheduling mechanisms at each process, only informed by measurements of their local neighborhood, suffice to realize a broad set of performance policies in SOAs. Wisp uses rate-limiting and backpressure mechanisms that operate at the granularity of groups of requests which we term *workflows*. Wisp rate limits workflows such that they share resources at every process according to throughput-related policies, e.g., bottleneck fairness [45] or dominant resource fairness [26]. Wisp also operates at the level of individual requests and prioritizes their execution at each process according to latency-related policies, such as Earliest Deadline First (EDF) [65] or Least Slack Time First (LSTF) [35].

Enforcing the above policies, however, requires end-to-end knowledge of bottlenecks in the service topology, and characteristics of the request execution graphs. Wisp overcomes these obstacles through several mechanisms. Wisp uses causal propagation of workflow identifiers throughout the system to attribute resource utilization to individual workflows. It then uses a novel distributed rate adaptation mechanism §4.1, where upstream services throttle workflows according to bottlenecks that emerge on their execution graph. The aggressiveness of the throttling is determined through a configurable parameter that tunes a tradeoff between high utilization and the request drop probability (due to overload). Lastly, Wisp realizes end-to-end variants of policies such as EDF and LSTF by dynamically estimating end-to-end properties of each request (e.g., remaining processing time). Importantly, Wisp decouples policy from mechanism, and meets different performance objectives by only leveraging building blocks that are already present in typical SOAs (namely, rate limiters and schedulers). Our contributions are:

- We present characteristics of SOAs through a measurement study of large production systems (§2) and highlight fundamental challenges in meeting end-to-end performance objectives in multi-tenant SOA settings (§3).
- We design Wisp (§4), a framework to enforce a diverse range of resource management policies in SOAs by adaptively tuning rate limiters and schedulers based only on local measurements at each process. Importantly, Wisp achieves these goals through *fully distributed* mechanisms, without requiring prior knowledge of request execution graphs and resource demands.
- We evaluate a Wisp prototype (§6). Our results show that Wisp enforces a wide range of performance objectives such as avoiding cascading failures, meeting soft end-to-end deadlines (e.g., 10x improvement in the 99th percentile latency-deadline ratio), and isolating low-latency workflows from high-throughput workflows (2x improvement in average latencies).

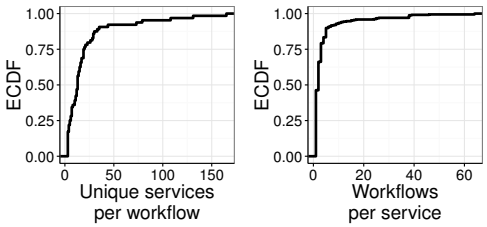

Figure 1: Workflows in production.

2 SOAS IN PRODUCTION

We now discuss relevant characteristics of SOAs using a combination of measurements from a large cloud provider and prior reports on systems from other environments [8, 37, 47, 48, 53, 64, 70].

2.1 Services, processes and workflows

A single application such as bing.com, amazon.com, or netflix.com is composed of multiple services. These services are typically maintained by separate teams (or even third parties) and communicate exclusively over well-specified APIs [52]. Each service runs multiple instances (OS processes), distributed across multiple servers or virtual machines. Requests are dispatched to instances based on the type of service; for example, requests can be load-balanced among processes of a stateless service, whereas routing in stateful services is typically based on some form of hashing. While a request typically enters the system through an *entry point* service such as a set of frontend web servers, requests may also originate from internal systems that access shared infrastructure services. A workflow represents application-specific "groups" of requests, that form an execution DAG across a set of services [37]. For instance, all requests from the same tenant may be classified as the same workflow.

2.2 Analysis of shared services

Bing. Figure 1 presents characteristics of the Bing SOA [37]. Here, each workflow is an execution DAG, and corresponds to different features of the larger offered application (including, but not limited to, web, video, and image search). These workflows contend for shared, in-memory resources such as threadpools at different services.

Figure 1 depicts the number of services involved in the execution of different workflows (left), and the number of unique workflows seen per service (right). We note that 50% of workflows execute across at least 13 services, and 5% of workflows even hit 78+ services. Similarly, while several services process only one workflow, we note that tens of core services are shared by multiple workflows.

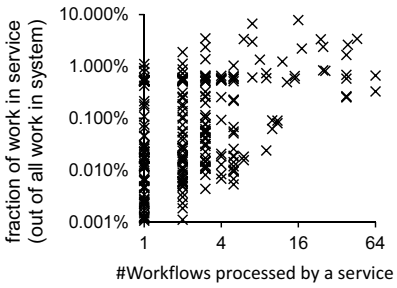

Figure 2: Fraction of overall request processing time spent per service in production. Each dot is a service, x-axis is the number of workflows in that service, y-axis is the fraction of overall processing time spent per-service.

For instance, 24 services are shared by at least 10 different workflows, with two services supporting 64 unique workflows. These workflows have further semantics that affect their performance needs (e.g., requests from real users require higher priority in the system than those from bots).

Importantly, *shared services* (services processing two or more workflows) are responsible for a large fraction of overall request processing time (or work) in the system. Misbehaving workflows in shared services can impact throughput and latency for other workflows (§2.4). We traced processing time statistics for each workflow across all services (Figure 2). We observe that 53.7% of services process at least two workflows. *Nevertheless, they account for a much larger fraction of the total work*: 86% of request processing time in the system is spent within shared services; 53% within services that handle 5+ workflows, and 31% within services handling 10+ workflows. Shared services are therefore critical for end-to-end performance.

Azure Storage. We also consider the Azure Storage platform [13]. It supports tens of external APIs, corresponding to reads, inserts, deletes and scans of both data and metadata. The system comprises services shared across workflows such as the front ends (FE), partition servers (PS), and extent nodes (ENs). The total CPU cycles for serving a request within a service can *vary by up to four orders of magnitude* across workflows [46]. Such variability calls for careful resource accounting and management, e.g., to avoid starvation of short requests. We discuss more characteristics of the system in the evaluation (§6).

2.3 Opaque request execution DAGs

A crucial aspect of request execution in all these SOAs is the *opacity* of the execution graph and its corresponding resource consumption at each service. That is, each service is typically oblivious to the *(i)* end-to-end execution graph of the request, which depends on load balancing, multiple levels of caching, number of instances per service, and API parameters used to invoke different services, *(ii) request amplification*, wherein a single request at an upstream service might correspond to thousands of requests at a downstream service, and *(iii) request cost*, where different requests at an upstream service may have varying costs further downstream; for instance, the cost of loading an object in Azure Storage is proportional to the object size and is potentially unknown when a request first enters the system at an entry point. Lastly, request execution characteristics may change as the codebase for individual services evolve, further aggravating the opacity of request execution graphs [39].

2.4 Shared services and outages

The fact that different workflows contend for common resources within shared services has led to outages among production systems. Visual Studio Online experienced an outage [34] caused by an interaction of two different workflows in a hierarchy of services. A single workflow was accessing a slow database deep in the service hierarchy. The blocking RPC calls from the upstream service eventually exhausted the service's thread pool. This subsequently starved *other unrelated* requests that were trying to connect to an authentication service, causing widespread application unavailability. A similar interaction across tiers led to an Amazon AWS outage [32]. Services therefore have to be aware of potential bottlenecks among their downstream services. In another episode, a slowdown of some Amazon EBS instances triggered a sequence of bottlenecks in related services. During the firefighting effort, operators *manually intervened* to throttle upstream EC2 service APIs to reduce load on the downstream EBS service, which affected more customers than necessary [7]. Such manual intervention is challenging and error prone.

