Protego: Overload Control for Applications with Unpredictable Lock Contention

Inho Cho, *MIT CSAIL;* Ahmed Saeed, *Georgia Tech;* Seo Jin Park,
Mohammad Alizadeh, and Adam Belay, *MIT CSAIL*

https://www.usenix.org/conference/nsdi23/presentation/cho-inho

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Protego: Overload Control for Applications with Unpredictable Lock Contention

Inho Cho[1] Ahmed Saeed[2] Seo Jin Park[1] Mohammad Alizadeh[1] Adam Belay[1]
[1]MIT CSAIL [2]Georgia Tech

Abstract

Modern datacenter applications are concurrent, so they require synchronization to control access to shared data. Requests can contend for different combinations of locks, depending on application and request state. In this paper, we show that locks, especially blocking synchronization, can squander throughput and harm tail latency, even when the CPU is underutilized. Moreover, the presence of a large number of contention points, and the unpredictability in knowing which locks a request will require, make it difficult to prevent contention through overload control using traditional signals such as queueing delay and CPU utilization.

We present Protego, a system that resolves these problems with two key ideas. First, it contributes a new admission control strategy that prevents compute congestion in the presence of lock contention. The key idea is to use marginal improvements in observed throughput, rather than CPU load or latency measurements, within a credit-based admission control algorithm that regulates the rate of incoming requests to a server. Second, it introduces a new latency-aware synchronization abstraction called Active Synchronization Queue Management (ASQM) that allows applications to abort requests if delays exceed latency objectives. We apply Protego to two real-world applications, Lucene and Memcached, and show that it achieves up to 3.3× more goodput and 12.2× lower 99th percentile latency than the state-of-the-art overload control systems while avoiding congestion collapse.

1 Introduction

One of the key objectives of datacenter operators is to maximize the utilization of limited resources. While operating a server close to its capacity maximizes its throughput, it also makes it susceptible to overload due to surges in demand. Such surges can occur due to variability in request arrival patterns and sizes, and service failures. The resulting server overload can cause *receive livelock*, 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 [22].

The conventional solution is to use *overload control* to regulate incoming requests and shed excess load, ensuring that the server can achieve both high utilization and low latency. Existing overload control schemes focus on CPU overload [9, 34] or end-to-end response time [32]. However, we found these approaches perform poorly under lock contention, especially with blocking synchronization (e.g., mutexes) that causes a thread to yield rather than spinning on the CPU (§2). For these cases, contention leads to long queues of requests waiting to acquire a critical section, increasing tail latency and wasting CPU resources.

To better understand the challenge of managing lock contention, consider a key-value store, where the key-value pairs are grouped together based on the hashes of their keys. Access to a bucket (i.e., a group of items with the same hash) is protected by an item lock. This means that in a key-value store, the number of locks corresponds to the number of buckets. However, a `GET` request acquires only a single lock which synchronizes access to the bucket holding the data it's accessing. As a specific piece of data becomes popular, the lock protecting its bucket becomes highly contended, negatively impacting the latency of all requests attempting to access that bucket. However, it is important to note that such contention and high delay impact *some but not all* of the requests the application handles. The remainder of the requests can be accessing different buckets incurring no contention, finishing with minimal latency.

To maintain good performance under lock contention, one must reduce the load on the contended lock, and thus the latency of requests attempting to acquire it. On the other hand, this should not be done in a way that affects the throughput of requests not facing contention. The classic tension between throughput and latency is exacerbated in this case due to the unpredictability of request behavior: the locks accessed by a request can only be known after the execution of the request starts. Thus, the delay faced by different requests, that look identical when admitted to the server, can be very different depending on whether they attempt to access a contended resource or not. This renders overload signals that consider the

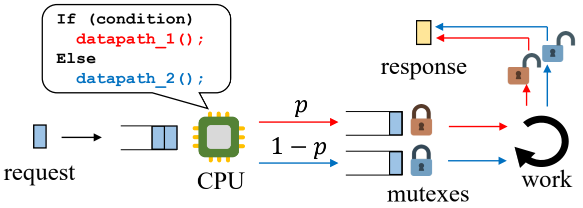


Figure 1: A simple example application with two global mutexes. With a probability p, the request takes the first data path (red arrow).

overall delay of requests ineffective. Furthermore, blocking locks can prevent the load from saturating the CPU, rendering CPU-based overload signals ineffective as well.

