Overload Control for µs-scale RPCs with Breakwater

Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay, *MIT CSAIL*

https://www.usenix.org/conference/osdi20/presentation/cho

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Overload Control for µs-Scale RPCs with Breakwater

Inho Cho   Ahmed Saeed   Joshua Fried   Seo Jin Park   Mohammad Alizadeh   Adam Belay
MIT CSAIL

Abstract

Modern datacenter applications are composed of hundreds of microservices with high degrees of fanout. As a result, they are sensitive to tail latency and require high request throughputs. Maintaining these characteristics under overload is difficult, especially for RPCs with short service times. In this paper, we consider the challenging case of microsecond-scale RPCs, where the cost of communicating information and dropping a request is similar to the cost of processing a request. We present Breakwater, an overload control scheme that can prevent overload in microsecond-scale services through a new, server-driven admission control scheme that issues credits based on server-side queueing delay. Breakwater contributes several techniques to amortize communication costs. It engages in demand speculation, where it assumes clients have unmet demand and issues additional credits when the server is not overloaded. Moreover, it piggybacks client-side demand information in RPC requests and credits in RPC responses. To cope with the occasional bursts in load caused by demand speculation, Breakwater drops requests when overloaded using active queue management. When clients' demand spikes unexpectedly to 1.4× capacity, Breakwater converges to stable performance in less than 20 ms with no congestion collapse while DAGOR and SEDA take 500 ms and 1.58 s to recover from congestion collapse, respectively.

1 Introduction

Modern datacenter applications are composed of a set of microservices [15, 16, 36], which use Remote Procedure Calls (RPCs) to interact. To satisfy the low latency requirements of modern applications, microservices often have strict Service Level Objectives (SLOs), some measured in microseconds. Examples of microsecond-scale microservices include services that operate on memory-resident data, such as key-value stores [2, 25] or in-memory databases [41, 47]. Achieving microsecond-scale SLOs is possible under normal loads due to recent advances in operating systems [40] and network hardware [1]. However, maintaining tight SLOs remains a challenge during overload, when the load on a server approaches or exceeds its capacity.

Server overload can cause *receive livelock* [33], where the server builds up a long queue of requests that get starved because the server is busy processing new packet arrivals instead of completing pending requests. This scenario is especially challenging for microsecond-scale RPCs because small delays or bottlenecks can cause SLO violations. Further, the small resource requirements of a short RPC allows a single server to process millions of requests per second, potentially from thousands of clients [10, 35, 50]. Thus, server overload can be caused by "RPC incast" [39, 48], where a large number of clients make requests simultaneously, leading to large queue build-up at the server.

The goal of overload control is to shed excess load to ensure both high server utilization and low latency. Existing overload control schemes broadly fall into two categories. One class of approaches drop requests at an overloaded server or proxy [11, 32, 38]. Other schemes throttle the sending rate of requests at clients [4, 29, 46]. Neither of these approaches performs well for short, microsecond-scale RPCs. Dropping very short requests at the server is not practical as the overhead is comparable to the service time of the request. On the other hand, client-based rate limiting requires clients to know the state of congestion at the server to accurately configure their rate limit, but it takes at least a network round-trip time (RTT) to obtain this information. For requests with service times comparable to the RTT, the delay in reacting to congestion can hurt performance significantly.

A further challenge is to scale the overload control system to large numbers of clients. In a large-scale system, many clients have sporadic demand for a specific server, sending it requests infrequently. Determining the right rate limit for such clients is difficult since they have a stale view of the extent of congestion at the server when making a request. One solution is to explicitly probe the server before sending a request. However, exchanging messages per request to obtain congestion information can impose a high overhead for microsecond-scale RPCs.

In this paper, we present Breakwater, an overload control system for µs-scale RPCs. Breakwater relies on a server-driven admission control scheme where clients are allowed to send requests only when they receive credits from the server. It uses queuing delay at the server as the overload signal. If queuing delay is below an SLO-dependent threshold, Breakwater issues more credits to clients. Otherwise, it reduces the number of credits it issues.

Breakwater minimizes the overhead of coordination (i.e., the communication overhead for the server to know which clients need credits) using *demand speculation*. In particular, a Breakwater server only receives demand information from clients when such information can be piggybacked on requests. When all known demand is satisfied, the server distributes credits randomly to clients. This approach does not require coordination messages to determine demand in clients. However, demand speculation can lead to issuing credits to clients who do not need them at that moment. These unused credits lower server utilization. Thus, Breakwater issues extra credits to ensure high utilization. Such overcommitment introduces the potential for queue buildup at the server if many clients with credits send requests simultaneously (i.e., RPC incast). To mitigate the negative side effects of incast, Breakwater employs delay-based AQM to drop requests that arrive in bursts.

We implemented Breakwater as an RPC library on top of the TCP transport layer. Our extensive evaluation of various workloads demonstrates that Breakwater achieves higher goodput with lower tail latency compared to SEDA [48] and DAGOR [51], the best available overload control systems. For example, Breakwater achieves 6.6% more goodput and 1.9× lower 99%-ile latency with clients' demand of 2× capacity, compared to DAGOR with a synthetic workload. In addition, Breakwater scales to a large number of clients without degrading its benefits. For example, when serving 10,000 clients with memcached, Breakwater achieves 14.3% more goodput and 2.9× lower 99%-ile latency than DAGOR. Compared to SEDA for the same workload, Breakwater achieves 5% more goodput and 1.8× lower 99%-ile when the clients' demand is 2× capacity.

Breakwater is available as open-source software at https://inhocho89.github.io/breakwater/.

2 Motivation and Background

2.1 Problem Definition and Objectives

Overload control is key to ensuring that backend services remain operational even when processing demand exceeds available capacity. Overload was identified as the main cause of cascading failures in large services [11]. Transient overload can occur for a variety of reasons. For example, it may not be cost-effective to provision enough capacity for maximum load [51]. Services can also experience unexpected overload conditions (faulty slow nodes, thermal throttling, hashing hot spots, etc.) despite capacity planning.

Without proper overload control, a system could experience livelock [33], where incoming requests are starved because the server is busy processing interrupts for new packet arrivals, producing no useful work as the majority of requests fail to meet their SLOs. Even when the average of clients' demand is less than the capacity, short-timescale bursty request arrival can degrade latency for short requests. Microsecond-timescale RPCs are much more prone to performance degradation due to short-lived congestion than RPCs with longer service times [45].

RPCs with microsecond-scale execution time are prevalent in modern datacenters. Such RPCs span a variety of operations on data residing in memory or fast storage like M.2 NVMe SSDs (e.g., key-value stores [2, 25] or in-memory databases [41, 47]). The move towards microservice architectures has only increased the prevalence of such RPCs [15, 16, 36]. Further, a single server must process µs-scale requests at very high rates, possibly from thousands of clients [10, 35, 50]. To cope with µs-scale RPCs, an ideal overload control mechanism should provide the following properties:

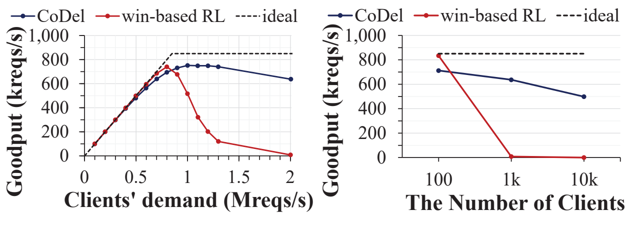
1. No loss in throughput. An RPC server should be processing requests at its full capacity regardless of overload, avoiding livelock scenarios. Further, the overhead of performing the overload control must be minimal.

2. Low latency. An ideal overload control scheme should ensure that any request that gets processed spends minimal time queued at the server. Low queuing latency ensures that processed RPCs meet their SLOs, and is particularly important for µs-scale RPCs which tend to have tight SLOs.

3. Scaling to a large number of clients. For such short RPCs, clients with sporadic demand consume very little resources at the server. Thus, high server utilization requires scaling to a large number of clients. The ideal overload control system should be resilient to "incast" scenarios when a large number of clients send requests within a short period of time. In particular, overload control should prevent queue build-ups that result from incast without harming throughput.

4. Low drop rate. Dropping requests wastes resources at the server because it must spend time processing and parsing packets that will eventually be dropped. Furthermore, dropping requests harms the tail latency of RPCs, especially when network round-trip time (RTT) is comparable to RPC execution time, making retries more expensive. Thus, overload control should minimize the drop rate at the server.

5. Fast feedback. Clients have more flexibility to decide the next action if they can discover when a request is unlikely to be served within its SLO. Thus, if a server expects a request will violate its SLO, it should notify the client as soon as possible so that it can decide an alternative action without having to wait for the request to timeout (e.g., giving up on the request, sending it to another replica, issuing a sim-



(a) Goodput vs. clients' demand with 1,000 clients (b) Goodput vs. # clients with clients' demand of 2M reqs/s

Figure 1: Goodput of CoDel and window-based rate limiting with different clients' demands and different numbers of clients

pler alternative request, degrading the quality of the service, etc. [18]).