3 OVERVIEW OF WISP

Wisp is a framework for building SOAs that enables end-to-end resource management for diverse request types. Wisp creates groups called *workflows*, defined as a set of requests that belong to the same class or tenant and are bound by the same resource management criteria. Tagging workflows provides fine-grained visibility into request execution, which facilitates attribution of resource utilization and processing activity to specific request groups. It allows processes to differentiate between requests that are causally connected to different workflows. Wisp enforces resource sharing policies through *workflow-level* mechanisms. In addition, Wisp uses *request-level* mechanisms to prioritize request execution according to latency-related policies. For instance, in the *Bing SOA* discussed in §2.2, each request type may be classified as a workflow, allocated a fair share of resources, and scheduled according to its urgency. On the other hand, all requests originating from bots can be treated as a low priority workflow, and be scheduled with no deadline targets.

In this section, we describe the performance goals of Wisp (§3.1), and highlight the challenges in achieving them in a distributed setting (§3.2). We then present in §3.3 the main building blocks of our solution. A detailed description of the design follows in §4.

3.1 Performance objectives

We note that there are two classes of performance objectives of interest in the SOA setting.

The first class pertains to regulating workflow-level end-to-end throughput. This includes achieving high utilization to avoid wasteful over provisioning [20], avoiding overload [32, 34], fairness and performance isolation among competing workflows [26, 45, 61], and provisioning shared services to offer a subset of users minimum throughput guarantees (with best effort for others) [21].

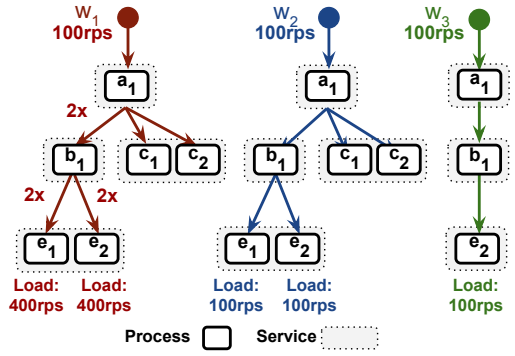

Figure 3: Example service topology, workflows in §4.1. Services are sets of processes. Workflows execute within processes, forming an execution DAG. We depict the relevant parts of the service topology separately for each workflow.

The second class of performance policies relate to managing request-level end-to-end latencies. This includes differentiated services or statically prioritizing some workflows over others (e.g., premium vs. free users, or interactive vs. background tasks) [20], or meeting end-to-end latency deadlines (user facing web-sites often need to load a page in under 100-400ms)[21, 63].

3.2 Challenges

The complex request execution DAGs and lack of global visibility in the SOA setting poses unique challenges in meeting the above performance goals.

Rate limiting must account for bottlenecks end-to-end. Consider the services from Figure 3 and the three workflows w_1, w_2 and w_3, all of which contend at processes a_1. Assume a purely local approach where all processes rate limit according to their local bottlenecks. In this case, if w_1 requests execute and timeout in overloaded e_2, the work they executed (and contention they introduced) in a_1, b_1, c_1 and c_2 is wasted. Instead, an approach that rate limits w_1 at a_1 reduces wasted resources *and* increases the throughput of w_2 and w_3 since more resources in a_1 and e_2 would be available. However, a_1 is not aware of *(i)* the downstream bottleneck e_2, *(ii)* which workflows use e_2, and *(iii)* the load imposed by each workflow on e_2. This makes it challenging to determine rate limits for each workflow at a_1. Furthermore, as workflows contend for shared resources, their rate limits at every hop have *mutual dependencies* and therefore cannot be tuned independently of each other.

Meeting latency deadlines requires dynamic request-specific information. Scheduling requests at every process to meet *end-to-end* latency guarantees is challenging due to the complex structure of execution DAGs and the inherently stochastic nature of the problem (e.g., queuing effects at each process). Specifically, achieving latency goals depends on the processing times for each workflow at every service of their execution DAGs [37]. In Figure 3, if w_3 has a 300ms end-to-end deadline and requires 250ms of processing time at e_2, it only has a budget of 50ms to complete at a_1 and b_1.

Despite myriad existing scheduling algorithms to prioritize requests with different performance objectives (e.g., shortest remaining time first (SRTF) [18] and least slack time first (LSTF) [35]), *realizing* these policies in a fully distributed setting remains non-trivial. These algorithms rely on information such as the remaining processing time and slack to deadlines, which need to be dynamically estimated across diverse workflows.

Table 1: Notation used for algorithm description

w	Workflow w
s	Service s, defined as a set of processes
p	Process p
$\alpha^w_{p,d}$	Amplification factor for w from p to process d
σ^w_p	Admission rate of w at p

3.3 Design principles

Given the scale and heterogeneity of the applications we aim to support, the design of Wisp is driven by three core requirements: *(i)* avoid centralized coordination, *(ii)* exchange minimal information between services, *(iii)* operate without prior knowledge of a workflow's costs and graph structure. At the same time, Wisp must provide building blocks for operators to enforce flexible system-wide policies depending on their requirements with minimal tuning. These considerations led us to a design with the following key functionalities:

Workflow-level distributed rate adaptation. Processes use a local policy to identify admission rates for requests of different workflows and share them with their upstream neighbors. Next, Wisp uses a novel distributed rate adaptation protocol to bubble these admission rates through the service chain, calibrating rate limiters at upstream services to account for bottlenecks downstream. This ensures that workflows are rate limited as *early as possible* instead of only being throttled at the point of congestion (§3.2). Furthermore, we perform admission control to avoid wasting resources on requests that will not complete within their deadlines.

Request-level scheduling. Wisp leverages request schedulers at every process to mediate access to local resources. Schedulers may enforce fair queuing across requests from different workflows to guarantee performance isolation, or use policies such as shortest job first (SJF), earliest deadline first (EDF), and least slack time first (LSTF) to optimize for a range of end-to-end performance goals. Wisp dynamically estimates end-to-end properties of requests such as their remaining processing time to execute algorithms such as LSTF.

4 DESIGN

We now discuss our solutions for workflow-level distributed rate adaptation and request-level scheduling.

4.1 Workflow-level rate adaptation

Following from §3.2, we design an algorithm which seeks to balance the overall system utilization and the request drop probability (due to overload), by setting appropriate rate limits at all processes in a fully distributed manner. We outline our distributed algorithm for adapting per-workflow rate limits system-wide in §4.1.1, provide an

Algorithm 1 Rate adaptation at $p \in w$ (every 100ms)

Constants: q: quantile parameter

1: $\sigma_p^w \leftarrow LocalResourceSharingPolicy()$
2: **for all** D | downstream services **do**
3: **for all** d | processes in service D **do** ▷ In parallel
4: $\sigma_d^w \leftarrow GetRates(d)/\alpha_{p,d}^w$
5: $\sigma_p^w \leftarrow min(\sigma_p^w, Q(\sigma_d^w(d \in D), q))$

illustrative example in §4.1.2, and further comment on its tuning in §4.1.3.

4.1.1 Distributed rate adaptation algorithm

The goal of each process p is to set a rate limit σ_p^w for each workflow w, to enforce a specific resource sharing policy across workflows, while balancing the system utilization and the request drop probability. Concretely, every process p, for a workflow w, computes a rate limit σ_p^w as the minimum between the local rate limit and the rate limits of downstream services of p. p periodically executes **Algorithm 1** (see notation in Table 1).

Monitoring workflow characteristics. To determine their local rate limits as well as those of their downstream services, processes require information about each workload w. Specifically, each process p monitors the average load on local resources by each workflow. It also maintains $\alpha_{p,d}^w$, the *amplification factor* of w from process p to a downstream process d. For instance, in Figure 3, if the measured arrival rates at a_1 and b_1 for w_1 are 100 and 200, respectively, then $\alpha_{a1,b1}^{w1} = 2$; in other words, a single request of w_1 in a_1 triggers two requests to b_1 on average.