In this paper, we attempt to answer the following question: *how should an overload controller decide to admit a request when it can't estimate the delay the request will face?* Tackling this challenge is exacerbated by the fact that some applications have thousands of locks. Moreover, shedding load after processing a request requires cleaning up the state and resources touched by that request.

We present **Protego**, a system that provides overload control for applications that can experience lock contention (§3). Instead of using traditional overload control signals, it admits load as long as it observes throughput improvements. This approach ensures high throughput for requests not experiencing contention. However, it can exacerbate lock contention. Thus, Protego introduces new latency-aware synchronization primitives that allow applications to maintain low latency at contended critical sections, aborting requests when lock contention is too severe. As a result, Protego can offer the right load to maximize a server's throughput, even if some requests must be aborted during processing. We implemented Protego and compared it to SEDA and Breakwater, two state-of-the-art overload schemes, for three applications: Memcached, Lucene, and a synthetic application (§4). Our evaluation demonstrates that Protego outperforms SEDA and Breakwater for a wide range of workloads and applications (§5). For example, when Memcached is handling a `SET`-heavy workload, Protego achieves up to 1.6× more goodput with 5.7× lower 99th percentile latency compared to SEDA.

Protego has some limitations. It requires application-level code changes to adopt our synchronization API. Furthermore, existing overload control schemes can achieve slightly higher throughput than Protego when locks are not the bottleneck and requests are shorter than a microsecond.

Protego is an open-source software available at https://inhocho89.github.io/protego/.

2 Motivation

2.1 Locking Complicates Overload Control

In modern datacenter applications, RPC requests often require blocking synchronization (e.g., mutexes, semaphores, and conditional variables) to serialize access to shared data. However, blocking synchronization primitives can experience contention when multiple requests attempt to access the same critical section, leading to a performance bottleneck. This is further complicated by the fact that the locks required by each request may be different depending on the request payload and the program's state. This makes it hard to know the data path a request will take before its actual execution.

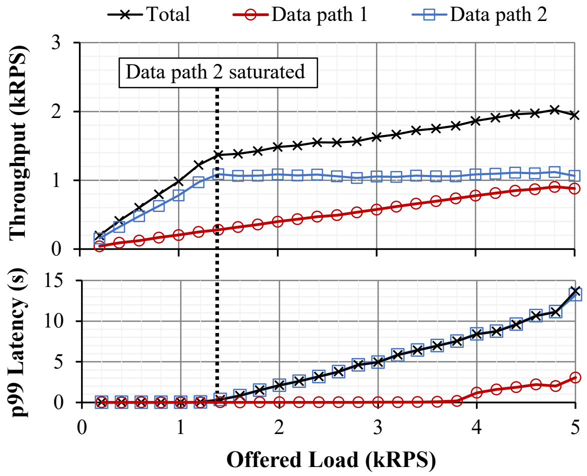
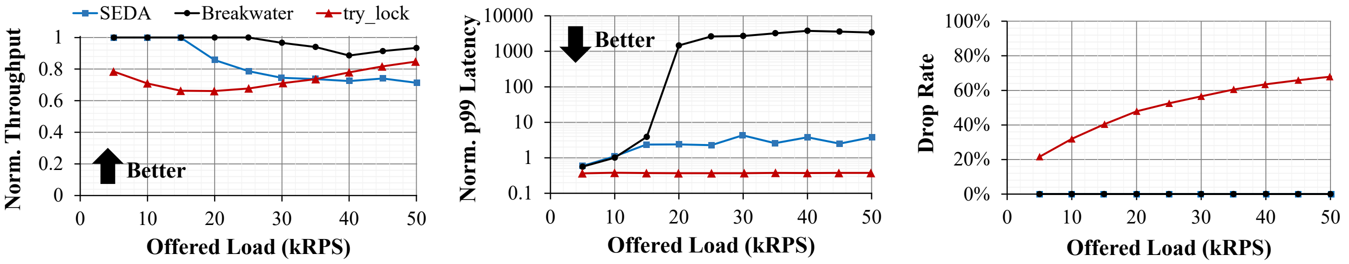


