Charon: A Framework for Microservice Overload Control

Jiali Xing, Henri Maxime Demoulin, Konstantinos Kallas, Benjamin C. Lee
University of Pennsylvania
{xjiali,maxdml,kallas,leebcc}@seas.upenn.edu

ABSTRACT

Overload control is an important feature of modern cloud applications. As these applications grow increasingly complex, designing efficient overload control schemes at scale is tedious. In this paper we argue part of the challenge is a lack of first principles mechanisms one can use to design scalable and verifiable policies.

We present CHARON, a market-based scheme for large scale service graphs. Unsurprisingly, CHARON relies on tokens to negotiate the acquisition of compute resources. However, unlike existing receiver-driven systems, CHARON decouples the mechanisms used to generate and value tokens and proposes efficient price propagation mechanisms. We motivate CHARON with a set of representative system conditions that existing frameworks cannot handle well, detail an example policy one can build using the proposed mechanisms, and discuss open research challenges our framework exposes.

ACM Reference Format:
Jiali Xing, Henri Maxime Demoulin, Konstantinos Kallas, Benjamin C. Lee. 2021. Charon: A Framework for Microservice Overload Control. In *Proceedings of The 20th ACM Workshop on Hot Topics in Networks (HotNets'21).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Modern cloud applications are increasingly complex. Providers of Internet services, such as Twitter, Netflix, and Amazon, deploy complex service graphs made of hundreds of microservices [4, 8, 9]. Detecting, diagnosing, and mitigating performance losses for these applications is difficult. Detection is delayed because losses become apparent only after the consequences of a slow microservice propagate through the service graph [11, 17]. Diagnosis is challenging when application latency is affected by several dependent microservices. Decision making is complicated as the system must control load and allocate resources for each microservice.

Overload is a particularly challenging source of performance loss in service graphs. Load variations and bursts of traffic often cause an application's computational demands to exceed allocated system resources, causing requests and tasks to linger in queues and, eventually, violate service level objectives (SLOs). As a result, overload control is crucial for application performance. Control frameworks usually propagate performance signals between pairs of senders and receivers. These signals trigger actions, such as rate limiting at the senders or dropping requests at the receivers, to manage application performance and system utilization. Despite recent advances, however, designing an effective set of signals and actions for the unique requirements of microservice graphs remains difficult. An effective overload control scheme should satisfy the following desiderata.

Topology Awareness. Each microservice may act with a local view of system conditions, but must also anticipate broader implications for the application and system. Actions taken to mitigate overload should account for a request's computational path through the microservice graph. Topology awareness allows overload control to pursue broader objectives such as application fairness and resource efficiency.

Request Awareness. Applications may serve diverse queries of varying complexity and computational intensity. Data center workloads often follow a heavy-tailed distribution where infrequent, complex queries require an order of magnitude more time or resources than common, simple queries [3, 10, 19, 20]. Request awareness allows to differentiate between load classes and their computational demands.

Objective Flexibility. Data center operators may optimize diverse objectives for applications and the system. Control policies may need to balance fairness, goodput, and efficiency. Objective flexibility requires overload control to define mechanisms (*i.e.*, signals, actions) expressive enough to support varied policies.

Finally, given the distributed nature of service graphs, minimizing communication costs is particularly important. Performance signals must flow quickly through the graph and mitigation actions must be timely to handle bursty arrivals.

HotNets'21, November 10-12, 2021, Virtual Event, UK

ACM ISBN 978-1-4503-7020-2.
DOI: 10.1145/3484266.3487378

The trend toward microsecond-scale computation only exacerbates these requirements [12, 16, 22, 29].

We pursue these desiderata with CHARON, a scalable framework that controls overload for complex service graphs. CHARON integrates market primitives — tokens and price tables — to determine how an upstream microservice (sender) consumes resources of a downstream microservice (receiver).

CHARON employs tokens that are used as currency when a client (sender) requests a computation from a microservice (receiver). Prior approaches also use tokens and draw inspiration from network flow control [6, 24], requiring receivers to generate tokens based on system conditions and to communicate those tokens to senders as a means of acquiring access to compute resources. In contrast, CHARON decouples token generation and valuation, shifting the responsibility for token generation from receivers to senders. This eliminates the network RTT from token generation and ensures generation proceeds smoothly even when services are overloaded. This also decouples mechanism and policy as senders can generate tokens at rates determined by a customizable policy.