Adapting local rate limits (Line 1). Our algorithm starts with p computing its local rate limits through an operator-specified policy (§4.3). Policies observe the load on local resources by different workflows to infer their *costs* (§5). They then compute *local rate limits* such that the resulting share of each resource by a workflow corresponds to a policy (say, bottleneck fairness). As the resource share or cost for w changes, the local rate limits adapt accordingly. In Line 1, p initializes σ_p^w to the local rate limits by executing the policy.

Trading off utilization for dropped requests (Lines 2-5). Next, for each downstream service D, p queries the rate limits of the processes $d \in D$, scaled by the amplification factor $\alpha_{p,d}^w$ (Lines 2-4). Intuitively, if *every* process p sets its σ_p^w according to the *minimum* of its local rate limit and that of its downstream processes, we avoid overload along the service topology. However, this approach is conservative and may significantly reduce resource utilization. For example, if a process communicates with 100 downstream processes, a slowdown in one of those processes directly reduces the admission rate which then propagates upstream.

Instead of using the minimum, we propose using a quantile function Q, which depends on a quantile *knob* $q \in [0, 1]$ (Line 5). For example, $q = 0.5$ uses the median downstream rate and makes Q robust to outliers; however, the overloaded downstream services drop requests that are in excess of their announced rate limits. Navigating this trade-off allows us to increase resource utilization. We discuss the tuning of the knob q in §4.1.3.

Computing σ_p^w (Line 5). Finally, p adjusts its announced rate σ_p^w as the minimum between the current rate and the value of Q, which can be regarded as the *per-service* aggregate rate (of D). p distributes its announced σ_p^w to its upstream processes in proportion to their demands, when the upstream processes invoke $GetRates(\cdot)$ (Line 4).

In summary, every iteration of **Algorithm 1** *bubbles up* admission rates through the service topology, with processes of upstream services enforcing rate limits that account for downstream service rates according to the quantile knob q.

4.1.2 Rate adaptation trade-off by example

We now present a simple example to describe the behavior of Algorithm 1. Consider the setting in Figure 3. The three workflows w_1, w_2, and w_3 have an arrival rate (demand) of 100rps each at a_1, and a subsequent load at (e_1, e_2) of $(400, 400)rps$, $(100, 100)rps$, and $(0, 100)rps$ respectively. Assume that all processes have a capacity of 500rps each.

This implies that the total load at e_2 from all three workflows exceeds its capacity of 500rps. Assume running a max-min fairness policy at e_2, which, observing the load per workflow, asserts that w_1 is exceeding its fair share of 300rps and needs to be rate limited.

With $q = 0$, $\sigma_{a_1}^{w_1}$ is computed to be 75rps, which guarantees that e_1 and e_2 receive 300rps of load from w_1, but thereby leaves 100rps of spare capacity at e_1. With $q = 1$, $\sigma_{a_1}^{w_1}$ in turn becomes 100rps. This maximizes utilization at e_1 (400rps), but e_2 now drops 100rps (incident load of 400rps, and rate limit of 300rps). This is a fundamental trade-off in the workflow rate limiting problem: calibrating end-to-end rate limits to match the slowest process of bottlenecked services risks under-utilization ($q = 0$), whereas matching the fastest process risks wasting resources and dropping requests ($q = 1$).

4.1.3 Guidelines for setting the quantile knob

As the advertised rates in Algorithm 1 are non-decreasing in the quantile knob q, both the request drop probability and system utilization increase with q. Operators can leverage this property and set q to properly balance the two. Suppose that the operator wishes to minimize a weighted sum between the average request drop rate and the average unused capacity. Let us focus first on a system with a single service and a large number of processes. Assume that the advertised rates σ_d^w from Line 4 in Algorithm 1 are represented by a random variable X with support $[0, M]$ and density $f(x)$, and also $\alpha_{p,d}^w = 1$. The goal of the quantile knob q can be expressed in terms of minimizing a weighted sum of residual values

$$\begin{aligned} h(a) &:= E\left[(a - X)1_{a \geq X}\right] + \beta E\left[(X - a)1_{a \leq X}\right] \\ &= \int_0^a (a - x)f(x)dx + \beta \int_a^M (x - a)f(x)dx \,. \end{aligned}$$

The expectations correspond to the average drop rate and average unused capacity under some arrival rate (demand) a; $1_{\{\cdot\}}$ denotes the indicator function and β is the weighting factor. Differentiating $h(a)$,

$$h'(a) = \int_0^a f(x)dx - \beta \int_a^M f(x)dx$$

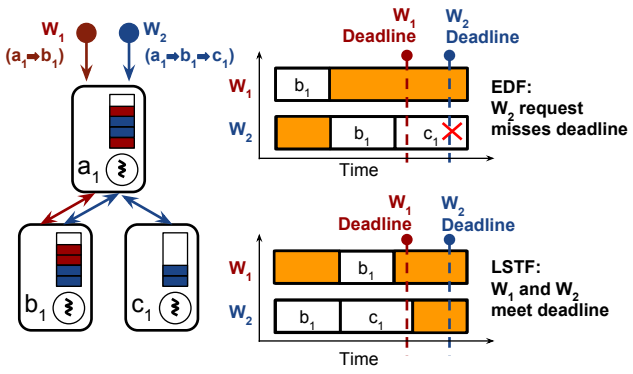

Figure 4: Scheduling example: requests from w_1 and w_2 contend at a_1 and b_1. (Right) requests from w_1 have a shorter deadline than w_2, but w_2 requires additional processing time (at c_1). (Top schedule) EDF prioritizes w_1 at b_1 because of the closer deadline; w_2 misses its deadline. (Bottom schedule) LSTF considers the slack for w_2 with progress metrics, and meets both deadlines.

and setting $h'(a) = 0$, we obtain that the optimal value of a, which would be returned by the quantile function $Q(\cdot)$ in Line 5, must satisfy $\frac{P(X \leq a)}{P(X \geq a)} = \beta$. In the particular case when $\beta = 1$ then a is the median of X and consequently $q = 0.5$; as another example, if $\beta = 3$ then a would be the third quartile and $q = 0.75$.

Now consider a more general system with a DAG of n services. Suppose the admission rates in each (process, service) pair are identically distributed. For $\beta = 1$, it can be shown that setting the quantile to $q = 0.5$ and using our algorithm for rate adaptation would lead to an optimal solution (the only difference in the analysis is that the differential of the weighted sum of residual values would be $nh'(a)$); different values of β would lead to different optimal values of q.

Naturally, it is hard to a-priori determine the optimal value of q for a general system as it depends on the distributions of the admission rates of the processes for each service, and also on the distributions of the amplification factors. In fact, the operator can theoretically benefit by setting different quantile values for each (service, workflow) pair, e.g., via line-search procedure at a slow time-scale. Nonetheless, guided by the above analysis and for simplicity, we use $q = 0.5$ by default in our experiments (§6 presents a sensitivity analysis of q).

4.2 Request-level scheduling

While regulating the system load already improves latency, we also schedule requests at processes to further meet different performance goals. In particular, we combat stochastic effects that inflate end-to-end latencies such as bursty arrivals, and queuing at every hop.

Schedulers in Wisp enforce policies such as performance isolation between requests of different workflows (e.g., protect low latency workflows from head-of-line blocking due to throughput heavy workloads [20]) or prioritize their execution based on end-to-end performance objectives (e.g., meeting deadlines).

Need for estimating progress. Consider the goal of meeting end-to-end deadlines. A natural scheduling algorithm to execute at every hop is EDF, which prioritizes requests with closer deadlines. We discuss an execution of EDF in the context of Figure 4. Requests from w_1 execute serially at a_1 and then b_1. Requests from w_2 contend with those from w_1 at a_1 and b_1, but additionally also execute at c_1. At b_1, requests from w_1 have a closer deadline than those from w_2. EDF in this scenario causes b_1 to prioritize w_1 over w_2, eventually causing w_2 to miss its deadline (Figure 4, top schedule).