Next, we examine existing overload control mechanisms, which were developed for RPCs with relatively long execution times. Our goal is to understand the challenges of designing an overload control system for µs-scale RPCs.

2.2 Overload Control in Practice

The fundamental concept in overload control is to shed excess load before it consumes any resources [33]. This is typically achieved by either dropping excess load at the server or throttling the sending rate of requests at the client. We look at the performance impairments of these two popular overload control approaches, developed for RPCs with long execution times, when used for µs-scale RPCs.

Active Queue Management (AQM). Such approaches operate as circuit breakers, dropping requests at a server or at a separate proxy under certain conditions of congestion. The simplest approach maintains a specific number of outstanding requests in the queue at the server, typically manually tuned by the server operator [11, 32, 37]. More advanced algorithms can improve performance and avoid the need for manual tuning. For example, CoDel maintains the queuing delay within a specific target value, dropping requests if the queuing delay exceeds the target [11, 32, 38]. RPC servers are typically required to report on success and on failure to avoid expensive timeouts [2, 37, 51]. This means that packets are processed, and failure messages are generated for dropped requests. This overhead is trivial when the message rate is low with a long execution time. However, it becomes a significant overhead in the case of µs-scale RPCs.

To demonstrate the limitations of the AQM approach, we implemented an RPC server that uses CoDel for AQM. Our main evaluation metric is the *goodput* of the server, defined as the throughput of requests whose response time is less than the SLO. Figures 1 (a) and (b) demonstrate the goodput of CoDel with different clients' demands and different numbers of clients. This experiment uses a synthetic workload of requests with exponentially-distributed service time, with a mean of 10 µs. The drop threshold parameter is tuned to achieve the highest goodput given an SLO of 200 µs. As the clients' demand increases, more CPU is used for packet processing even though majority of requests are dropped at server. As a result, less CPU can be used for RPC execution, which leads to goodput degradation. The goodput degradation gets worse with more number of clients. The reason is that the overhead of sending failure messages increases with more clients since fewer messages can be coalesced with the increased number of clients.

Client-side Rate limiting. In order to eliminate the overhead caused by dropping requests at the server, some overload control mechanisms limit the sending rate at the clients. With client-side rate limiting, clients probe the server, detect its capacity, and adjust their rate to avoid overloading the server [4, 29, 46, 49]. The reaction of clients to overload is delayed by a network RTT, which can lead to long delays when the execution time of RPCs is comparable to or less than the RTT. Further, the delay in getting feedback increases with the number of clients; consider the impact this has on overload control performance.

When the number of clients is small, the load generated by each individual client is large and each client exchanges messages with the server at a high frequency. This means that each client has a fresh view of the state of the server, allowing it to react quickly and accurately to overload. In this case, client-based approaches outperform AQM approaches because they have fresh enough information to prevent overload at the server.

As the number of clients increases, the load generated by each client becomes more sporadic and messages are exchanged at a lower frequency between any individual client and the server. This means that in the presence of a large number of clients, each client will have a stale or inaccurate estimate of server overload, leading to clients undershooting or overshooting the available capacity at the server. When many clients overshoot server capacity, it can lead to incast congestion, causing large queueing delays. AQM avoids high tail latency by dropping excess load at the server, leading to AQM outperforming client-based approach for a large number of clients, despite having less than ideal goodput.

To illustrate the limitation of client-side rate limiting with µs-scale execution time, we implement window-based rate limiting used in ORCA [29]. The mechanism is similar to TCP congestion control. The client maintains a window size representing the maximum number of outstanding requests. Upon receiving a response, if the response time is less than the SLO, it additively increases the window size; otherwise, it multiplicatively decreases the window size. Figure 1 (a) and (b) depict the goodput of window-based rate limiting for exponentially-distributed service time of 10 µs (SLO = 200 µs) on average. We optimized the parameters (i.e. additive factor and multiplicative factor) to achieve the highest goodput. Window-based schemes typically support a minimum of one open slot in the window (i.e., a minimum of one outstanding request at the server). This is problematic when there is a large number of clients as each client can always send one request, leading to incast and overwhelming the server. Rate-based rate limiting [4, 49] overcomes this limitation, but it still suffers from incast with a larger number of clients which results in high latency and low goodput.

Hybrid approaches that combine client-side rate limiting and AQM have also been proposed. We provide a more comprehensive evaluation of rate-based rate limiting and hybrid approaches in §5.

2.3 Challenges

Existing overload control schemes, developed for long RPCs, suffer significant performance degradation when handling µs-scale RPCs. The fundamental challenge facing existing schemes is the need for coordination of clients in order to schedule access to the server under very tight timing constraints. This challenge is exacerbated by the following characteristics of short RPCs:

1. Short average service times. We aim to support execution times for RPCs on the order of microseconds. This requires devising an overload control scheme that can react at microsecond granularity while keeping coordination overheads significantly less than request service times. Achieving this compromise is challenging, and any errors in devising or implementing the overload control scheme can lead to either long queues and overload, or underutilization of the server.

2. Variability in service times. RPC execution times typically follow a long-tailed distribution [11, 17, 18]. The stochastic nature of RPC service times limits the accuracy of any coordination or scheduling at the client or server. Accurate scheduling requires knowledge of the execution time of each request in advance, which is not possible in the presence of long-tailed variability of execution times. Further, this variability creates ambiguity for overload detection because a single request can be long enough to cause significant queueing delay.

3. Variability in demand. Scheduling the access of clients to the server requires some knowledge of both the demand of clients and the capacity of the server. RPCs have various arrival patterns, and clients can have sporadic demand with periods of inactivity [10, 50]. Variability in demand can lead to low utilization because clients that are granted access to server capacity might not have enough demand to utilize it.

4. Large numbers of clients. All previous challenges are exacerbated as the number of clients increases: accurate coordination becomes more challenging and overheads become higher (§5.2). Furthermore, a larger number of clients increases demand variability because it makes the system more susceptible to bursts (i.e., many clients generating demand simultaneously).

The challenges a server overload control system faces bear some similarities to those observed in network congestion control. At a surface level, network and compute congestion can be managed by similar mechanisms, but they each have fundamentally different requirements. Both are necessary to achieve good performance. Network congestion control aims to maintain short packet queues at switches while maximizing network link utilization. By contrast, overload control aims to maintain short request queues at the RPC server while maximizing CPU utilization. There are two critical differences between these problems: (a) RPC processing often has high dispersion in request service times while packet processing times are constant, and (b) client-side demand can fluctuate more significantly at the RPC layer because clients may give up after a timeout or choose to send an RPC to a backup server. On the other hand, once a network flow starts, it generally completes. With such high variability in processing time and demand, designing an overload control system requires overcoming different challenges than a network congestion control system.

2.4 Our Approach

Our work begins with insights from receiver-driven mechanisms proposed in recent work on datacenter congestion control. In receiver-driven congestion control, a receiver issues explicit credits to senders for controlling their packet transmissions, which provides better performance than conventional sender-based schemes [14, 24, 34]. Inspired by this line of work, our design has the following components:

1. Explicit server-based admission control: A client is only allowed to send a request if it receives explicit permission from the server. A server-based scheme allows for coordination that is based on the accurate estimation of the state of the server. Explicit admission control means that the load received by the server is completely controlled by the server itself. This allows for more accurate control that maintains high utilization and low latency. Server-based admission control can add an extra RTT for a client to request admission. We avoid this through piggybacking and overcommitting credits, as detailed later.

2. Demand speculation with overcommitment: The server requires knowledge of clients' demand in order to decide which client should be permitted to send requests. This is comparable to the need for clients to know about the state of the server in client-based schemes. Exchanging such information introduces significant overhead as the number of clients increases. Furthermore, as the execution time of RPCs decreases, the frequency of exchanging the demand information increases, further increasing overhead. The key difference between server-based schemes and client-based schemes is that we can relax the need for the server to have full information about clients' demand without harming performance. In particular, we allow the server to speculate about clients' demand and avoid lowering server utilization by allowing the server to overcommit, issuing more credits than its capacity.

3. AQM: Due to overcommitment, the server can occasionally receive more load than its capacity. Thus, we rely on AQM to shed the excess load. In our scheme, the need for AQM to drop requests is rare, as credits are only issued when the server is not overloaded.

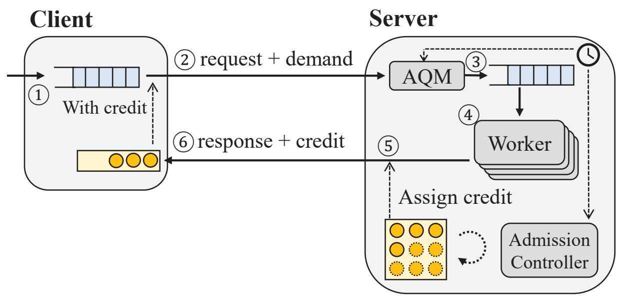


Figure 2: Breakwater overview

3 System Design

We present Breakwater, a scalable overload control system for µs-scale RPCs. Figure 2 depicts an overview of the interaction between a Breakwater client and server pair. A new client joining the system sends a register message to the server, indicating the number of requests it has in its queue. The client piggybacks its first request to the registration message. The server adds the client to its client list, and if it is not overloaded it executes the request. The server then replies to the client with the execution result or a failure message. The server piggybacks with the response any credits it issued to the client depending on the demand indicated by the client. The client issues more requests depending on the number of credits it received. When the client has no further requests, it sends a deregister message to the server returning back any unused credits.