CHARON employs price tables, which allow microservices to determine how many tokens to spend. Each microservice updates its prices based on the current policy and other dynamic system conditions. Prices are propagated across the service graph alongside responses using an application-level protocol, enabling topology-aware policies that scale to complex, multi-layer applications. The cost of entry-level endpoints are similarly propagated and advertised to clients. With tokens and prices, CHARON creates a market for microservice computation. By varying policies for tokens generation and price updates, operators can tailor overload control for application needs.

2 BACKGROUND

Overload control relies on rate limiting senders and active queue management (AQM). Both mechanisms have seen widespread use to avoid congestion in networks; for example, see Jacobson's CoDel [26] and protocols such as RCP [13, 34]. At the application level, SEDA [32] and Gatekeeper [14] adaptively control overloads with request-aware AQM, which drops only the requests responsible for overload, and rate limiting senders with token bucket algorithms [28]. In today's microservice architectures, applications rate limit with circuit breakers [30] and traffic shapers [25].

Overload control for microservices presents unique challenges. Assessing load between sender-receiver pairs is difficult as the number of microservices scales. Clients issuing requests may suffer from limited network visibility and inaccurate rate limiting, which risks RPC incast [5]. When many clients issue requests simultaneously, overloaded queues may drop requests and cause goodput to collapse. In addition, dropping requests is expensive and create overheads comparable to processing times when services compute in microsecond timescales [12, 16, 22, 29]. As a result, client goodput becomes more sensitive to inaccurate rate limiting. The risks of dropping requests and wasting computation in the execution path increases with the number of services on that path.

Addressing these issues, recent work proposes receiver-driven overload control at the network [24] and application levels [6]. With these techniques, receivers manage their own utilization by deciding the rate at which senders can request service and consume resources. Both techniques use a basic notion of currency. Homa grants senders the right to issue a certain number of bytes, prioritizing short RPCs. Breakwater grants senders tokens and requires one token for every request sent, adjusting the number of available tokens based on receivers' load.

These recent works draw inspiration from strategies in network flow control, but do not fully address the complex interactions between multiple layers or tiers of a service graph. DAGOR [35] proposes overload control for multiple layers of a service graph by enforcing performance objectives and business priorities between pairs of senders and receivers. However, its rate limiting scheme is hard-coded for each pair of services. It does not consider load on dependent services that are not immediate neighbors, nor does it consider rate limiting end-users.

Finally, no existing approach supports diverse SLOs. Applications may differ in required performance and tolerable tradeoffs between response time and goodput. Fairness between classes of computation may be important for applications that serve third-party users. Efficiency and hardware utilization may be important for applications that serve first-party users.

3 MOTIVATION

We now present our system model and CHARON's application targets. We then present scenarios showcasing how topology awareness, objective flexibility, and request awareness are important for an effective overload control scheme.

3.1 System Model

We consider applications represented as a Directed Acyclic Graph (DAG) describing microservices (nodes) and their dependencies (edges). Clients issue requests, which enter through the graph's source nodes and follow varied execution paths. Upon receiving a request for computation, each node may issue one or more sub-requests, creating chains and fanout patterns. A node may require all downstream nodes to serve their sub-requests before replying to its upstream node. We assume that given a request type, each node knows precisely and deterministically the set of downstream nodes

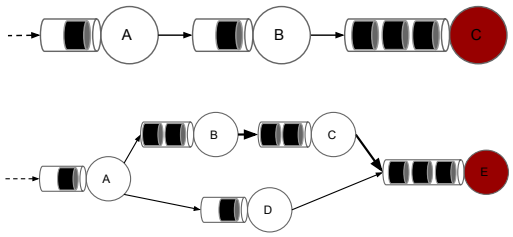

Figure 1: Overload scenarios in (a) microservice chain and (b) where two execution paths contribute unevenly.