Algorithms such LSTF remedy this by prioritizing requests according to their *remaining processing* time and their deadline (Figure 4, bottom schedule). However, Wisp by design operates without prior knowledge of the costs and DAG structure of workflows. Therefore, to benefit from scheduling algorithms such as LSTF, Wisp needs to estimate metrics such as the total and the remaining processing time for each request *as they execute*. We achieve this through *progress metrics*. Note, the processing time for a request differs from its end-to-end latency (which includes waiting times). This is important, because if a request needs only 1ms of processing at a process, but the same workflow experienced 100ms of queuing delay in the past, an "expected processing time" of 100ms mischaracterizes the request's priority for algorithms such as LSTF.

Table 2: Propagated metadata and components that use them (*progress metrics (§4.2)).

Metadata	Used by
Workflow ID	Rate limiters, fair queuing, resource accounting
Elapsed service time *	LSTF, SRTF
Total service time *	LSTF
Work so far *	LASF
Total work *	SJF
Deadline	EDF, LSTF, drop logic

Progress metrics. We refer to metrics that reflect a request's true execution progress as *progress metrics*. Progress metrics can be queried for the total end-to-end estimate, elapsed, and remaining values at any point in the request execution. Example metrics we track are the processing time and total work (demand divided by capacity), which enable multiple scheduling algorithms (Table 2). Every request therefore is "tagged" with the necessary progress metrics, which is updated by processes as the request executes.

Our solution to track a request's progress emulates the standard recursive algorithm to compute the sum of all vertices in a DAG (every process adds its local sum to the sum reported by its child sub-trees). For metrics such as the remaining processing time, processes scale the computed sums from their children by the degree of parallelism. An alternative approach is to consider the maximum reported sub-tree sum, which however tends to over-estimate the expected processing time at other sub-trees. For metrics such as the total expended work, a sum gives the desired result. As each request executes, processes have an estimate of the elapsed value of the metric m so far ($m^{elapsed}$). When a request's execution completes end-to-end at the originating process, the resulting total for every metric (m^{total}) is maintained at the entry points as an exponentially weighted moving average (EWMA). For future invocations of the workflow, m^{total} is propagated with the request. Scheduling

algorithms that use the remaining value of a metric (e.g., SRTF and LSTF) estimate it as $m^{total} - m^{elapsed}$ at any instant.

4.3 Example operator policies

Policies to compute rate limits. Processes may choose to provide static throughput guarantees, calculate rates based on bottleneck fairness, or receive feedback from the local queue schedulers and resources. As long as processes expose their per-workflow rate limits to their upstream neighbors, the rate aggregation mechanism *transparently* ensures that upstream processes converge to rate limits that factor in downstream restrictions (§4.1.1). We implemented a bottleneck fairness policy similar to [45]. With this policy, each process p checks if a local resource is overloaded. If not, it ramps up the announced rate limits for every workflow for which p is a leaf (no further downstream services), by an additive probe factor β, scaled according to the amount of spare capacity available (this increases the rate faster when there is spare capacity available and is conservative otherwise). If instead a local resource is bottlenecked, the system calculates max-min fair shares for the contending workflows.

Local scheduling policies. We now discuss multiple scheduling policies realized using our framework. We implemented a multi-resource fair queuing scheme similar to [62]. Fair queuing across workflows protects short and bursty workflows that do not benefit from rate limiting (§6). The scheduler uses the deficit round-robin algorithm [61], wherein every workflow gets a number of credits per-round and credits are consumed based on the expected cost of the requests. A fixed number of credits per-round are budgeted across each workflow in proportion to the shares per-workflow (computed via a bottleneck fairness allocation or via DRF [26]). To meet end-to-end deadlines, our LSTF policy favors requests with the least remaining slack (§6). All scheduling policies are enabled by progress metrics and other metadata propagated via the requests.

Admission control and drop policies. To regulate queue lengths system-wide (rate limiters, scheduler, and resource queues), requests need to be dropped according to different policies. For instance, a drop policy we use ensures that when a request from a workflow w arrives at a rate-limiter in p (shaping at rate σ_p^w), the rate-limiter only queues a request such that it is feasible to meet the deadline. The policy computes the maximum tolerable queuing delay for a request from the request's deadline, the average observed end-to-end latency for w from p onwards, and the elapsed time so far. If the expected queuing delay on the rate-limiter (inferred from the backlog) exceeds the calculated tolerance, the request is dropped. Wisp thus drops requests that have little chance of completing within their deadlines, freeing up resources for other requests. Note, typically, SOAs gracefully degrade service when sub-systems cannot service a request [12].

5 WISP IMPLEMENTATION

We find that the basic building blocks to introduce Wisp are available in most SOA frameworks [50, 69, 72], making it feasible to realize these ideas today. Our prototype is implemented as a C# library.

Execution model. We align our design with execution models of SOA frameworks today. Wisp comprises three components (Figure 5): *(i)* the user code which contains the business logic for the application, *(ii)* a core that monitors workflow and resource characteristics, and transparently executes the distributed rate adaptation algorithm and metadata propagation for scheduling, and *(iii)* the operator specified policies that define how to compute local rate limits and scheduling decisions. Developers building micro-services express business logic as compositions of *sub-tasks* triggered by the arrival of requests (sub-tasks are equivalent to Hystrix *commands* [50] and Xenon tasks [72]). Rate limiting decisions are made against requests, whereas scheduling decisions are made against sub-tasks.

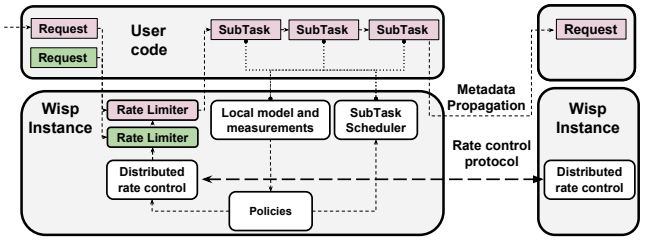

Figure 5: Wisp architecture. Policies examine resource utilization by different workflows locally and determine rate limiting and sub-task scheduler behavior. Distributed rate control automatically tunes upstream rate limits to reflect downstream bottlenecks. Metadata propagation enables end-to-end scheduling policies.

The core bridges user code and operator policies. As sub-tasks execute, they utilize resources such as connection pools, locks, and threadpools. Each request in Wisp has a context object propagated with it, which holds necessary metadata required for the operator policies such as the workflow ID, deadline and metrics that estimate request progress (§4.2). Furthermore, Wisp monitors utilization of the local resources and infers properties of the workflows. The metadata propagation and local model inform resource management decisions by the operator policies (§4.3). The distributed rate adaptation then automatically translates the constraints exposed by the rate limiting policies at each process into upstream rate limits, while factoring in workflow characteristics (Algorithm 1). The sub-task scheduler is invoked between each execution of a sub-task, wherein it prioritizes sub-tasks based on the scheduling policies specified by the operator.

Estimating workflow and resource characteristics. The algorithms in §4 assume the availability of some measurements at every process. This includes the arrival rates of requests per-workflow, the number of further calls per-request to downstream services (amplification factors, α), the load per workflow per resource, and the average completion time of a workflow once admitted. We track EWMAs of these measurements over a control interval. Furthermore, resources managed by Wisp track the load by workflow (similar abstraction as in Retro [45]). For instance, for threadpool resources, the average service time of each workflow's task execution gives us the load. Other in-process resources such as connection pools and objects are wrapped by semaphores as in Hystrix [50] to limit concurrency, and the duration for which a workflow holds a semaphore is used to compute the load.

Metadata propagation. Mechanisms to propagate metadata across process boundaries are a standard feature in several SOA frameworks.