For the rest of the section, we present how Breakwater detects overload and how it reacts to it. In particular, we present how a server determines the number of credits it can issue, how to distribute them among clients, and how clients react to credits or the lack thereof.

3.1 Overload Detection

There are multiple signals we can utilize to determine whether a server is congested. CPU load is a popular congestion signal—it is often used to make auto-scaling decisions in cloud computing [5]. However, CPU utilization indicates only one type of resource contention that can affect RPC latency. For instance, requests contending for a hard disk can have high latency, but CPU utilization will remain low [22]. Moreover, using CPU utilization as a signal does not allow an overload controller to differentiate between the ideal scenario of 100% utilization with no delayed RPCs and a livelock state.

Another potential congestion signal is queue length at the server. A similar signal is widely used in network congestion control [8, 52]. Unfortunately, when RPC service times have high dispersion, queue length is a poor indicator of request latency. A more reliable signal is queuing delay, as it is accurate even under RPC service time variability. Furthermore, it is intuitive to map a target SLO to a target queueing delay at the server. Thus, Breakwater uses queuing delay as its congestion signal.

Effective overload control requires accurate measurement of the queuing delay signal. In particular, the signal should account for the sum of each of the queueing delay stages a request experiences, ignoring non-overload induced delays. This ensures that the system only curbs incoming requests when it is overloaded. This is especially critical for microsecond-scale RPCs, as they leave little room for error.

Breakwater has two stages of queueing. Packets are queued while they await processing to create a request. Then, threads created to process requests are queued awaiting execution. Breakwater tracks and sums queuing delay at both of these stages. In particular, for every queue in the system, each item (e.g., a packet or a thread) is timestamped when it is enqueued. Each queue maintains the oldest timestamp of enqueued elements in a shared memory region, and this timestamp is updated on every dequeue. When the delay of a queue needs to be calculated, Breakwater computes it by taking the difference between the current time and the queue's oldest timestamp. We use this approach instead of measuring explicit delays of each request (i.e., the timestamp difference between request arrival and the request execution) because we must keep track of the total queueing delay as a request moves from one queueing stage to another.

There are multiple sources of delay that are not caused by high utilization or overload. For example, long delays due to head-of-line blocking do not indicate a thread is waiting for resources, but rather it is a sign of poor load balancing. Accurate queueing delay measurement requires the system to avoid such delays. We find that the biggest source of such delays is the threading model used by the system. Our initial approach for developing Breakwater relied on the in-line threading model [19, 25] where a single thread handles both packet processing and request processing. This choice was made as the in-line model provides the lowest CPU cost. However, it leads to head-of-line blocking as a single request with a large execution time can block other requests waiting at the same core. The alternative is relying on the dispatcher threading model [41] where a dispatcher thread processes packets and spawns a new thread for request processing incurring inter-thread communication overhead. However, this overhead is minimal when the dispatcher model is implemented using lightweight threads in recently proposed low-latency stacks (e.g., Shenango [40] and Arachne [43]). Thus, Breakwater employs the dispatcher model for request processing.

3.2 Overload Control

During overload, the system has to decide which requests to admit for processing and which requests to drop or possibly queue at the client. In this section, we explain our design for Breakwater's approach to overload control.

3.2.1 Server-driven Credit-based Admission Control

A Breakwater server controls the admission of incoming requests through a credit-based scheme. Server-driven admission control avoids the need for clients to probe the server to know what rate to send at. It also allows the server to receive the exact load it can handle. A credit represents availability at the server to process a single request by the client that receives the credit. A Breakwater server manages a global pool of credits (C_{total}) that is then distributed to individual clients. C_{total} represents the load the server can handle while maintaining its SLO. This is achieved by controlling C_{total} such that the measured queuing delay (d_m) remains close to a target queuing delay (d_t), which is set based on the SLO of the RPC.

Every network RTT, Breakwater updates C_{total} based on the measured queuing delay (d_m). If d_m is less than d_t, Breakwater increases C_{total} additively.

$$C_{total} \leftarrow C_{total} + \mathcal{A} \tag{1}$$

Otherwise, it decreases C_{total} multiplicatively, proportional to the level of overload.

$$C_{total} \leftarrow C_{total} \cdot \max(1.0 - \beta \cdot \frac{d_m - d_t}{d_t}, 0.5) \tag{2}$$

Note that $\mathcal{A}$ controls the overcommitment and aggressiveness of the generation of credits. On the other hand, β controls the sensitivity of Breakwater to queue build-up. We explain how we select $\mathcal{A}$ and β in the next section.

Once C_{total} is decided, credits are distributed to clients. When C_{total} increases, new credits are issued to clients by piggybacking the issued credits to response messages sent to the clients. Explicit *credit* messages are only generated when piggybacking is not possible (i.e., server has no messages bound for the client). When C_{total} decreases, the server does not issue additional credits to the clients, or if the clients have unused credits, the server sends negative credits to revoke the credits issued earlier. The server can tell how many unused credits each client has by keeping track of the number of credits issued and the number of requests received. In the following section, we explain how Breakwater decides which client should be issued credits.

3.2.2 Demand Speculation with Overcommitment

There is a tradeoff between accurate credit generation and messaging overhead. Choosing which client should receive a credit can be simply determined based on the demand at the client. This requires clients to inform the server whenever their number of pending requests changes. The server can then select which clients to send a credit to based on demand. This ensures that all issued credits are used, allowing the server to generate credits that accurately represent its capacity. However, as we scale the number of clients, the overhead of exchanging demand messages overwhelms the capacity of the server.

In our design of Breakwater, we choose to eliminate the messaging overhead completely. A client notifies the server of its demand only if the demand information can be piggybacked on a request (i.e., the client already has a credit and can send a request to the server). The server therefore does not have accurate information about clients with sporadic demand as they can't update the server as soon as their demand changes. Thus, Breakwater speculatively issues credits based on the latest demand information even though it may be stale. Speculative generation of credits means that some clients that receive credits will not be able to use them immediately. If credits are generated to exactly match capacity, the server may experience underutilization because some credits are left unused when they are issued to clients with no queued requests. To achieve high utilization, speculative demand estimation is coupled with credit overcommitment to ensure that enough clients receive credits to keep the server utilized.

Overcommitment is achieved by setting the $\mathcal{A}$ and β parameters of the admission control algorithm. In particular, we set $\mathcal{A}$ to be proportional to the number of clients (n_c).

$$\mathcal{A} = \max(\alpha \cdot n_c, 1) \tag{3}$$

where α controls the aggressiveness of the algorithm. Further, each client is allowed to have more credits than its latest demand. The number of overcommitted credits per client (C_{oc}) is based on the number of clients (n_c), the total number of credits in the credit pool (C_{total}), and the total number of credits presently issued to clients (C_{issued}).

$$C_{oc} = \max(\frac{C_{total} - C_{issued}}{n_c}, 1) \tag{4}$$

The server makes sure that each client does not have unused credits more than its (latest) demand plus C_{oc} by revoking already issued credits if necessary.

Further, Breakwater attempts to avoid generating explicit credit messages whenever possible. This means that a new credit will be given to a client to whom the server is about to send a response unless that client has reached the maximum number of credits it can receive. Explicit credit messages are only generated when piggybacking a credit on a response is not possible. In the current version of Breakwater, the client that receives an explicit credit message is selected randomly, but we expect the selection could be smarter with per-client statistics. For example, the server can choose a client based on its average request rate to increase the likelihood of the client using the credit immediately.

3.2.3 AQM

The drawback of credit overcommitment is that the server may occasionally receive a higher load than its capacity, leading to long queues. To ensure low tail latency at all times, Breakwater relies on delay-based AQM to drop requests if the queueing delay exceeds an SLO-derived threshold. In our results, we find that drops are rare because our credit-based admission control scheme avoids creating bursts. Drops can be further reduced with by setting a large SLO budget. In particular, a system administrator can set a large threshold for AQM to reduce the drop rate at the expense of having a looser SLO.

3.3 Breakwater Client

Breakwater allows a client to queue requests if it does not have a credit for it. Client-side queuing is critical in a server-driven system as the client has to wait for the server to admit a request before it can send it. However, if the client queue is too long, the request will experience high end-to-end latency. In Breakwater, in order to achieve high throughput and low end-to-end latency, we allow requests to expire at the client. The request expiration time is set based on its SLO.