Figure 2: gRPC performance for the example application of Figure 1 ($p = 20\%$). After acquiring a mutex, requests busy-loop for a time sampled from an exponential distribution with 1 ms average. Four cores are allocated for this experiment, one for each data path and two to absorb any system overhead, ensuring that the CPU is not bottlenecked.

The crux of this problem is that seemingly identical requests can have different execution paths at the server with different latency and throughput characteristics. This unpredictable behavior makes admission control hard, leading to the question: *which data path should admission control consider when admitting new requests?* To better understand this dilemma, consider the scenario in Figure 1. Incoming requests can take one of two paths, each protected with a different mutex. Requests can take the first data path with probability p, where $0 \le p \le 1$, and the second path with probability $1 - p$. We implemented this simple scenario in gRPC running on Linux. Figure 2 shows the performance of this scenario with $p = 20\%$ under various loads generated by client machines with an open-loop Poisson arrival process.

The existence of multiple data paths with different lock bottlenecks creates a dilemma. As shown in Figure 2, different datapaths are saturated at different offered load levels. Typically, clients and servers can't predict whether a request will take the datapath currently bottlenecked (data path 2 in the example). Here, the admission control dilemma emerges from the existence of multiple desirable operating points. If the operator desires low latency for all paths, then they have to sacrifice throughput, admitting only enough load to saturate the most congestion execution path (i.e., 1.2 kRPS in this example). On the other hand, if they desire high throughput, then they have to admit a high load and deal with the congested path through other means (e.g., dropping a request



(a) Normalized Throughput

(b) Normalized 99%-ile latency

(c) Drop Rate

Figure 3: Performance of Breakwater, SEDA, and trylock for the example application of Figure 1 ($p = 20\%$) with 100 µs average service time on Shenango. Throughput and 99th percentile latency are normalized by the performance of Protego.

after admitting it). Next, we show that no existing overload control scheme can navigate this dilemma and produce good results in such scenarios.

2.2 Problems with Existing Overload Control Schemes

Overload control attempts to operate a server near its capacity with minimal SLO violations and request drops. The basic idea behind overload control is to keep track of the load on the server using a signal, adjusting the admitted load based on that signal. Multiple signals have been proposed to improve the accuracy of admission control, including CPU utilization [30], end-to-end delay [32], and queuing delay [9,19,34]. However, none of these signals are useful in lock contention scenarios where the operator attempts to maximize throughput while maintaining low latency.

For example, Breakwater [9] and Swift [19] use past observations to predict the amount of queueing delay each request will face. However, in the presence of thousands of locks, it's unclear which queueing delay value (or statistic), if any, can be used to perform admission control. This is because admission control doesn't know in advance which locks requests will access, making it impossible to decide which value to react to without overestimating or underestimating overload. Note that any CPU-based metrics also fail as the CPU might not be the bottleneck in lock contention scenarios.

One possible approach to handle problematic or unpredictable lock behavior is to leverage existing primitives like `try_lock()` or `timed_mutex()`. Specifically, such primitives will allow requests to fail, avoiding latency, if the lock cannot be acquired due to congestion. However, overload control schemes that rely exclusively on request drops do not scale well due to the large overhead of packet drops. Furthermore, relying on existing primitives is not straightforward; `try_lock()` is a very aggressive overload control mechanism because it causes a request to fail on the first failed attempt to acquire a lock. On the other hand, `timed_mutex()` is too relaxed, forcing a request to wait for the full waiting time even under severe congestion conditions.