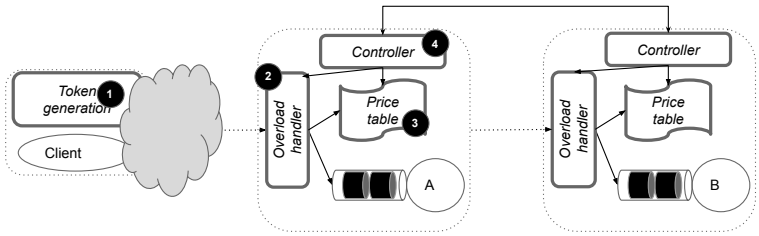

Figure 2: CHARON design. Clients use a generation module to periodically generate tokens. Application services maintain a price table used by an overload handler to perform admission control. A local controller set with the desired policy configures these modules.

that will next process the request. We also assume each node implements a First Come, First Serve (FCFS) request schedulers. However, the choice of scheduler is orthogonal because CHARON's tokens are used only to determine admissions and not scheduling priorities.

The system is evaluated from two perspectives. From the client's perspective, requests' performance is evaluated against SLOs defining the conditions under which computation is useful — for example, 99.9% of requests should complete within 10ms. From the operator's perspective, system dynamics are evaluated against utilization, efficiency, and fairness metrics. Utilization measures resource usage (*e.g.*, throughput). Efficiency examines whether usage translates into useful work (*e.g.*, goodput). Finally, fairness determines whether resource allocations and service rates align with applications' relative importance (*e.g.*, reservations, entitlements, priorities).

3.2 Microservice Characteristics

Long request paths. Long paths in the application graph delay the propagation of performance signals, harming responsiveness for applications with bursty requests. For example, Figure 1(a) presents an application that consists of three microservices arranged in a chain. Suppose leaf node *C* becomes overloaded. Overload control schemes that manage sender/receiver pairs must first rate limit *B*, which then accumulates requests in its queue and eventually triggers rate limiting for *A*. In contrast, *topology-aware* overload control would directly signal the path's entry point to rate limit the end-users and reach its management objective more quickly.

Heterogeneous request paths. Heterogeneous execution paths through the graph of microservices motivate *objective-flexible* mechanisms that can support diverse policies. For example, Figure 1(b) presents an application with two independent paths both requesting service from *E*. Suppose *C*'s requests induce overload at *E*. In this setting, overload control schemes encounter a trade-off between efficiency and fairness. For efficiency, dropping requests from the short path via *D* avoids wasting the work completed earlier in the long path via *C*. For fairness, requests might need to be dropped and served at the same rate.

Heterogeneous requests. Data center workloads are often heavy-tailed and characterized by rare, expensive requests intermingled with many, simple ones. Such heterogeneous requests motivate overload control mechanisms that can offer differentiated service rates. Consider Facebook's USR pool [3] where requests are 99.8% GETs and 0.2% DELETEs. GET service times are much shorter than DELETE — sometimes orders of magnitude shorter [27]. If workers are busy processing DELETEs and receive a burst of GETs, the latter will be queued and, under simplistic overload control schemes, dropped. In contrast, a request aware system can price DELETE requests higher than GETs. Differentiated prices will more effectively rate limit requests, preventing overload and dropped requests before the requests queue.

Fanout request paths. We now discuss an example showcasing the need for both topology and request awareness. Consider an application for movie ratings that consists of three services: a frontend service *A* and two backend services *B* and *C* serving user timelines and movie profiles respectively. *A* provides three endpoints, *get-user-timeline* that loads the timeline of a specific user by requesting *B*, *get-movie-profile* that loads the profile of a specific movie by requesting *C*, and finally *frontpage* that combines several movie profiles and user-timelines by requesting both *B* and *C*. If *C* is the only overloaded service, the control policy should not rate-limit *get-user-timeline* requests at the frontend. Further, in cases where both *B* and *C* are overloaded, the control policy needs topology and request awareness to drop sub-requests belonging to the same user's request, minimizing wasted work, increasing goodput, and optimizing user experience.