When a client receives credits, it can immediately consume them if its queue length is equal to or larger than the number of credits it receives. Due to overcommitment, a client can receive credits which it cannot immediately consume (c_{unused}). When a client receives negative credits with decreased C_{total} at the server, the client decrements c_{unused}. However, if a client has already consumed all of its credits (i.e., $c_{unused} = 0$), no action is taken by the client.

4 Implementation

Breakwater requires a low-latency network stack in order to ensure accurate estimation of the queuing delay signal. This requires minimal variability in packet processing and no head-of-line-blocking between competing requests. We use Shenango [40], an operating system designed to provide low tail latency for µs-scale applications with fast core allocations, lightweight user-level threads, and an efficient network stack. Shenango achieves low latency by dedicating a busy-spinning core to reallocate cores between applications every 5 µs to achieve high utilization and minimize the latency of packets arriving into the server.

We implement Breakwater as an RPC library on top of the TCP transport layer. Breakwater handles TCP connection management, admission control with credits, and AQM at the RPC layer. Breakwater abstracts connections and provides a simple individual RPC-oriented interface to applications, leaving applications to only specify request processing logic. Breakwater provides a single RPC layer per application (i.e., overload signal, credit pool, etc.) regardless of the number of cores allocated to the application and the number of clients of that application. A request arriving at a Shenango server is first queued in a packet queue. Then a Shenango kernel thread processes packets and moves the payload to the socket memory buffer of the connection. Once all the payload of a request is prepared in the memory buffer, a thread in Breakwater parses the payload to a request and creates a thread to process it. Threads are queued pending execution, and when they execute, they execute to completion.

Threading model. As explained earlier, Breakwater relies on a dispatcher threading model for accurate queueing delay measurement. A Breakwater server has a listener thread and the admission controller thread running. When a new connection arrives, the listener thread spawns a receiver thread and a sender thread per connection. Receiver threads read incoming packets and parse them to create requests. After parsing a request, AQM is performed, dropping requests if the current queueing delay is greater than the AQM drop threshold. If a request is not dropped, the receiver thread spawns a new thread for the request. The new thread is enqueued to the thread queue. The sender thread is responsible for sending responses (either success or reject) back to the clients. If there are multiple responses, the sender thread coalesces them to reduce the messaging overhead. For all threads in Breakwater, we use lightweight threads provided by Shenango's runtime library.

Queueing Delay Measurement. With a separate receiver thread minimizing the delay from the socket memory buffer, the two main sources of queueing delay in Shenango are packet queueing delay (i.e., time between when a packet arrives till it is processed by a Shenango kernel thread) and thread queueing delay (i.e., time between when a thread is created to process a request until it starts executing). In Shenango, each core has a packet queue and a thread queue shared with IOKernel. We instrumented packet queues and thread queues so that each queue maintains the timestamp of the oldest item, and we modified Shenango's runtime library to export the queueing delay signal to the RPC layer. When Shenango's runtime is asked for the queueing delay, it returns the maximum of the packet queue's delays plus the maximum of the thread queue's delays.

Lazy credit distribution. The admission controller updates C_{total} every RTT. Once the credit pool size is updated, the admission controller can re-distribute credits to clients to achieve max-min fairness based on the latest demand information. However, this requires the admission controller to scan the demand information of all clients, requiring $O(N)$ steps. To reduce the credit distribution overhead, Breakwater approximates max-min fair allocation with lazy credit distribution. In particular, Breakwater delays determining the number of credits a client can receive until it has a response to send to that client. The sender thread, responsible for sending responses to a client, decides whether to issue new credits, not to issue any credits, or to revoke credits based on C_{issued}, C_{total}, and the latest demand information. It first calculates the total number of credits the server should grant to client x (c_x^{new}). If C_{issued} is less than C_{total}, c_x^{new} becomes

$$c_x^{new} = \min(demand_x + C_{oc}, c_x + C_{avail}) \qquad (5)$$

where $demand_x$ is the latest demand of client x, c_x is the number of unused credits already issued to client x and C_{avail} is the number of available credits the server can issue ($C_{avail} = C_{total} - C_{issued}$). If C_{issued} is greater than C_{total}, c_x^{new} becomes

$$c_x^{new} = \min(demand_x + C_{oc}, c_x - 1) \qquad (6)$$

The sender thread then piggybacks the number of credits newly issued for client x ($c_x^{new} - c_x$) to the response. It also updates c_x to c_x^{new} and C_{issued} accordingly.

5 Evaluation

Our evaluation answers the following questions:

- Does Breakwater achieve the objectives of overload control defined in §2 even given tight SLOs?
- Can Breakwater maintain its advantages regardless of load characteristics (i.e., average RPC service time and service time distribution)?
- Can Breakwater scale to large numbers of clients?
- Can Breakwater react quickly to a sudden load shift?
- What is the impact of Breakwater's key design decisions: demand speculation and credit overcommitment?
- How sensitive is Breakwater's performance to different parameters?

5.1 Evaluation Setup

Testbed: We use 11 nodes from the Cloudlab xl170 cluster [20]. Each node has a ten-core (20 hyper-thread) Intel E5-2640v4 2.4 GHz CPU, 64 GB ECC RAM, and a Mellanox ConnectX-4 25 Gbps NIC. Nodes are connected through a Mellanox 2410 25 Gbps switch. The RTT between any two nodes is 10 µs. We use one node as the server and ten nodes as clients. The server application uses up to 10 hyper-threads (5 physical cores) for processing requests, and the client application uses up to 16 hyper-threads (8 physical cores) to generate load. All nodes dedicate a hyper-thread pair for Shenango's IOKernel.

Baseline. We compare Breakwater to DAGOR [51] and SEDA [48]. DAGOR is a priority-based overload control system used for WeChat microservices. Priorities are assigned based on business requirements across applications and at random across clients. We only consider a single application in our evaluation. DAGOR uses queueing delay to adjust the priority threshold at which a server drops incoming requests (i.e., requests with a priority lower than the threshold are dropped). To reduce the overhead of dropped requests, the server advertises its current threshold to clients, piggybacked it in responses. Clients use that threshold to drop the requests. Note that DAGOR does not drop its threshold to zero, meaning that a request with the highest priority value (i.e., a priority of one) will never be dropped. SEDA uses a rate-based rate limiting algorithm. It sets rates based on the 90%-ile response time. Since we evaluate the performance of Breakwater using the 99%-ile latency metric, we modified SEDA's algorithm so that it adjusts rates based on 99%-ile response time. We implement DAGOR and SEDA as an RPC layer in Shenango with the same dispatcher model as Breakwater.

Setting end-to-end SLO. We set tight SLOs to support low-latency RPC applications. We budget SLOs based on the server-side request processing time and the network RTT. An SLO is set as 10× the sum of the average RPC service time measured at the server and the network RTT; the multiplicative factor of 10 was inspired by recent work on µs-scale RPC work [17, 42]. The RTT in our setting is 10 µs, leading to SLOs of 110 µs, 200 µs, and 1.1 ms for workloads with 1 µs, 10 µs, and 100 µs average service times, respectively. These are comparable with SLO values used in practice [30].

Evaluation metrics: We report goodput, 99%-ile latency, drop rate, and reject message delay. Goodput represents the number of requests processed per second that meet their SLO. Reported latency captures all delays faced by a request from the moment it is issued till its response is received by the client. This includes any queuing delay at the client, communication delay, and all delays at the server. We report drop rate at the server only, as it is the factor the directly impact overall system performance. *Note that SEDA does not support any AQM at the server and has zero drop rate in all experiments.* Reject message delay represents the delay between the departure of a request from a client and the arrival of a reject message back to the client when that request is dropped at the server.

Parameter tuning. We tune the parameters of all systems so that they achieve the highest possible goodput. We re-tune the parameters when we change the average service time, service time distribution, and the number of clients. Note that Breakwater and DAGOR do not require parameter re-tuning for a different number of clients while SEDA does. Specifically, we need to scale adj_i parameter in SEDA based on the number of clients to get the best goodput. For Breakwater, we set $\alpha = 0.1\%$, $\beta = 2\%$, d_t to 40% of SLO, and AQM threshold to $2 \cdot d_t$ (e.g., $d_t = 80\,\mu s$ and AQM threshold $= 160\,\mu s$ for exponential service time distribution with 10 µs average and 200 µs SLO). For DAGOR and SEDA, which are devised for ms-scale RPCs, we scale down the hyperparameters from the default values. For DAGOR, we update the priority threshold every 1 ms (instead of 1 s) or every 2,000 requests and use $\alpha = 5\%$ and $\beta = 1\%$. We assign random priority for each request ranging from 1 to 128, which is the default priority setting with one type of service in DAGOR [51]. We tune $DAGOR_q$ for each workload (e.g., $DAGOR_q = 70\,\mu s$ for exponential service time distribution with 10 µs on average). For SEDA, we used the same default parameter from [48] except for $timeout$, adj_i, and adj_d. We set $timeout = 1\,ms$ (instead of 1 s) and tune adj_i and adj_j for each workload (e.g., $adj_i = 40$, $adj_d = 1.04$ for exponential workload with 10 µs average with 1,000 clients). AQM in Breakwater and DAGOR drops requests right after parsing packets to requests, following the drop-as-early-as-possible principle [33]. We run all the experiments for four seconds. We measure steady state performance with converged adaptive parameters by collecting data two seconds after an experiment starts.