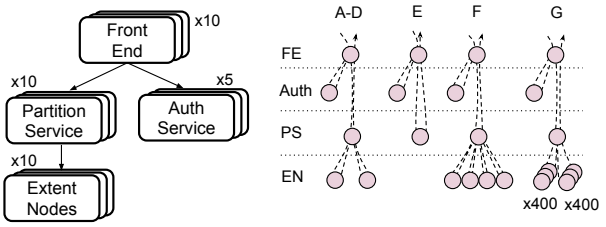

Figure 6: *(Left)* Azure Storage topology used in the evaluation. Workflows traverse four services (FE, Auth, PS and EN). Requests to a PS trigger multiple read/write sub-tasks to the ENs and local computations. *(Right)* Workflow DAGs in performance isolation experiment.

For instance, Finagle [69] has *Broadcast Request Contexts*, and Xenon [72] has *Operation Contexts*. We use similar infrastructure to not only propagate metadata in each request such as the workflow ID and end-to-end deadlines, but also to dynamically estimate request progress.

Sharing rate limits with upstream processes. In Wisp, upstream processes observe the rate limits per-workflow of their downstream neighbors in order to calibrate their own rate limits (Algorithm 1). In our implementation, upstream processes directly probe their downstream processes for their rate limits. Alternative approaches include leveraging available publish-subscribe infrastructure [56, 60], or using in-band mechanisms such as packing rate limits per-workflow in message response headers. We leave a detailed exploration of such implementation trade-offs to future work.

6 SYSTEM EVALUATION

We now demonstrate how Wisp enforces different resource management policies: *(i)* Avoid overload and provide isolation in the presence of aggressive workflows, *(ii)* Meet end-to-end deadlines, and *(iii)* Isolate low-latency traffic from high throughput traffic. We show *(iv)* how distributed rate adaptation reacts to hotspots, and *(v)* how to navigate the goodput vs utilization trade-off using the quantile knob q.

Experimental Setup. We run our experiments on a testbed comprising forty virtual machines. Each VM has a single 2.40 GHz CPU core, 2GB of RAM and runs Windows Server 2012 R2. All services make use of the .NET CLR version 4.5. Each instance of a service in our experiments runs as a process inside a VM.

We setup a topology of services and processes according to that of Azure Storage, discussed in §2.2; this system exhibits complex DAGs of operations, as shown in Fig. 6. We reproduce request routing, execution DAGs, and sub-task cost characteristics of Azure Storage. Our setup comprises four tiers: front-end (FEs), authentication (AUTH), partition service (PS), and extent nodes (ENs). FEs are the entry points that accept client requests. FEs first verify client requests against an AUTH server. They then route requests to a PS that holds the table for a tenant, determined via consistent hashing. The PS process then issues multiple reads and writes to the EN service before executing compute work locally and returning results.

Our setup comprises ten FEs, five AUTH servers, ten PS instances and ten EN servers. Wisp monitors resource utilization by workflow across all thread pools and connection pools in the system. For a processing stage each within the EN and PS, we vary the service times for different workflows to study different bottleneck scenarios (service times are drawn from exponential distributions). We drive client workloads from five VMs. Every workflow has a fixed number of PS partitions. Each PS partition corresponds to a fixed number of blocks on the EN tier, uniformly distributed across all EN processes. Clients generate requests according to a Poisson process [55].

Can Wisp enforce performance isolation? The workflow DAGs for this experiment are indicated in Fig. 6 (right). Workflows *A-D* are read-write workloads with an arrival rate of 100rps each at the FEs. Every request from these workflows at a PS triggers a read and write request to the EN in sequence followed by some compute work at the PS. Workflow *E* issues metadata queries that are serviced locally by the PS without any interactions with the EN layer. Workflow *F* is an aggressive tenant's workload generated by four clients that exceeds its fair share at the EN tier. Workflow *G* is bursty traffic with an arrival rate of one request every two seconds, each of which triggers 400 reads in parallel, followed by 400 writes to the ENs, each of which requires a processing time of 200ms on average.

We compare performance across three scenarios: *(i)* Timeouts only, where the system only makes use of deadline based timeouts and does not use fair-queuing or rate-control (baseline), *(ii)* rate control only (RC), and *(iii)* rate control and fair scheduling (FQ+RC).

Fig. 7 demonstrates system behavior across the three scenarios. When workflow *F*, the aggressive tenant, activates at time t=50s, the resulting overload drives all workflows to throughput collapse. Given the RPC library's request timeout of ten seconds, requests queue up internally in the system, blocking different resources and thus inflating latencies for all workflows.[1] Instead, with Wisp's rate-adaptation (Fig. 7) this is not the case. The rate-limiting throttles workflow *F* whereas other workflows retain their expected throughput. Since *F* is an open-loop workload that does not lower its sending rate despite being throttled, its throughput exhibits the instability seen in the oscillations in Fig. 7 (center). However, rate-control alone does not guarantee fair access to local resources at each service. This means that workflows with a stable rate have a higher degree of presence across system-wide resource queues, which cause bursty workloads to suffer from head-of-line blocking. Workflow *G* thus suffers because of the higher queue occupancies from the other workflows, indicated by a timeout fraction of 12% (Fig. 7 (right)). Instead, with the combination of per-service fair-scheduling and rate-control, *G* is guaranteed progress at each stage. The key take-away here is that thresholds such as timeouts are challenging to set correctly system-wide with an end-to-end performance objective in mind. In practice, such thresholds are often hard-coded [45] and therefore fragile. On the other hand, Wisp automatically adapts rate limits based on dynamic system conditions.

Can Wisp meet end-to-end deadlines? We replay a trace of 30K requests from a production instance of Azure Storage, which involves a mix of different APIs. Since the traces do not indicate deadlines per request, we measure the completion times for each request in isolation. We then correct for the higher system loads in

[1] API timeouts for platforms such as Azure [4] and Google Cloud Platform [2] are often tens of seconds, if not minutes.

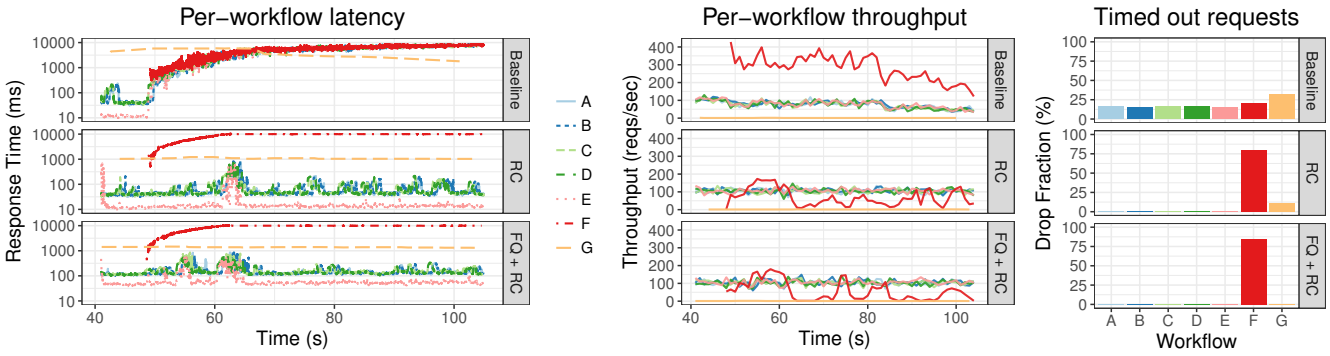

Figure 7: Performance isolation experiment. (*left*) The median smoothed latency timeseries of the experiment shows the aggressive workflow (*F*, red) driving the system into overload. End-to-end rate-control throttles *F* at the ingress and protects other workflows. (*center*) Throughput obtained by all workflows. In the baseline, all workflows' throughputs gradually degrade as requests time out. (*right*) When running rate-control only, the bursty tenant is not guaranteed performance isolation as the steady workflows dominate system queues. A fair-queuing scheduler resolves this.