4 CHARON

Figure 2 presents CHARON's overall design. ❶ Clients are extended with a *token generation* module that can be configured for the policy at hand. ❷ Application microservices are extended with an *overload handler* responsible for admission control. ❸ Using a *price table*, the overload handler credits incoming requests while a ❹ *controller* updates prices based on signals such as queueing delay and local resource utilization, and configures the overload handler based on the policy in effect. CHARON's components extend existing RPC libraries, such as Thrift [2] and gRPC [7], and use an application-layer protocol to convey price tables and tokens.

Note that only end-users are responsible for generating tokens, allowing the system to control arrival rates without explicitly granting tokens to clients and reducing CHARON's protocol overheads. Responses piggy-back relevant price tables entries to clients such that, after the first message, clients can estimate the number of tokens required to request service from an application endpoint.

4.1 Mechanisms

Token Generation. Clients generate tokens. To send a request, a client first checks the price table to determine whether it holds a sufficient number of tokens. If so, the client accordingly decrements its token holdings. Generating tokens at the client has two significant advantages. First, servers need not consume bandwidth granting tokens to clients and clients need not wait for grants to request the application. Second, the system is flexible and can pursue varied objectives. The overload control policy determines the rate at which tokens are generated and rates can be configured relative to other clients to explore trade-offs between utilization, efficiency, and fairness. Third, the system enables dynamic decision making and flow control. The overload control policy might allow clients to spend more tokens for prioritized service, especially during periods of high load. Dynamic spending and pricing decisions may lead to more efficient allocation of system resources.

Price Tables. Each application endpoint is associated with a price, quantified in terms of tokens, for processing and service. The price for a request is affected by prices for all of the sub-requests made to various microservices along the execution path. Prices are dynamic, reflecting congestion and load levels at the microservice. Price updates propagate through the microservice graph, piggybacked on responses, so that clients are aware of the latest prices.

Price tables, when coupled with a token mechanism, implement rate limiting. For example, when clients double their request rates and cause overload, microservices can respond by doubling prices such that clients must throttle their request rate. CHARON relies on differentiated prices to implement request-aware overload control. Prices may depend on request type, which in turn dictates priority, execution path through the application, processing time, *etc.*

Token Usage. A microservice's overload handler enqueues a request if its accompanying tokens are sufficient given the table's posted price. Otherwise, the handler can drop the request and take additional action. One such action might be informing the sender that the request was dropped due to insufficient tokens. Another action might be to share the latest price table.

The overload handler may take one of several approaches to process a used token. One approach consumes part of the accompanying tokens based on its price for service and forwards the remaining tokens for sub-requests to downstream microservices. Another approach à la DAGOR examines queued requests' token holding to determine their relative priority, no tokens are consumed and all tokens are forwarded for sub-requests so that downstream microservices can similarly assess priority.

Controller Communication. CHARON controllers can communicate with each other out-of-band for fast information propagation. For example, when a microservice is overloaded, its controller can either update prices locally at the upstream microservice or directly at the source that issues application requests. In contrast, existing systems that rely only on pairwise sender/receiver protocols suffer propagation delays that are proportional to the length of the execution path (*c.f.*, Section 3). Out-of-band communication can be selective, minimal and, since application graphs are not known to grow above a few hundred microservices, should incur bounded overhead.

Application-Level Protocol. CHARON allows client agents and overload handlers to add metadata to requests traversing the application graph. For example, a request could carry the amount of tokens that it has spent already in the graph services, allowing the overload handlers to ensure system-wide fairness objectives by, for example, prioritizing requests that have spent more tokens in the system.

4.2 Policy

CHARON offers configurable parameters to support varied policies. First, operators must specify fixed pricing, like Breakwater's,[1] or dynamic differentiated pricing (*c.f.*, Section 5). Operators must also configure how often prices are updated. Price updates depend on evolving system conditions such as processor utilization, queueing delays, or number of requests waiting for responses from downstream microservices. Local prices might be influenced by the topology and the node's location in the application graph. In preliminary experiments,

[1] But because static prices do not react to overload, additional control mechanisms will be needed.