5.2 Performance for Synthetic Workload

Workload: We run 1,000 clients divided equally between the ten nodes in our CloudLab setup. We generate the workload with exponential, constant, and bimodal service time distributions with 1 µs, 10 µs, and 100 µs average where each client generates the load with an open-loop Poisson process. We change the demand by varying the average arrival rate of requests at the server between 0.1× to 2× of server capac-

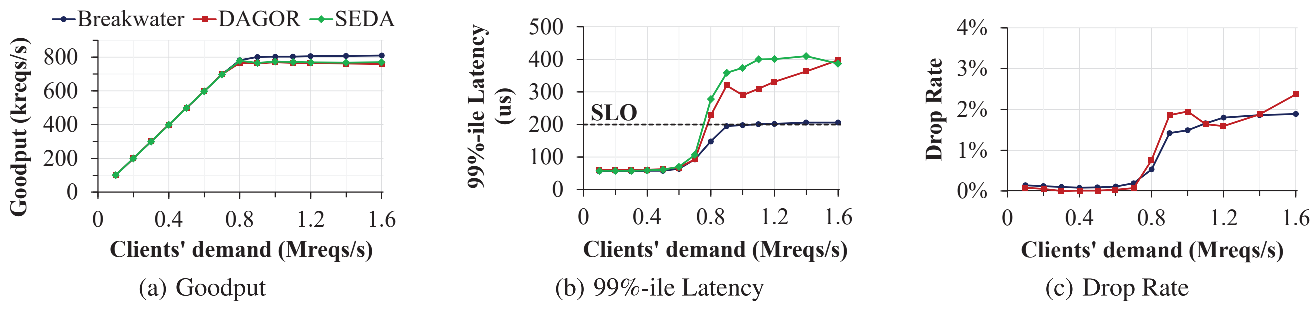


Figure 3: Performance of Breakwater, DAGOR, and SEDA for synthetic workloads with the exponential distribution of 10 µs average

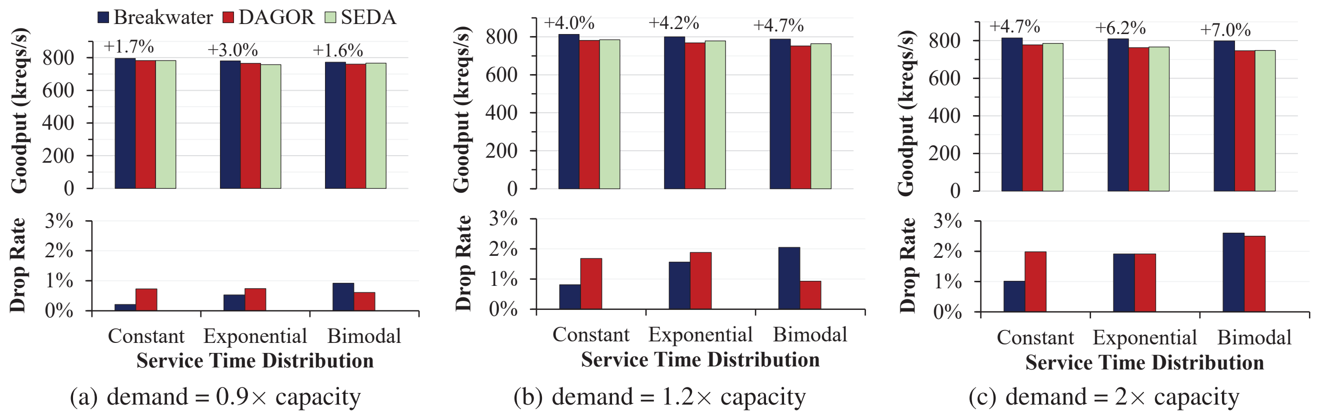


Figure 4: Goodput and drop rate with different service time distribution of 10 µs average with 1,000 clients (The label represents the goodput gain compared to the worst of baselines.)

ity. Exponential service time distribution models applications waiting for a shared resource while busy-spinning; constant distribution models applications with a fixed amount of latency such as fetching value from memory or flash drive; bimodal distribution models applications that caches frequently requested values, which will have shorter execution time compared non-cached results. In particular, 20% of the requests take four times the average service time, and 80% of the requests take one fourth of the average following the Pareto principle.

Overall performance: Figure 3 shows the performance for a workload whose service time follows an exponential distribution with 10 µs average. The capacity of the server in this case is around 850k requests per second.

When the clients' demand is less than the capacity, all three systems perform comparably in terms of goodput, latency, and drop rate. The only noticeable difference among them is that, at 700k reqs/s, SEDA has a 15% higher 99%-ile latency than Breakwater or DAGOR. This is because SEDA doesn't drop requests at servers.

When the clients' demand is around the capacity of the server, Breakwater achieves 801k requests per second for goodput (or 808k reqs/s of throughput), which is around 5% overhead when compared to the maximum throughput with no overload control. Other systems have higher overhead than Breakwater.

When the demand exceeds the capacity, incast becomes the dominant factor impacting performance. Breakwater handles incast well by preventing clients from sending requests unless they have credits, limiting the maximum queue size. Thus, Breakwater achieves higher goodput with lower and bounded tail latency. On the other hand, SEDA experiences high tail latency because clients do not coordinate their rate increase, making multiple clients increase their rate simultaneously and overwhelm the server. Delayed reaction to overload does not allow SEDA to react quickly to incast. DAGOR's high tail latency is also explained by delayed reaction as it updates its priority threshold every 1 ms or every 2,000 requests. Breakwater is also impacted by incast due to the overcommitted credits, which lead to increased tail latency and higher drop rate with overload. However, Breakwater relies on delay-based AQM which effectively bounds the tail latency while maintaining a comparable drop rate to DAGOR.

Impact of Workload Characteristics: To verify that Breakwater's performance benefits are not confined to a specific workload, we repeat the experiments with different service time distributions and different average service time values. Figure 4 shows goodput and drop rate with three different distributions of the service time whose average is 10 µs, where the load generated by 1,000 clients is 0.9×capacity, 1.2× capacity, and 2× capacity. The service time distributions are aligned over the x-axis in ascending order of variance. Break-

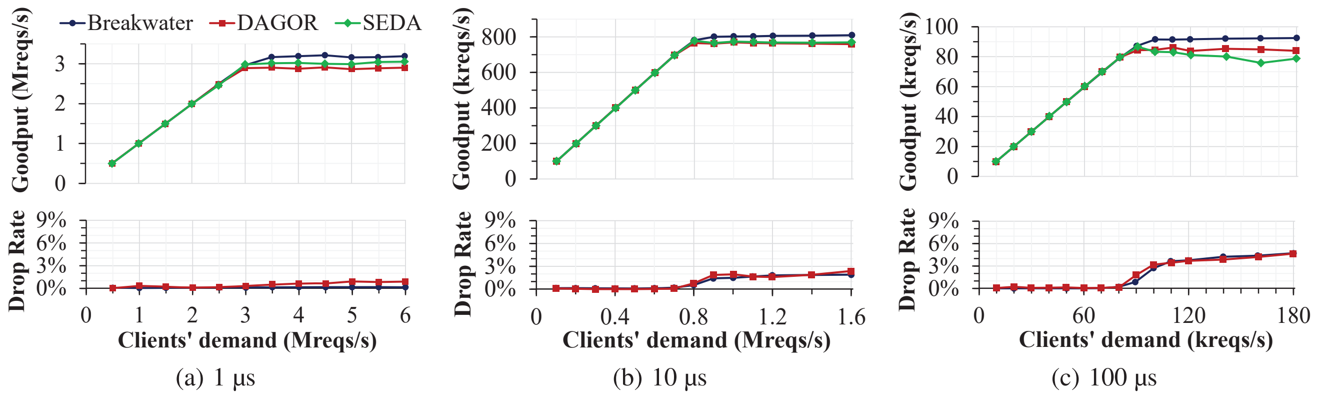


Figure 5: Goodput and drop rate with different average service time with 1,000 clients

water achieves the highest goodput regardless of the load and service time distribution. All three systems experience small goodput reduction with a higher variance, especially when the load is 2× the server capacity. The goodput reduction of DAGOR and SEDA comes from their poor reaction to incast, whose size increases as the load increases. As a result, Breakwater's goodput benefit becomes larger as the clients' demand increases. Breakwater achieves 5.7% more goodput compared to SEDA and 6.2% more goodput compared to DAGOR with exponential distribution at a load of 2× capacity. With a higher variance of the service time distribution, the drop rate of the Breakwater tends to increase because a larger number of credits are overcommitted with higher variance, but it is still comparable to DAGOR.