We demonstrate the limitation of existing overload control schemes, including the usage of `try_lock()`, by implementing those schemes for the scenario described in Figure 1, setting the average service time to 100 µs. However, rather than using gRPC, we leverage the existing implementation of SEDA and Breakwater [3]. Breakwater spawns a new thread per incoming request. We limit the number of spawned threads to bound the memory usage of the system. When a request is aborted, a failure message is reported to the client. The results are shown in Figure 3, comparing the throughput, tail latency, and drop rate of existing schemes, normalized by the performance of Protego.

SEDA successfully bounds the tail latency as it rate-limits clients based on the measured tail end-to-end latency. However, by considering only the tail latency, it reacts to the most congested path, leading to poor throughput as it underutilizes the uncongested path. Breakwater reacts only to queueing delay in the thread queue or the packet queue, reacting only to CPU and network overload. Thus, it doesn't perform any rate-limiting because neither the CPU nor the network is the bottleneck. Breakwater's behavior leads to high utilization and very high latency. Using `try_lock()` allows the system to achieve near-ideal latency while suffering from an extremely high drop rate and poor throughput. This is caused by `try_lock()`'s aggressiveness in dropping requests, wasting CPU and throughput even at low loads. Our proposal overcomes the shortcomings of existing systems, achieving the highest throughput while keeping the latency and drop rate low.

2.3 Challenges

Existing overload control schemes, developed for CPU overload scenarios, suffer significant performance degradation when handling lock contention. The key issue when dealing with lock contention is the unpredictability of the latency that a request will face. Particularly, the overload controller doesn't know which lock a request will require. This issue leads to the following challenges:

1. No existing overload control signal is viable. As discussed earlier, delay reflects the state of the most congested path. On the other hand, CPU utilization is not helpful when the bottleneck is not the CPU. Thus, we need a new approach to assessing the capacity of the server in order to make accurate admission control decisions.

2. Drops are inevitable to achieve high throughput. An overload controller that doesn't react to the most congested data

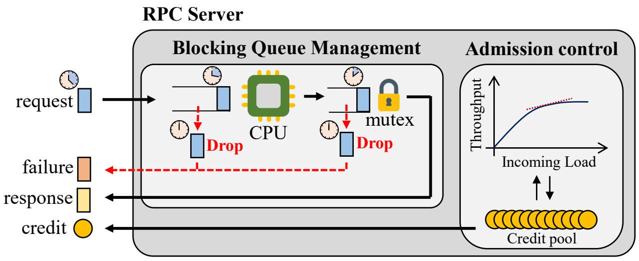


Figure 4: Protego Overview

path will incur a high delay for requests taking that path. However, it must offer enough load to keep other, less-congested paths busy. Therefore, maintaining both an acceptable SLO and high CPU utilization requires dropping requests on the most congested paths and reporting failures to the client. Early failure reporting allows the client to issue the requests to another replica while maintaining the SLO of the request.

3. Any viable solution must scale to a large number of locks. Modern programs can have thousands of data paths and synchronization primitives. An incoming request can take any of them depending on the data it carries. Thus, the admission control scheme needs to scale to a large number of locks with minimal per-lock overhead.

3 System Design

There is a fundamental tradeoff between throughput and drop rate in the presence of unpredictable synchronization. To achieve high throughput, clients should offer enough load for the server to fully utilize its uncontended data paths. Unfortunately, this permits some congestion to occur in its contended data paths. Thus, our high-level strategy is to use an admission control scheme that admits enough load to keep all data paths operating at full capacity, combined with an Active Queue Management (AQM) mechanism that drops excess load on the contended data paths. Our admission control scheme draws insight from network congestion control algorithms like PCC [12]. Specifically, Protego does not react to a specific overload signal. Rather, it observes the impact of its current admission rate on the behavior of the system, admitting more load only when it improves overall system performance.

Figure 4 illustrates an overview of Protego combined with a simple RPC server that uses a global mutex. Protego is composed of two main components: an admission controller and an AQM mechanism. The admission controller leverages a credit-based scheme, similar to the scheme used in Breakwater [9]. Protego only changes the way the number of available credits is decided, adjusting the number of credits by observing the impact of increasing the number of available credits on achieved throughput. The AQM mechanism uses *Active Synchronization Queue Management* (ASQM), a novel form of AQM that drops requests at lock acquisition time to prevent blocking on a critical section for an excessive amount of time. When a request is dropped, Protego reports this failure as quickly as possible to clients, allowing them to resend their requests to another replica.