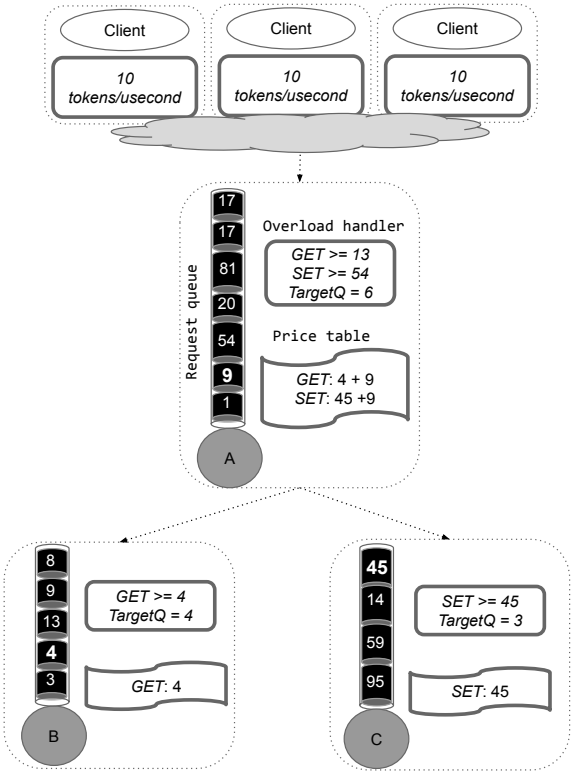

Figure 3: Microservice Graph for Policy.

we use round-trip time as the price update interval. Tuning this frequency may reveal interesting trade-offs between responsiveness and efficiency.

Second, operators must instantiate each client with a token generation policy. The rate at which tokens are generated will determine clients' sending rate. Clients' token holdings correspond to resource allocations in a shared system. These holdings can be affected by priority or entitlement policies. Tokens generation may depend on how they are spent. An overload handler might admit only requests owning the required number of tokens, or tokens might be used to signal relative priorities and influence AQM in the event of overload.

5 CASE STUDY

Figure 3 presents a simple application with two types of requests. *Get* requests traverse nodes *A* and *B* while *Set* requests traverse *A* and *C*. Assume *Set* requires longer service times than *Get*.

This application illustrates several challenges in overload control from Section 3. When *C* is severely overloaded, pairwise control cannot prevent clients from sending Set requests to *A* even as *C* throttles the volume of Sets it can receive from *A*. Unless it can distinguish Set and Get requests from clients, *A* will perform wasted computation for Set requests that cannot get serviced by *C*.

Furthermore, suppose the application in Figure 3 also serves *Plus* requests, that require subrequests to both *B* and *C*. If *Plus* overloads both *B* and *C*, forcing them to drop some of their requests instead of admitting them, overload control should coordinate the drop of dependent sub-requests at *B* and *C* to avoid wasted computation and improve goodput.

Objectives. The SLO specifies service time targets for *Get* and *Set* that are 4 and 3 times their average service times, respectively. Queue lengths at *B* and *C* are proxies for overload. *A* sets a target buffer depth (*e.g.*, six). Subject to these constraints, the control policy seeks to increase goodput, which increases with request throughput and decreases with dropped requests. The pursuit of goodput motivates topology-aware rate limiting and load shedding.

Policy Design. Clients generate tokens at a fixed rate (*e.g.*, 10 tokens per microsecond) and can spend any number of tokens up to its current holdings when issuing a request to the application graph. Requests must hold at least the required number of tokens to receive service. Requests holding fewer tokens than the posted price are dropped. Admitted or dropped, any extra tokens accompanying a request are not returned to sender. Nonetheless, it would be interesting to explore policies where extra tokens are returned. Local prices are set based on the k-th highest demand, where k is the buffer depth.

For example, suppose *B* sets its overload threshold to $k = 4$. Normally, the price of service is one token. With overload, when more than four tasks are queued, the price is set to the number of tokens held by the fourth wealthiest request in the queue. In Figure 3, *B*'s price is set to four tokens and requests holding fewer tokens are dropped. *C*'s price is set similarly. Thus, pricing policy links admission control and demand fluctuations at each microservice.

Additive Price Tables. Similar price policies apply to *A* except that its overload handler must additionally account for downstream prices. Suppose *A*'s target queue length is six requests and it has enqueued a number of *Get*s and *Set*s. The *Get* requests hold {17, 17, 20, 9, 1} tokens while the *Set* requests hold {81, 54} tokens. Further suppose that *A*'s current views of *B* and *C*'s prices are 4 and 45, respectively.