Figure 5 depicts performance with an exponential service time distribution and different average service times with 1,000 clients. Breakwater outperforms DAGOR and SEDA regardless of the clients' demand and the average service time. As the average service time increases, clients and servers exchange messages less frequently, exposing the delayed reaction problem in SEDA and DAGOR. With short service times (i.e., 1 µs), clients and servers exchange messages very frequently, giving clients a fresh view of the state of the server in case of DAGOR and SEDA, allowing clients to react quickly to overload. With high demand, the size of incast gets larger which is poorly handled by SEDA and DAGOR. With clients' demand of 2× capacity with 100 µs (i.e., 180k reqs/s), Breakwater achieves 17.5% more goodput than SEDA and 10.2% more goodput with a comparable drop rate compared to DAGOR.

Scalability to a Large Number of Clients: We vary the number of clients from 100 to 10,000 with synthetic workload whose service time follows exponential service time distribution of 10 µs average. Note that the server capacity is around 850k requests per second. Figure 6 depicts the goodput with different numbers of clients. As clients' demand nears and exceeds the capacity, the goodput of all systems degrades as the number of clients increases. As the number of clients increases, the size of incast increases, leading to

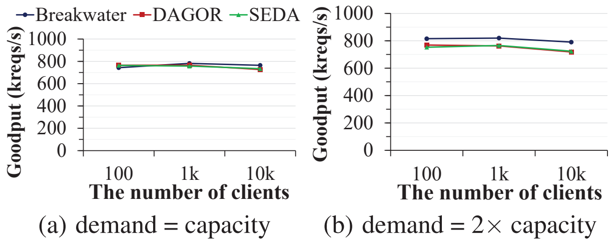


Figure 6: Goodput with different numbers of clients for exponential workload with 10 µs average service time

performance degradation. This is problematic for Breakwater as well since overcommitment can occasionally result in large bursts of incoming requests. The performance of DAGOR and SEDA drops more than Breakwater as the number of clients increases. This is because each client exchanges messages with the server less frequently as the number of clients increases. The stale view of the server status leads clients to overwhelm the server. Note that for SEDA's best performance, we scale the additive rate increase factor (adj_i) to the number of clients. This helps mitigate any bursty behavior that can result from multiple clients sharply increasing their rate simultaneously. A small increase factor is not practical for a small number of clients as it will lead to slow ramp-up of rates after an overload, leading to lower utilization of the server. Because of this issue, SEDA has a much slower convergence time to the right rate, making it impractical for load shift scenarios as we show next.

Further, it is hard to tune SEDA dynamically. The rate control algorithm in SEDA is implemented at the client, and dynamic tuning requires each client to know the total number of *active* clients. Such a dynamic approach will lead to performance degradation as the client will retune its parameter to at least an RTT after the number of clients changes. The drawbacks of such a delayed reaction can be seen in the behavior of DAGOR. Further, exchanging such information might not feasible in practice due to messaging overhead as well as privacy concerns (e.g., a FaaS cloud provider will not want any of its clients to know the total number of clients). Note that even though Breakwater also scales the number of newly issued credits to the number of clients (Equation 1 and 3), Breakwater is server-driven, and the server has perfect knowledge of the number of active clients at all times with no need to expose this information outside. In SEDA, by contrast, each client cannot have perfect knowledge of the number of active clients. Each client would have to guess or receive feedback from the server to scale the increment factor.

Reaction to Sudden Shifts in Demand: An RPC server may experience sudden shifts in demand for many reasons, such as load imbalance, packet bursts, unexpected user traffic, or redirected traffic due to server failure. To verify Breakwater's ability to converge after a shift in demand, we measure its performance with a shifting load pattern. We use a workload whose service time follows an exponential distribution with 10 µs average and calculate goodput, 99%-ile latency, and mean reject message delay every 20 ms. When the experiment starts, 1,000 clients generate requests at 400k reqs/s (0.5× capacity). Then, clients double their request rate to 800k reqs/s (0.9× capacity) at time = 2 s, then triple their demand to 1.2M reqs/s (1.4× capacity) at time = 4 s. Clients sustain their demand at 1.2M reqs/s for 2 seconds. Then, clients reduce their demand back to 800k reqs/s at time = 6 s and finally to 400k reqs/s at time = 8 s. Figure 7 depicts a time series behavior of all systems.

When the clients' demand is far less than the capacity, all three overload control schemes maintain comparable goodput and tail latency at a steady state. When demand increases to near server capacity, Breakwater converges fast, exhibiting a stable behavior in terms of both goodput and tail latency. On the other hand, DAGOR and SEDA experience higher tail latency because of the poor reaction to the transient server overload. As the server becomes persistently overloaded with a sudden spike at time = 4 s, Breakwater converges quickly while DAGOR and SEDA suffer from congestion collapse. Breakwater experiences a momentary tail latency increase (reaching 1.4× the SLO) with the sudden increase of clients' demand due to more incast caused by overcommitted credits. However, credit revocation and AQM rapidly limit the impact of any further incast. When demand returns back below the capacity at time = 6 s, Breakwater doesn't show a noticeable goodput drop while the DAGOR and SEDA experience a temporary goodput drop down to 77.5% and 82.6% of the converged goodput, respectively.

SEDA reacts slowly to the demand spike since each client needs to wait for a hundred responses or 1 ms to adjust its rate. After the demand spikes beyond the capacity, the server builds up long queues, and the latency goes up beyond SLO, resulting in almost zero goodput. SEDA takes around 1.6 s to recover its goodput. DAGOR also has the delayed reaction problem, but its goodput converges more quickly than SEDA thanks to AQM, taking 500 ms to recover its goodput. During the congestion collapse period, the 99%-ile latency of DAGOR soars up to 300 ms and its mean delay of reject message reaches 220 ms. This is problematic as clients cannot receive the feedback in a timely manner, making them rely on expensive timeout.

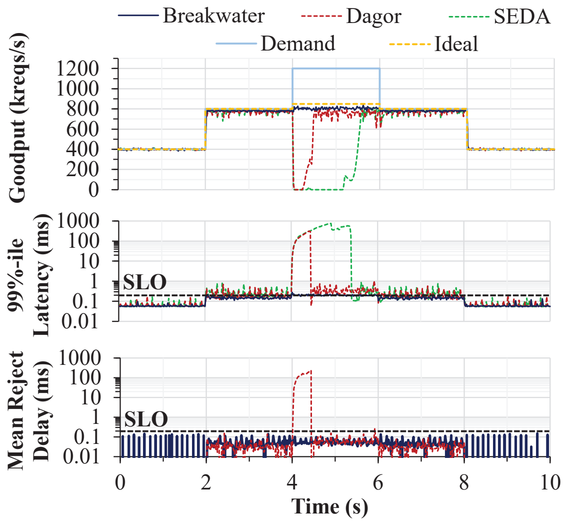


Figure 7: Goodput, 99%-ile latency, and mean rejection delay with a sudden shift in demand with 1,000 clients

The Value of Demand Speculation: To quantify the performance benefits of demand speculation, we compare the two strategies for collecting demand information: demand synchronization and demand speculation. With demand synchronization, clients notify the server whenever their demand changes using explicit demand messages, and the server generates explicit credit messages to clients if it cannot be piggybacked to responses. With demand speculation, the server speculatively estimates client demands based on the latest demand information and piggybacks credits to the responses as much as possible. The load is generated by 1,000 clients where the service time per request follows an exponential distribution with an average of 10 µs. The message overhead is measured by the number of packets received (RX) and sent (TX) at the server. With demand synchronization, both RX and TX message overhead increase as the clients' demand increases, leading to goodput degradation (Figure 8 (a)). In particular, explicit demand and credit messages doubles RX and TX message overhead below and at the capacity (i.e., 850k requests per second). As the system gets overloaded, the overhead of demand messages keeps increasing because per-client demand changes more frequently with increased clients' demand. Further, the overhead of generating credits contributes to the cost of synchronization. The server sends more credit messages during low demand as they cannot be piggybacked on responses due to low request rates. As load increases beyond capacity, more credits can be piggybacked to the responses, which results in the reduction of TX overhead. Demand synchronization has a smaller number of overcommitted credits, leading to a lower drop rate than demand speculation (Figure 8 (c)). Overall, the cost of synchronization between the clients and the server is high in terms of goodput degradation and network overhead, with the small