3.1 Performance-driven Admission Control

Our goal is to develop an admission control algorithm that allows a server operator to navigate the tradeoff between throughput and drop rate. Note that the admission control algorithm should support scaling to a large number of data paths. Thus, we avoid developing an algorithm that has to take into account the state of every data path in the server.

Intuition. To better understand the intuition behind our algorithm, we go back to the setup in Figure 1. Specifically, we rerun the experiment discussed in Section 2.2. However, we use a smaller service time per request (10 µs rather than 100 µs) because these results help to make our point clearer. Moreover, we don't use any admission control scheme but rely on the AQM scheme, discussed in the next section, to keep latency bounded. The results are shown in Figure 5. The design of our admission control scheme stems from observing that as the load increases, the system operates in four different phases:

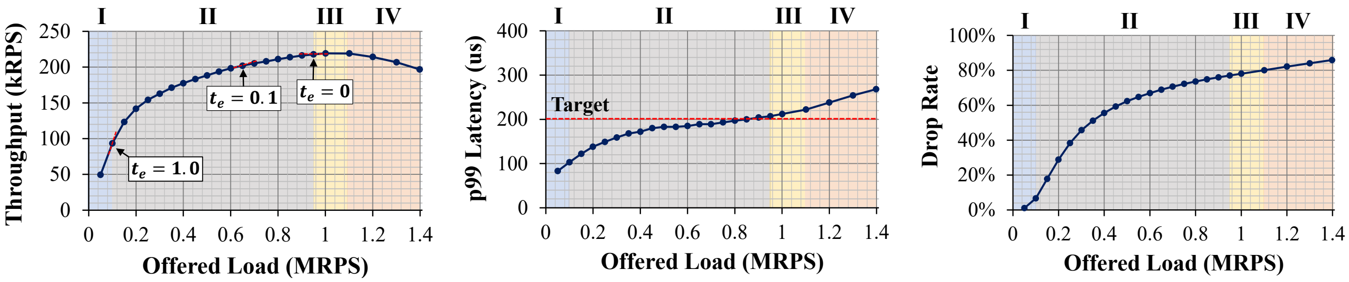
Phase I (uncongested) is the phase where none of the locks or CPUs is congested. Throughput grows linearly with load increases because the system has capacity to handle all incoming demand. Further, tail latency increases only marginally because of bursts in the queue caused by the variable request arrivals, modeled as a Poisson arrival process. With no congestion, AQM does not drop the requests.

Phase II (partially congested) is the phase where a subset of locks are contended. As load increases, throughput increases sub-linearly because the system has capacity to handle only a fraction of incoming demand (i.e., the uncongested path still has capacity). Incoming requests that take the congested path will face high queueing delay, leading AQM to start dropping requests while keeping the tail latency near the target value. To generalize, different applications will produce a different concave line like that shown in Figure 5(a), where the slope of the curve decreases as more paths become congested. The exact shape of the curve depends on the number of congested paths, and their capacities along with the load.

Phase III (congested) is the phase where all the data paths become congested. Thus, as the load increases, the throughput doesn't change. However, the increase in load increases CPU utilization because of the increase in network processing load and the increasing overhead of dropping requests. Eventually, the CPU also becomes congested, increasing tail latency.

Phase IV (congestion collapse) is the phase where the system enters a livelock state, spending more time dropping requests than processing them. During that phase, throughput degrades and latency keeps increasing.