A computes its prices based on queued requests' token holdings and downstream prices. First, because its overload threshold is six requests, *A*'s base price is set to the number of tokens held by the sixth wealthiest request in its queue (here 9 tokens). Next, *A*'s prices are differentiated for request types and adds the price of downstream services (*i.e.*, 9+4=13 tokens for *Get*, 9+45=54 tokens for *Set*).

Topology- and Request-aware Rate Limiting. Nodes in the application graph modulate their request rate based on topology-aware prices. Because *A* knows *B* and *C*'s prices, it

will not waste work on requests with an insufficient number of tokens. Similarly, because clients observe A's additive price table, which reflects the total price for *Get* and *Set* requests, they will refrain from issuing requests if they hold an insufficient number of tokens.

The case study also illustrates request awareness. Overload on the *Set* path does not affect the *Get* path as long as A can still enqueue and serve requests. In contrast, control that is oblivious to request types would waste work, computing at A for requests that are then dropped at B.

Topology-aware Load Shedding. During overload, rate limiting might not be adequate and the system might resort to dropping requests. In the case of the *Plus* requests, which require sub-requests to both B and C, priced at $9+4+45=58$ tokens, requests with fewer than 58 tokens are dropped even before entering A's queue, thereby avoiding work that will be wasted further downstream due to drops.

Now suppose that A receives a burst of *Plus* requests leading to an overload at both B and C. Due to the burst, it is possible that A has enqueued many *Plus* requests before receiving the price updates from B and C. In this case, the objective is to minimize wasted work by ensuring that B and C drop the subrequests for the same *Plus* requests. Our policy spreads surplus tokens to downstream services evenly, and therefore both B and C will drop and admit the same subsets of subrequests *Plus*. The policy assumes some variability in token holdings across requests. Thus, our policy coordinates load shedding such that B and C drop the same subrequests (those with fewer tokens) without requiring communication between them.

6 DISCUSSION

Trust Model. We have focused on CHARON functionality, assuming trustworthy clients and services. However, a client might try to game the framework to obtain more resources or a malicious client could try to disrupt the entire service. When clients can tamper with token generation and spending modules, CHARON's server side would need a validation module that checks whether token spending is consistent with generation rate and history. Exploring trust models and validation techniques is a promising avenue for future research.

Auto-Scaling and Scheduling. Overload might be resolved by enlisting more resources; auto-scaling is widely supported service providers [1, 18, 23]. However, there are likely limits to the number of additional resources due to cost and setup delays. Today's auto-scaling is often an order of magnitude slower than what is necessary to maintain quality-of-service for high-performance applications [16]. Overload can be analyzed and mitigated with queueing theory [21] and scheduling, but identifying the optimal policy is difficult [33]. Coordinated auto-scaling, scheduling, and overload control is an open research challenge.

Information Staleness. In all distributed systems, there is a tension between centralization and decentralization that leads to accuracy and performance trade-offs. Achieving "0-RTT" overload control is an open research problem. One direction might be an optimized token spending policy that minimizes dropped requests.

Topology Awareness and Dynamic Control Flow. Some policies might operate with an incomplete view of the service graph whereas others might require a comprehensive view. For instance, in Section 3, only out-of-band overload control for long request paths requires a complete view of the graph. In addition, we have assumed that requests' execution path are deterministic. Alhough this assumption holds for many applications, such as data analytics and streaming engines, it falls short in others. Dynamically building views of execution paths and inferring downstream prices for a new request is an open research question. One solution maintains expected values in price tables, reflecting the likelihood that a request follows a specific downstream path.

Market Theory for Overload Control. CHARON's token and price mechanisms permit studying overload control from the perspective of market dynamics. User fairness can be viewed from a game-theoretic perspective and we might seek policies that produce solution concepts, such as a Competitive Equilibrium from Equal Incomes (CEEI), with attractive properties [15, 31]. In another example, a sudden influx of users may cause inflation as the number of available tokens increases and overload control would increase prices until the system reaches a new equilibrium.