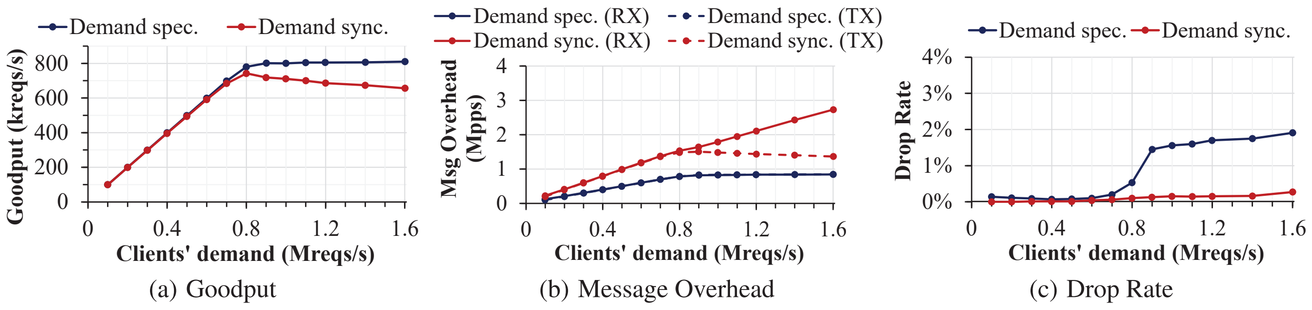


Figure 8: Goodput, message overhead, and drop rate with demand speculation and demand synchronization in Breakwater

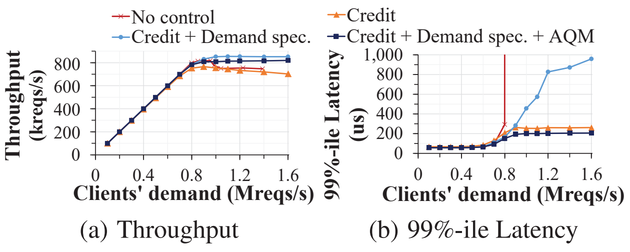


Figure 9: Breakwater performance breakdown

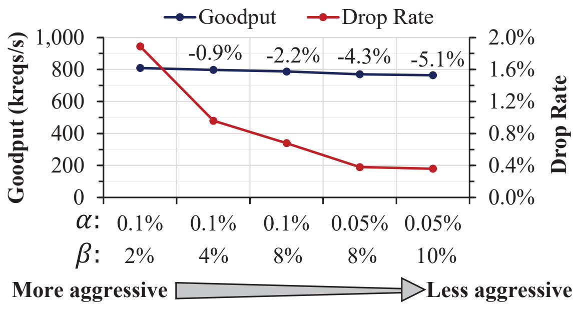


Figure 10: Goodput and drop rate with different aggressive parameters of Breakwater

benefit of lowering the drop rate at the server.

Performance Breakdown: To quantify the contribution of each component of Breakwater to its overall performance, we measure the throughput and 99%-ile latency after incrementally activating its three major components: credit-based admission control, demand speculation, and delay-based AQM. The results are shown in Figure 9. We use the synthetic workload whose service time is exponentially distributed with 10 µs average (SLO = 200 µs). With no overload control at all, throughput starts to degrade, and tail latency soars, making almost all requests violate their SLO as demand becomes higher than server capacity. Credit-based admission control effectively lowers and bounds the tail latency, but throughput still suffers due to the messaging overhead. Demand speculation with message piggybacking reduces the messaging overhead, but it worsens tail latency due to incast caused by credit overcommitment. By employing delay-based AQM, Breakwater effectively handles incast, leading to high throughput and low tail latency.

Parameter Sensitivity: Breakwater parameters are set aggressively to maximize the goodput, resulting in a relatively high drop rate. With less aggressive parameters, Breakwater can drop fewer requests sacrificing goodput. Figure 10 demonstrates the trade-off between the goodput and the drop rate for the workload with exponential service time distribution with 10 µs average with 1M reqs/s demand from 1,000 clients. The values of pairs of α and β are aligned in descending order of aggressiveness over the x-axis. Breakwater achieves 0.7% of drop rate by sacrificing 2.2% of goodput (with $\alpha = 0.1\%$, $\beta = 8\%$) and 0.4% of drop rate by sacrificing 5.1% of goodput (with $\alpha = 0.05\%$, $\beta = 10\%$).

In practice, it is not easy to find the best parameter configuration for an operational system. It is even more difficult when traffic patterns change over time because parameter adjustments could be required to achieve the best possible performance. Thus, it is desirable to develop systems that are robust to parameter misconfiguration and changes in traffic patterns, providing consistently good performance even with small errors in parameter settings. Breakwater is robust. In particular, it provides high throughput and low tail latency despite parameter misconfiguration. We compare it against DAGOR and SEDA, measuring their performance for the same workload while varying their parameters. Specifically, we measure the throughput and 99%-ile latency after reconfiguring the three most sensitive parameters for each system: target delay, increment factor, and decrement factor (d_t, α, β for Breakwater; threshold of the average queueing time, α, β for DAGOR; and $target, adj_i, adj_d$ for SEDA). Given the set of parameters producing best goodput, we measure 27 data points with -10, 0, +10 µs of target queueing delay, 0.5×, 1×, 2× of the increment factor, and 0.5×, 1×, 2× of the decrement factor. We use a synthetic workload with exponentially distributed service times with 10 µs average with 1,000 clients. The results are shown in Figure 11 where the circles filled with light color indicate the performance with the parameters tuned for the best goodput. All configurations of Breakwater achieve comparable performance in terms of both throughput and tail latency, achieving better throughput and latency trace-offs and more consistent performance with different sets of parameters. DAGOR tends to provide high throughput, but its tail latency is as high as four times the SLO in the worst case. SEDA's worst case tail latency is lower than DAGOR,

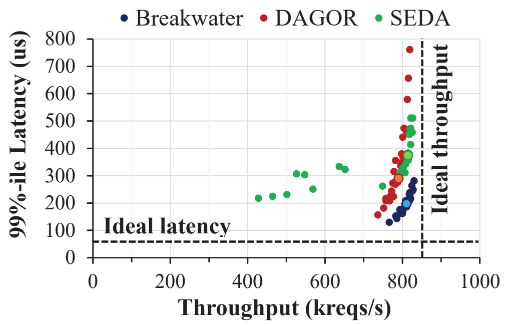


Figure 11: Throughput and 99%-ile latency trade-off with different sets of parameters (circle with light color indicates the point producing best goodput)

but it suffers from severe throughput degradation when its parameters are too conservative.

5.3 Performance under Realistic Workload

To evaluate Breakwater in a more realistic scenario, we create a scenario where one memcached instance serves 10,000 clients. We use the USR workload from [9] where 99.8% of the requests are GET, and other 0.2% are SET. Each client generates the load according to an open-loop Poisson process. We set an SLO of 50 µs considering that the latency of GET operation of memcached is less than 1 µs. Figure 12 shows goodput, median latency, 99%-ile latency, and drop rate of Breakwater, DAGOR, and SEDA. Breakwater achieves steady goodput, low latency, and low drop rate, whereas both DAGOR and SEDA suffer from goodput degradation with high tail latency caused by incast when the server becomes overloaded. With clients' demand of 2× capacity, Breakwater achieves 5% more goodput and 1.8× lower 99%-ile latency than SEDA; and 14.3% more goodput and 2.9× lower 99%-ile latency than DAGOR. Because of bimodally distributed service time with a mix of GET and SET requests, Breakwater shows around 25 µs higher 99%-ile latency than its SLO and about 1.5% point higher drop rate than DAGOR.

6 Discussion and Future Work

Auto-scaling. We do not consider auto-scaling [5, 23, 31] in this paper, where more resources are provisioned as load increases, as a potential solution for overload control. Auto-scaling can allocate enough capacity over time, but because it operates at the timescale of minutes, it is too slow to resolve microsecond-scale imbalances. Furthermore, over-provisioning resources can be cost-inefficient if used to handle transient spikes in demand, such as those that occur during temporary failures [3].

Fairness. When the server has a sufficient number of credits, it tries to approximate max-min fairness when distributing credits to clients. However, when the number of available credits is less than the number of clients, Breakwater does not provide fairness to clients. Instead, it favors clients for which it is currently processing requests. This allows the server to piggyback credits to the responses and avoid sending explicit

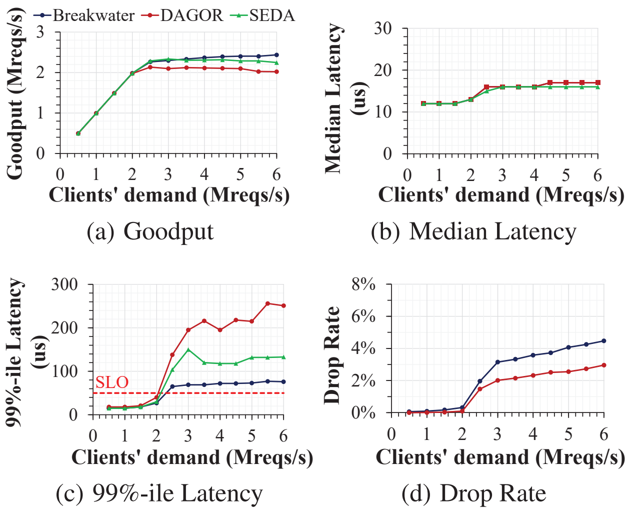


Figure 12: Memcached performance for USR workload with 10,000 clients (SLO = 50 µs)

credit messages. This preference toward a subset of clients is common in production services [51]. If a service operator wants to provide fairness among clients, the clients receiving the most credits could be timed-out over a longer timescale, so clients starved of credits can get a chance to send instead.