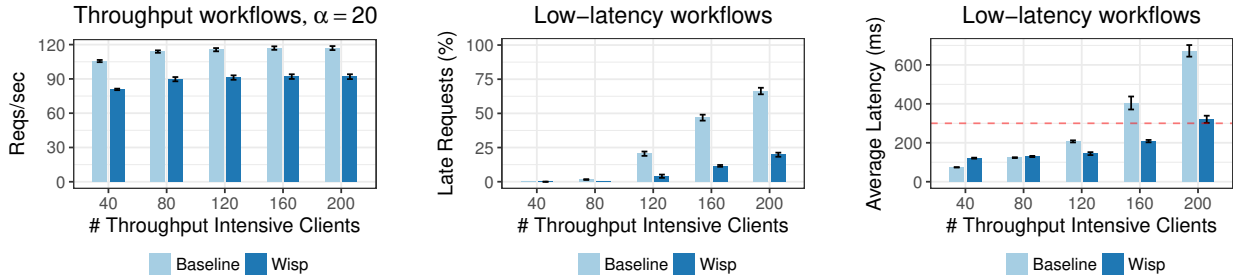

Figure 8: Average latencies and fraction of late requests for latency sensitive workflows in the presence of throughput focused clients. The low latency clients with 300ms deadlines are able to meet a high fraction of deadlines and improve average latency by 2x (200 clients).

the experiment by setting the deadline to four times the base completion time when measured in isolation. The workload includes APIs that trigger scans involving *thousands* of sub-tasks and multiple API calls that only contend for the local resource at the PS. We vary the number of client threads that generate these requests in a closed loop from 100 to 250 to increase system load.

We compare results across a baseline, FIFO+RC, EDF+RC and LSTF+RC. Table 3 indicates different percentiles for the latencies normalized by the deadline (LND). An LND of 1.0 implies that the latency equals the deadline. At higher loads, the baseline incurs increased latencies and thus misses deadlines by large factors. At the highest load of 250 clients generating requests, the baseline's average LND is 1.33, while the 99th percentile is as high as 19.71. The improvement in LND is evident when using rate limiting (FIFO+RC), since the heavier workflows are throttled and dropped before they cause downstream congestion. With 250 clients, all algorithms with rate limiting drop close to 21% of requests since it is infeasible to meet their deadlines (§4.3), freeing up resources for other requests. We also note that LSTF outperforms EDF across all runs and yields a 10x improvement over the baseline (250 clients, 99th percentile). Recall that EDF only prioritizes requests based on the proximity to the deadline. Therefore, requests might make little progress until it is too late [14]. On the other hand, LSTF *also* factors in the remaining service time which can be estimated because of Wisp's progress metrics. LSTF highlights the benefits of scheduling based on end-to-end characteristics of requests using Wisp.

Can Wisp protect low-latency workflows? A common scenario in cloud storage systems is the co-existence of throughput intensive workflows which involve bulk reads/writes as well as low-latency workflows that have soft deadlines. Here, we run Wisp with the bottleneck fairness policy in conjunction with the fair scheduler. We vary the number of throughput intensive clients from 40 to 200, each of which runs in a closed loop. Every request from this workflow arriving at a PS triggers twenty sub-tasks to the EN. Six low latency clients (one workflow each) submit requests at a rate of 10 requests per-second (60 rps in total), with every request having a 300ms deadline. Fig. 8 illustrates our results. When only 40 high-throughput clients are active, Wisp and the baseline successfully meet all deadlines. The baseline presents an improved average latency for the low-latency clients because it does not incur the added overhead of our DRR-based fair scheduler (also observed by [62]). However, at higher loads, Wisp's performance degrades gracefully, with a high fraction of admitted requests meeting their deadlines (~80% hit rate with 200 high throughput clients). Our implementation cannot guarantee latency for a request unless it *(i)* compromises on being work-conserving or *(ii)* preempts on-going work at any resource (which, for resources such as locks or connection objects is not practical, and has non-trivial ramifications on user code logic). Thus, a

# Clients	Algorithm	LND (Mean)	LND (p95)	LND (p99)
100	Baseline	0.39	0.75	0.98
	FIFO+RC	0.34	0.64	0.95
	EDF+RC	0.30	0.52	0.71
	LSTF+RC	0.32	0.71	0.81
150	Baseline	0.61	**1.32**	**5.68**
	FIFO+RC	0.33	0.60	0.84
	EDF+RC	0.31	0.52	0.69
	LSTF+RC	0.3	0.51	0.71
200	Baseline	**1.09**	**2.75**	**18.47**
	FIFO+RC	0.71	**1.63**	**2.95**
	EDF+RC	0.38	0.79	**1.17**
	LSTF+RC	0.34	0.61	0.87
250	Baseline	**1.33**	**4.83**	**19.71**
	FIFO+RC	0.98	**2.58**	**10.5**
	EDF+RC	0.49	**1.25**	**2.15**
	LSTF+RC	0.46	**1.12**	**1.82**

Table 3: Mean, 95th percentile and 99th percentile latencies normalized by the deadline (LND) for requests from the production workload. An LND of 1.0 means the end-to-end latency equals the deadline.

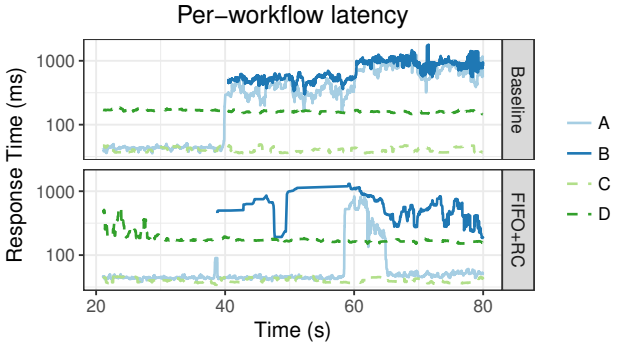

Figure 9: Skewed workload test, with a median smoothened timeseries of the latencies. Workfow *B* starts at t=40s, contends at the same PS workflow *A* is being routed through, and doubles its sending rate at t=60s. Wisp's rate-limiting shields *A* from *B* in both instances, without throttling background workflows (dashed, within their fair shares).

request at a service can get unlucky due to bad timing: a sub-task from a low-latency workflow may arrive *right after* a burst of other workflows are scheduled (and thus suffer head-of-line blocking at the local resources). Wisp shapes the high-throughput clients to their fair share of resources alongside the low-latency clients.

Can rate adaptation react to hotspots? We now evaluate a scenario where we create a skewed access pattern. Four workflows activate at different times. *A* and *B* are consistently routed to the same PS, causing a hotspot on the local semaphore protected resource. *A* is an open loop workflow generated by a single client. *B* starts at $t = 40s$ with thirty clients, and at $t = 60s$ doubles its sending rate with an additional thirty clients joining the system. To study the rate adaptation algorithm in isolation, we only use a FIFO scheduler at each process. *C* and *D* are background workflows that exert pressure on the ENs. Fig. 9 shows a rolling median of the latency timeseries with and without Wisp. In the baseline, when *A*

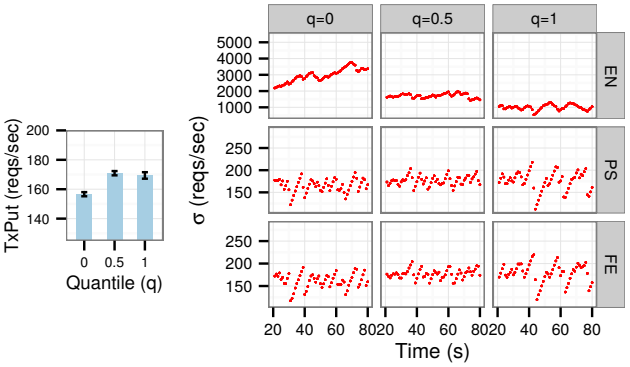

Figure 10: σ adaptation from the ENs to the FEs for a workflow, for different values of q. ENs adapt their rates independently based on their demands, PS's aggregate the rates to calculate their local σ using a quantile, and the FE's repeat the same procedure against the PS'.

activates at $t = 40s$, the surge of client requests immediately cause contention at all queues at the PS, including the shared sub-task scheduler queue as well as a semaphore being contended for. The resulting head-of-line blocking inflates latencies for *B*. When using Wisp, *A* experiences a spike in latencies at $t = 40s$ when *B* activates. However, when *B* doubles its sending rate at $t = 60s$, it forces head-of-line blocking and higher latencies for *A* as with the baseline (recall, we are not using local fair-queuing here). However, Wisp soon computes rate limits based on the observed costs of *B*. When the rate-limiters and system queues stabilize from the unexpected surge, *A* retains its expected latencies, whereas *B* is throttled at the entry point. Wisp also (correctly) avoids throttling the background workflows which are not contributing to congestion (and stay within their fair shares).