Overview. Admission control should bound the incoming load to make the server operate in Phase II. Note that the values of latency, drop rate, and CPU utilization do not help



(a) Throughput (b) 99%-ile latency (c) Drop Rate

Figure 5: Performance of the application in Figure 1 ($p = 20\%$) with 10 µs average service with the latency bounded by ASQM

Algorithm 1 Performance-driven credit management

```
$t_e$: efficiency threshold
$t_d$: maximum drop threshold
$C$: the size of credit pool
$in_{\{last,cur\}}$: # of incoming requests in {last, current} iteration
$out_{\{last,cur\}}$: # of outgoing responses in {last, current} iteration
$drop_{cur}$: # of request drops in current iteration
$a$: increment step size
$d$: multiplicative decrement factor

repeat Every 4 * end-to-end RTT
    if $drop_{cur} > t_d \cdot in_{cur}$ then
        $C \leftarrow (1-d) \cdot C$
    else if $(in_{cur} - in_{last})(out_{cur} - out_{last}) > 0$ then
        if $|out_{cur} - out_{last}| > t_e \cdot |in_{cur} - in_{last}|$ then
            $C \leftarrow C + a$
        else
            $C \leftarrow (1-d) \cdot C$
        end if
    else
        $C \leftarrow (1-d) \cdot C$
    end if
    $C \leftarrow \max(C, C_{min})$
    $C \leftarrow \min(C, C_{max})$
    $in_{last} \leftarrow in_{cur}$
    $out_{last} \leftarrow out_{cur}$
until Application exits
```

identify the phase in which the server operates. However, by observing the slope of the throughput curve, one can identify the boundaries of Phase II. Specifically, Phase II starts when the slope of the throughput curve drops from 1 (i.e., the system can no longer handle *all* incoming requests) and ends when the slope reaches 0 (i.e., the system can no longer handle *any* additional incoming requests). A server operator that's interested in achieving a near-zero drop rate would operate the server at the leftmost edge of Phase II, where the slope of the throughput curve is slightly lower than one. On the other hand, a server operator that's interested in achieving the highest possible throughput would operate the server at the rightmost edge of Phase II, where the slope of the throughput curve is slightly higher than zero. The server operator can operate between those two points by choosing desired slope value. Additionally, the operator could specify the region of operation further by capping the maximum allowed drop rate.

We propose a performance-driven admission control algorithm with two parameters: efficiency threshold (t_e) and maximum drop rate (t_d). The efficiency threshold represents the target operating point on the throughput curve in terms of the slope of the curve at that point. Specifically, t_e takes values between zero and one, with zero representing the highest possible throughput, and one representing zero drop rate. The maximum drop rate, t_d, allows a service operator to cap the drop rate at the expense of throughput to reduce the expected number of request drops. Protego uses the maximum drop rate in addition to the efficiency threshold to determine whether to accept more incoming load. Protego judges an RPC server to be overloaded, accepting no further load, if throughput improvement with additional load is less than the efficiency threshold or if the drop rate exceeds the maximum drop rate.

Operation. A Protego server controls the number of incoming requests through the credit-based scheme we developed for Breakwater [9]. We chose a receiver-driven credit-based admission control scheme because it was shown to be robust to incast scenarios, efficiently scaling to a large number of clients while maintaining its performance [8, 9, 17, 23]. Like 1RMA [29] and Breakwater [9], Protego requires a new client to declare its intent to send requests to the server by sending an initial Request To Send (RTS) message. For Protego, this message is needed only when a new client connects to the server and is not needed for every request. The server issues credits to clients. A credit represents availability at the server to process a single request by the client that receives the credit. A client only sends a request after it receives a credit. A client disconnecting from the server has to send a Disconnect message to inform the server to stop allocating credits to it. Note that credits in Protego provide minimal commitment as the server cannot know in advance whether an incoming request will take a congested or an uncongested path. Protego determines the total number of available credits, C, before distributing them to individual clients.