Overload control for multi-layer services. In this paper, we only consider a single-layer, single-server overload control scenario. Breakwater's receiver-driven, credit-based approach can be applied to multiple layers of microservices, preventing overload at each individual layer. However, when an overload occurs in an intermediate layer of a multi-layer service, the work performed in earlier layers is wasted. We leave propagating overload signals and coordinating overload control across several layers of microservices for future work.

7 Related Work

Receiver-driven transport protocols. Homa [34], NDP [24], and ExpressPass [14] schedule network packets with a receiver-driven mechanism to achieve high throughput and low latency. While Homa and Breakwater share some similarities including a credit-based, receiver-driven scheme and credit overcommitment, they are different in three significant aspects. First, Homa handles network congestion, whereas Breakwater handles server overload, which means that Breakwater must handle the additional challenges posed by overload control discussed in §2.3. Second, Homa relies on full knowledge of clients' demand, whereas Breakwater does not. Instead, the Breakwater server speculates clients' demand based on the latest demand information, the number of clients, and the number of available credits to minimize the message overhead. Third, both the motivation and the mechanism of overcommitment are different. Homa overcommits a fixed number of credits to handle an all-to-all workload, where a sender may get credits from multiple receivers and therefore not be able to send to all of them simultaneously. In Breakwater, however, the server does not know which clients have demand. Thus, it dynamically increases the amount of overcommitted credits until it receives sufficient requests to keep itself busy with demand speculation.

Transport protocol for µs-scale RPCs. R2P2 [28] is a request/response-aware transport protocol designed for µs-scale RPCs. It implements JBSQ inside a programmable switch to better load balance requests among multiple servers. R2P2 limits the number of requests in a server's queue by explicitly pulling the requests from the switch. Through this mechanism, R2P2 provides bounded request queueing and low tail latency when the clients' demand is less than the servers' capacity. However, R2P2 does not provide any server overload control mechanism. If the clients' demand exceeds the servers' capacity, the request queue will build up at the switch, causing requests to violate their SLO. SVEN [27] builds upon R2P2 by adding a server overload control mechanism. Specifically, it drops requests at the switch if sampled tail latency exceeds an SLO-derived threshold. SVEN avoids the cost of request drops at the server by dropping requests early at the switch. However, unlike Breakwater, message overhead increases as clients' demand increases.

Circuit breaker in proxy. Envoy [6], HAProxy [7], NGINX [44], and GateKeeper [21] provide circuit breaker mechanisms to prevent back-end server overload. These proxies sit in front of a back-end server and stop forwarding requests to the server when one of the load metrics (e.g., the number of connections, the number of outstanding requests, the response time, estimated load) exceeds a threshold. However, since those thresholds must be set manually, it's challenging to find the right threshold value that maximizes resource utilization while keeping latency low.

Server overload control. Session-based admission control [12, 13] prevents web server overloads by limiting the creation of new sessions based on the number of successfully completed sessions or QoS metrics. However, they are not compatible with request-response models as they cannot prevent server overloads caused by a single session from a proxy that forwards requests from multiple clients. CoDel [38] controls the queuing delay of a server to prevent server overloads. Still, if the incoming packet rate is high and CPU is used more for packet processing, the server becomes less CPU efficient and degrades throughput. ORCA [29], SEDA [48], and Doorman [4] rate limit clients so that their sending rates do not exceed the server capacity. Doorman requires manually setting of the server capacity threshold. Both ORCA and SEDA may suffer from long queueing delays or underutilization if clients make mistakes on their sending rate with stale congestion information from the server. DAGOR [51] takes a hybrid approach using both AQM and client-side rate limiting using adaptive parameter based on queueing delay. However, as DAGOR server updates congestion status with responses, clients still can undershoot or overshoot the server capacity with stale information on server congestion when client demand is sporadic.

Flow control. TCP flow control prevents the sender from transmitting more bytes than the receiver can accommodate. The objective of TCP flow control is to avoid memory overrun at the server, not to prevent server overload or SLO violations. More recently, an SLO-aware TCP flow control mechanism [26] was proposed where the server adjusts receive window size in TCP header based on SLO and the queueing delay at the server. This approach limits the "bytes" of the incoming requests to prevent server overload, but it's challenging to decide the appropriate receive window size, especially when the request size is variable.

8 Conclusion

In this paper, we presented Breakwater, a server-driven, credit-based overload control system for microsecond-scale RPCs. Breakwater achieves high throughput and low latency regardless of the RPC service time, the load at the server, and the number of clients generating the load. Breakwater generates credits based on queueing delay at the server, maintaining high utilization by targeting non-zero queueing delay while avoiding queue buildup. To minimize the overhead of coordination between the clients and the server, we propose demand speculation and credit overcommitment to realize the credit-based design for overload control with minimal overhead. By estimating clients' demand and issuing more credits than their capacity, Breakwater eliminates the extra messaging cost which is often required with a credit-based approach. Additionally, Breakwater reduces its remaining messaging overhead significantly by piggybacking demands and credits to requests and responses, respectively. Our evaluation of Breakwater shows that it outperforms state-of-the-art overload control systems. In particular, Breakwater achieves 25× faster convergence with 6% higher converged goodput than DAGOR and 79× faster convergence with 3% higher converged goodput than SEDA when the clients' demand suddenly spikes to 1.4× capacity.

Acknowledgments

We thank our shepherd Rachit Agarwal and the anonymous reviewers for their valuable feedback, and Cloudlab [20] for providing us with infrastructure for development and evaluation. We also thank the anonymous artifact evaluators for verifying our artifacts. This work was supported by the Cisco Research Center Award, NSF grants (CNS-1563826, CNS-1751009, and CNS-1910676), a Facebook Research Award, a Microsoft Faculty Fellowship, and a VMWare Systems Research Award.

References

[1] High-performance, feature-rich netxtreme® e-series dual-port 100g pcie ethernet nic. https://www.broadcom.com/products/

ethernet-connectivity/network-adapters/100gb-nic-ocp/p2100g.

[2] Memcached. http://memcached.org/.

[3] More on today's gmail issue, 2009. https://gmail.googleblog.com/2009/09/more-on-todays-gmail-issue.html.

[4] Doorman: Global distributed client side rate limiting., 2016. https://github.com/youtube/doorman.

[5] AWS Auto Scaling, 2020. https://aws.amazon.com/autoscaling/.

[6] Envoy Proxy, 2020. https://www.envoyproxy.io/.

[7] HAProxy, 2020. http://www.haproxy.org/.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (DCTCP). In *SIGCOMM*, 2010.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

[12] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *INFOCOM*, 2002.

[13] L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[14] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM*, 2017.

[15] J. Cloud. Decomposing twitter: Adventures in service-oriented architecture. In *QCon New York*, 2013.

[16] A. Cockroft. Microservices workshop: Why, what, and how to get there. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[17] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: Ni-driven tail-aware balancing of μs-scale rpcs. In *ASPLOS*, 2019.

[18] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[19] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *NSDI*, 2014.

[20] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of cloudlab. In *ATC*, 2019.

[21] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *International conference on World Wide Web*, 2004.

[22] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, 2019.

[23] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *ICAC*, 2014.

[24] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, 2017.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[26] M. Kogias and E. Bugnion. Flow control for latency-critical rpcs. In *KBNets*, 2018.

[27] M. Kogias and E. Bugnion. Tail-tolerance as a systems principle not a metric. In *APNet*, 2020.

[28] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *ATC*, 2019.

[29] B. C. Kuszmaul, M. Frigo, J. M. Paluska, and A. S. Sandler. Everyone loves file: File storage service (FSS) in oracle cloud infrastructure. In *ATC*, 2019.

[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[31] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *GridCom*, 2010.

[32] B. Maurer. Fail at scale. *Queue*, 2015.

[33] J. C. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 1997.

[34] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[35] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In *NSDI*, 2020.

[36] D. Namiot and M. Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2014.

[37] NGINX Documentation: Limiting Access to Proxied HTTP Resources, 2020. https://docs.nginx.com/nginx/admin-guide/security-controls/controlling-access-proxied-http.

[38] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 2012.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, 2013.

[40] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[41] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud storage system. *ACM Transactions on Computer Systems*, 2015.

[42] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[43] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *OSDI*, 2018.

[44] W. Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008.

[45] A. Sriraman and T. F. Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *OSDI*, 2018.

[46] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu. Distributed resource management across process boundaries. In *SoCC*, 2017.

[47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[48] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *SIGOPS European Workshop*, 2002.

[49] M. Welsh and D. E. Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[50] T. Zhang, J. Wang, J. Huang, J. Chen, Y. Pan, and G. Min. Tuning the aggressive tcp behavior for highly concurrent http connections in intra-datacenter. *Transactions on Networking*, 2017.

[51] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang. Overload control for scaling wechat microservices. In *SoCC*, 2018.

[52] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM*, 2015.