Quantile knob sensitivity analysis. We show the impact of the quantile knob q of the rate adaptation mechanism. We consider a scan workflow generated by 180 client threads, which triggers ten back-to-back requests between the PS and ENs. This workflow competes with a lighter open-loop workflow for system resources, triggering our max-min fairness policy. We study the impact of the q on the scan workflow's throughput. Fig. 10 (right) shows a timeseries where each data point represents $\sum \sigma$ (the total advertised rate limit) per-tier in a second for the scan workflow, for different values of q. With $q = 0$, each PS selects the minimum advertised σ from the individual EN processes, and the FEs repeat the procedure with the PS processes. This leads to conservative σ values at the FEs, leading to low utilization at the ENs. This is evident in that *(i)* the EN tier probes for more demand by advertising higher rates (Fig. 10 (top row, left)), and *(ii)* lower end-to-end throughput for the workflow (Fig. 10 (left)). With $q = 1$, the rates are *max* aggregated, and the system thus admits more work up to the ENs. This results in increased congestion at the ENs, which therefore exert back-pressure by announcing lower rates. With $q = 0.5$, for our setting, upstream services match the capacity of the ENs; σ at FEs does not oscillate unlike $q = 0$ and $q = 1$, leading to a stable load at the EN (Fig. 10 (right)).

Overheads. In our experiments, we did not see performance overheads from Wisp in comparison to a baseline with rate limiting and scheduling disabled (e.g., latencies in Table 3 and Figure 9). Furthermore, SOA frameworks already ship with the building blocks necessary for Wisp such as rate limiters, thread schedulers and metadata propagation. They often monitor several metrics that Wisp merely leverages to perform adaptive rate control and request scheduling (Fig. 5). For instance, Hystrix [50] already aggregates tens of metrics by "command type". Xenon [72] already performs throttling per-workflow by tenant using static rate limits.

7 DISCUSSION

Relation to auto-scaling. An alternative approach to handling overload is to dynamically scale resources [27, 54]. We believe that scaling resources is orthogonal to Wisp's rate control, as both approaches operate at different timescales. The former adds capacity in response to demand changes whereas Wisp regulates the demand according to the capacity. Auto-scaling typically works over slower time scales (minutes) than Wisp's mechanisms (per-request and sub-second decisions). Furthermore, overload can be triggered by performance bugs [32, 32, 34] and slowdowns in third party services. Auto-scaling is not a silver bullet for these settings.

Network congestion control. Intuitively, one can view the problems addressed by Wisp through the lens of network congestion control. However, the workflow rate limiting problem in SOAs differs fundamentally from the network congestion control problem. In the TCP context, sources perform rate limiting and coordinate with endpoints for flow control. They infer congestion in the network either indirectly through congestion signals (e.g., packet loss and latency) or through explicit feedback [1, 24] to tune their sending rates. Endpoints of a network flow are fixed (even for multicast congestion control [5]). These assumptions do not hold in an SOA. Upstream services do not know about their transitive dependencies, and due to effects such as request amplification, caching and routing, subsequent requests of the same workflow may be processed by entirely different downstream processes. Upstream services also do not know a-priori the load they impose downstream. Wisp therefore uses adaptive rate limiting and scheduling techniques that account for the unique characteristics of SOAs.

Centralized control. Finally, one may consider applying centralized control, as in Retro [45]. However, centralized coordination hinders *per-request* resource management (as opposed to per-workflow); e.g., scheduling decisions that account for complex workflow DAG structures. Furthermore, Retro's approach does not see causally disjoint request execution paths in the system, and instead, throttles *all* points through which a workflow traverses. Instead, Wisp only throttles a workflow along the causal path leading to a bottleneck.

Path to deployment. Wisp leverages building blocks typically available in SOA frameworks such as rate limiters, request schedulers and metadata propagation. The rate limiting and scheduling techniques do not depend on each other and can be used in isolation (e.g., §6 demonstrates the use of rate limiting without scheduling enabled). Lastly, a system may incrementally deploy Wisp, gradually expanding the set of services that rely on Wisp for adaptive control.

8 RELATED WORK

SOA libraries. Hystrix [50] and Finagle [69] are libraries created at Netflix and Twitter to harden their production systems. They use techniques such as circuit breaking to provide resilience to failures and overload. However, they require extensive tuning of configuration parameters [51], and cannot enforce multi-tenant resource management policies *end-to-end* akin to Wisp.

Admission control and latency reduction mechanisms. Trading off completeness for latency is a common approach to managing overload [37, 43]. User code in Wisp can use these techniques to gracefully degrade service. [23, 33, 44, 75] are potential admission control policies for Wisp. [59] focuses on the design of distributed rate limiters for network flows. Themis [40] manages overload for federated stream processing systems by degrading query quality fairly across users, presenting a potential rate limiting policy for Wisp.

Network flow scheduling and congestion control. Recent work has focused on resource allocation and scheduling to quickly complete one or multiple transfers [15, 16, 22], achieving low latency [9, 29, 36, 38, 76] and fair sharing network bandwidth [42, 57, 58]. While they provide insights for potential Wisp policies, these specific solutions do not directly apply because of differences between network flows and request execution in SOAs (§7).

Distributed systems. We discuss Retro [45] in §7 as an alternative to Wisp. Pulsar [10] provides an abstraction of a virtual data center where tenants run VMs, access appliances, and a centralized scheduler enforces rates at the level of *network flows*. Pulsar focuses only on throughput goals and cannot enforce request-level scheduling decisions since appliances are treated as blackboxes. Kwiken [37] considers interactive systems where requests execute in a DAG of services. However, it relies mostly on centralized resource management whereas Wisp is fully decentralized. Enforcing high-level scheduling policies and fair sharing have been explored in the context of distributed storage systems [30, 31, 62, 67, 73, 74]; however, they typically consider simpler execution structures (e.g., client to server) whereas Wisp focuses on a general DAG wherein individual processes lack end-to-end visibility. Lastly, several proposals exist for optimizing job completion times for DAGs of tasks in big-data systems [11, 28, 77, 78]. However, data analytics jobs are often orders of magnitude longer than those serviced by the SOA systems targeted by Wisp (which operate under the additional constraint of limited end-to-end visibility).

Stream processing systems. Stream processing systems often support backpressure along the processing topology. The topology of a streaming job is part of the job specification and is therefore known in advance. Heron [41] takes an approach where a slow processing node stops asking for new tuples, causing upstream buffers to fill up, eventually leading to queuing at the ingress nodes. Heron relies on static thresholds on the buffer sizes to trigger backpressure. Storm [66] uses Zookeeper [6] to coordinate backpressure across nodes. Das et al. [19] propose dynamically adjusting batch sizes to improve latency and throughput. Contrary to these approaches, Wisp's rate limiting approach does not assume knowledge of the full

service topology, dynamically computes rate limits based on measured resource utilization instead of static thresholds, is multi-tenant aware, and does not require centralized coordination.