As with other dynamic systems, there is a question of whether CHARON can converge and reach an equilibrium. We posit that as long as policies are stateful (*e.g.*, accounting for past price updates), the system can settle on price update intervals that lead to convergence.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Aurojit Panda, and the anonymous HotNets reviewers for their insightful comments. This work is partially funded by NSF CNS-1750158 and NSF 1763514.

REFERENCES

[1] Amazon Web Service. [n. d.]. AWS auto-scaling. https://aws.amazon.com/autoscaling/. ([n. d.]). Accessed: 2021-22-09.

[2] Apache Software Foundation. [n. d.]. Apache Thrift. https://thrift.apache.org/. ([n. d.]). Accessed: 2021-23-06.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[4] C4Media. [n. d.]. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture. ([n. d.]).

[5] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. 2009. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN '09)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1592681.1592693

[6] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for μs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 299–314.

[7] Cloud Native Computing Foundation. [n. d.]. gRPC, A high performance, open source universal RPC framework. https://grpc.io/. ([n. d.]). Accessed: 2021-23-06.

[8] Adrian Cockcroft. [n. d.]. Evolution of Microservices - Craft Conference. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference. ([n. d.]). https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference

[9] Adrian Cockcroft. [n. d.]. Microservices Workshop - Craft Conference. https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference. ([n. d.]). https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[11] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[12] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Loo, and Linh Phan. [n. d.]. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PersÃľphone. In *Proceedings of the 27th Symposium on Operating Systems Principles* (2021-10-26) *(SOSP '21)*. Association for Computing Machinery. https://doi.org/10.1145/3477132.3483571

[13] Nandita Dukkipati. 2008. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer.

[14] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. 2004. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*. 276–286.

[15] Duncan K. Foley. [n. d.]. Resource allocation and the public sector. 7, 1 ([n. d.]).

[16] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[17] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. [n. d.]. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019-04-04) *(ASPLOS '19)*. Association for Computing Machinery, 3–18. https://doi.org/10.1145/3297858.3304013

[18] Google. [n. d.]. Autoscaling groups of instances. https://cloud.google.com/compute/docs/autoscaler. ([n. d.]). Accessed: 2021-22-09.

[19] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 161–175. https://doi.org/10.1145/2694344.2694384

[20] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 625–638. https://doi.org/10.1145/3123939.3123956

[21] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.

[22] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for Msecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, USA, 345–359.

[23] Microsoft Azure. [n. d.]. Azure Autoscale. https://azure.microsoft.com/en-us/features/autoscale/. ([n. d.]). Accessed: 2021-22-09.

[24] 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*. 221–235.

[25] Network Working Group. [n. d.]. RFC 2475. https://datatracker.ietf.org/doc/html/rfc2475#section-2.3.3.3. ([n. d.]). Accessed: 2021-23-06.

[26] Kathleen Nichols and Van Jacobson. 2012. Controlling Queue Delay. *Commun. ACM* 55, 7 (July 2012), 42–50. https://doi.org/10.1145/2209249.2209264

[27] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 385–398.

[28] Craig Partridge. 1994. *Gigabit networking*. Addison-Wesley Professional.

[29] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325âĂŞ341. https://doi.org/10.1145/3132747.3132780

[30] Chris Richardson. [n. d.]. Evolution of Microservices - Craft Conference. https://microservices.io/patterns/reliability/circuit-breaker.html. ([n. d.]). https://microservices.io/patterns/reliability/circuit-breaker.html

[31] Hal R Varian. [n. d.]. Equity, envy, and efficiency. 9, 1 ([n. d.]), 63–91. https://doi.org/10.1016/0022-0531(74)90075-1

[32] Matt Welsh and David Culler. [n. d.]. Overload management as a fundamental service design primitive. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (2002-07-01) *(EW 10)*. Association for Computing Machinery, 63–69. https://doi.org/10.1145/1133373.1133386

[33] Adam Wierman and Bert Zwart. 2012. Is tail-optimal scheduling possible? *Operations research* 60, 5 (2012), 1249–1257.

[34] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/2018436.2018443

[35] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. [n. d.]. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing* (2018-10-11) *(SoCC '18)*. Association for Computing Machinery, 149–161. https://doi.org/10.1145/3267809.3267823