9 CONCLUSION

In this paper, we highlight unique challenges in managing resources for SOAs. We present Wisp, which targets achieving end-to-end throughput and latency objectives with minimal operator intervention. To this end, Wisp applies a combination of techniques, including estimating local workload models based on measurements of immediate neighborhoods, distributed rate control and metadata propagation. Our design incorporates important practical considerations: It leverages existing building blocks in SOA frameworks (rate limiters and schedulers), does not require centralized coordination and does not assume prior knowledge of request characteristics.

ACKNOWLEDGEMENTS

We thank our shepherd, Iqbal Mohomed, and the anonymous reviewers for their feedback. We thank Madan Musuvathi, Srikanth Kandula, and Virajith Jalaparti for the useful discussions that helped shape this paper.

REFERENCES

[1] 802.1Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html. 2016.
[2] Dealing with DeadlineExceededErrors. https://cloud.google.com/appengine/articles/deadlineexceedederrors. 2012.
[3] Openstack. https://www.openstack.org/.
[4] Setting Timeouts for Blob Service Operations. https://msdn.microsoft.com/en-us/library/azure/dd179431.aspx. 2016.
[5] TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification. https://tools.ietf.org/html/rfc4654, 2006.
[6] Apache Zookeeper. http://zookeeper.apache.org/, 2008.
[7] Summary of the October 22, 2012 AWS Service Event in the US-East Region, 2012. https://aws.amazon.com/message/680342/.
[8] Docker at Spotify. http://goo.gl/53t3XN, 2013.
[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.
[10] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 233–248, Berkeley, CA, USA, 2014. USENIX Association.
[11] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 285–300, Broomfield, CO, Oct. 2014. USENIX Association.
[12] E. A. Brewer. Lessons from giant-scale services. *Internet Computing, IEEE*, 5(4):46–55, 2001.
[13] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.
[14] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Real-Time Systems*, 3(3):307–336, 1991.
[15] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *ACM Sigcomm Computer Communication Review*, 41(4):98–109, Aug. 2011.
[16] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.
[17] B. Christensen. Application Resilience in a Service-oriented Architecture. http://goo.gl/0TKDmQ, 2013.
[18] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, volume 10 of *USITS '99*, pages 243–254, 1999.
[19] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA, 2014. ACM.
[20] J. Dean and L. A. Barroso. The Tail At Scale. *Communications of the ACM*, 56:74–80, 2013.
[21] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.
[22] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '14, pages 431–442, New York, NY, USA, 2014. ACM.
[23] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, USITS'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.
[24] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *ACM Sigcomm Computer Communication Review*, 36(1):59–62, Jan. 2006.
[25] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design* . Prentice Hall, 2005.
[26] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.
[27] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. IEEE, 2010.
[28] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '14, pages 455–466, New York, NY, USA, 2014. ACM.
[29] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 1–14, Oakland, CA, May 2015. USENIX Association.
[30] A. Gulati, I. Ahmad, C. A. Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *Proceedings of the International Conference on File and Storage Technologies*, FAST '09, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.
[31] A. Gulati, A. Merchant, and P. J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, pages 437–450, Berkeley, CA, USA, 2010. USENIX Association.
[32] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '14, Berkeley, CA, 2013. USENIX.
[33] V. Gupta and M. Harchol-Balter. Self-adaptive admission control policies for resource-sharing systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 311–322, New York, NY, USA, 2009. ACM.
[34] B. Harry. Explanation of July 18th outage. http://goo.gl/DPuJBm, 2014.
[35] R. G. Herrtwich. *An introduction to real-time scheduling*. International Computer Science Institute, 1990.
[36] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 127–138, New York, NY, USA, 2012. ACM.
[37] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up Distributed Request-Response Workflows. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pages 219–230, New York, NY, USA, 2013. ACM.
[38] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '15, pages 435–448, New York, NY, USA, 2015. ACM.

[39] E. Jones. Retries considered harmful. http://www.evanjones.ca/retries-considered-harmful.html, 2015.

[40] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch. Themis: Fairness in federated stream processing under overload. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, pages 541–553, New York, NY, USA, 2016. ACM.

[41] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[42] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[43] G. Kumar, G. Ananthanarayanan, S. Ratnasamy, and I. Stoica. Hold 'em or fold 'em?: Aggregation queries under performance variations. In *Proceedings of the European Conference on Computer Systems*, EuroSys '16, pages 7:1–7:14, New York, NY, USA, 2016. ACM.

[44] L. Lu, L. Cherkasova, V. de Nitto Persone, N. Mi, and E. Smirni. *AWAIT: Efficient overload management for busy multi-tier web services under bursty workloads*. Springer, 2010.

[45] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 589–603, Oakland, CA, May 2015. USENIX Association.

[46] J. Mace, P. Bodik, M. Musuvathi, R. Fonseca, and K. Varadarajan. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '16, pages 144–159, New York, NY, USA, 2016. ACM.

[47] C. Munns. I Love APIs 2015: Microservices at Amazon. http://goo.gl/aVWlpY, 2015.

[48] Netflix. Adopting Microservices at Netflix: Lessons for Architectural Design. https://goo.gl/sNuQPj, 2012.

[49] Netflix. Embracing the Differences : Inside the Netflix API Redesign. http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html, 2012.

[50] Netflix. Introducing Hystrix for Resilience Engineering. http://goo.gl/h9brP0, 2012.

[51] Netflix. Strategy for tuning the hystrix configuration. https://github.com/Netflix/Hystrix/issues/866, 2015.

[52] S. Newman. *Building Microservices*. O'Reilly Media, 2015.

[53] Nginx. Adopting Microservices at Netflix: Lessons for Team and Process Design. https://goo.gl/KOrUfT, 2015.

[54] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '13, pages 69–82, San Jose, CA, 2013. USENIX Association.

[55] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[56] Pivotal Software. RabbitMQ. https://www.rabbitmq.com/, 2012.

[57] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 187–198, New York, NY, USA, 2012. ACM.

[58] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pages 351–362, New York, NY, USA, 2013. ACM.

[59] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '07, pages 337–348, New York, NY, USA, 2007. ACM.

[60] Y. Sharma, P. Ajoux, P. Ang, D. Callies, A. Choudhary, L. Demailly, T. Fersch, L. A. Guz, A. Kotulski, S. Kulkarni, S. Kumar, H. Li, J. Li, E. Makeev, K. Prakasam, R. V. Renesse, S. Roy, P. Seth, Y. J. Song, B. Wester, K. Veeraraghavan, and P. Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 351–366, Oakland, CA, 2015. USENIX Association.

[61] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *ACM Sigcomm Computer Communication Review*, 25(4):231–242, Oct. 1995.

[62] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association.

[63] S. Souders. Velocity and the bottom line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009.

[64] SoundCloud. Building products at SoundCloud - part I: Dealing with the monolith. https://goo.gl/Qra2tA, 2014.

[65] J. A. Stankovic, K. Ramamritham, and M. Spuri. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[66] A. Storm. http://storm.apache.org/, 2011.

[67] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[68] S. Tonse. MicroServices at Netflix - challenges of scale. http://goo.gl/9j5wSv, 2014.

[69] Twitter. Finagle: A Protocol-Agnostic RPC System. https://goo.gl/ITebZs, 2011.

[70] Uber. Service-oriented Architecture: Scaling the Uber Codebase as We Grow. https://eng.uber.com/soa/, 2015.

[71] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, Bordeaux, France, 2015. ACM.

[72] VMware. Xenon. https://github.com/vmware/xenon, 2017.

[73] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '12, pages 14:1–14:14, New York, NY, USA, 2012. ACM.

[74] H. Wang and P. Varman. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-aware Allocation. In *Proceedings of the International Conference on File and Storage Technologies*, FAST '14, pages 229–242, Berkeley, CA, USA, 2014. USENIX Association.

[75] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 63–69, New York, NY, USA, 2002. ACM.

[76] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.

[77] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[78] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '12, pages 53–62, New York, NY, USA, 2012. ACM.