The server measures its efficiency (the change in throughput divided by the change in admitted load). If measured efficiency is less than the efficiency threshold (t_e), the server reduces the credit pool size, reducing the admitted load; otherwise, it increases the credit pool size. In particular, the server operates in iterations, each lasting a few end-to-end RTTs.[1] We measure the end-to-end RTT with the elapsed time between credit issue and the successful response return which is tracked with an 8B unique credit ID. The server keeps track of the number of admitted requests from the current iteration and the previous iteration, in_{cur} and in_{last}, respectively. It also keeps track of the current throughput and the throughput in the previous iteration, out_{cur} and out_{last}, respectively. The efficiency metric $e = (out_{cur} - out_{last})/(in_{cur} - in_{last})$ is compared to the efficiency threshold t_e. The server continuously monitors the drop count $drop_{cur}$ and decreases the admitted load if $drop_{cur}$ exceeds $t_d \cdot in_{cur}$. Protego uses additive increase / multiplicative decrease (AIMD) for credit management due to its simplicity. The details of the algorithm are shown in Algorithm 1.

3.2 Active Synchronization Queue Management (ASQM)

Protego assumes a standard queue abstraction per blocking synchronization object. However, to ensure scalability, Protego requires no coordination between queues, no per-queue parameter setting, and only minimal changes to the existing implementation of the synchronization API. Specifically, ASQM caps the total time a request is allowed to spend in a queue, assigning each request a queueing delay budget. The value of the budget represents the maximum queueing delay a request can tolerate for the server to respond within a target latency. The queueing delay budget is computed by subtracting the 99th percentile network latency and 99th percentile service time from the *target delay* of the request, leaving the slack time that the request can afford to spend in the server.

When a request arrives at the server, Protego assigns it a queueing delay budget. Before placing the request in each queue for a contended resource, it first checks the instantaneous queueing delay of the queue and drops the request if the queueing delay is larger than the request's remaining queueing delay budget. After the request is dequeued, it deducts the queueing delay it incurred from its budget. The queueing delay is measured by computing the difference between the current timestamp and the enqueue timestamp of the oldest item in the queue. In this paper, we only consider the runnable thread queue in the CPU scheduler and the wait queues for blocking synchronization primitives. However, we believe the same idea can be applied to other queues for contended blocking interfaces such as blocking I/O.

Target delay vs. SLO. It's critical to note that the target delay used to compute the queueing delay budget is different from the RPC's Service Level Objective (SLO). The target delay is a per-server metric: a single server should finish a request or report failure within the target delay. On the other hand, an SLO is a per-request metric: a request of a specific type should finish within its SLO, taking into account that multiple attempts at multiple servers might be needed for the request to succeed. In Protego, the target delay is set by default to SLO divided by the maximum number of retries.

Handling dropped requests. Upon a request drop, the server returns a failure message immediately to the client. At the server, a request drop incurs some CPU overhead to partially process the request and generate the failure message. Further, the failure message and retransmission of the request can incur networking overhead. If the overhead of dropping requests is large, a service operator can reduce the drop rate by choosing a higher value for the efficiency threshold (t_e), sacrificing throughput. At the clients, the dropped request may be handled in various ways: retransmission to another replica, triggering failure handling operations (e.g., online banking transaction), or degrading the quality of the response (e.g., search). For systems with replication and auto-scaling, retransmission is the most common failover mechanism. For the rest of the paper, we focus on scenarios where an overloaded server has a non-overloaded replica which can serve dropped requests.

Retransmission of dropped requests introduces additional latency, inflating the overall delay faced by such requests, potentially harming their SLOs. Protego drops requests before they consume their delay budget. Thus, clients receive failure messages within the target delay. In the worst case, for each retransmission, a request will be delayed by at most the target delay (§5.3). Alternatively, if the SLO is tight, the client can send tied or hedged requests to multiple replicas to avoid the retransmission delay but incur the cost of coordination overhead and/or CPU wasted by duplicate executions [11].

3.3 System Parameters

In total, Protego has five parameters: four universal parameters whose value can be fixed across workloads, and one workload-specific parameter.

Universal parameters. The efficiency threshold and maximum drop rate parameters, t_e and t_d, do not need to change per workload. We show that the performance of Protego is not very sensitive to the choice of t_e (§5.4). We use an efficiency threshold of 10% by default. The maximum drop rate puts a cap on the allowed drop rate. Operators that want to maximize throughput should set it to 100%, which is the default value we choose in the paper.

AIMD algorithms have two parameters: an increment step size (a) and a decrement factor (d). Large values of a and d make the algorithm more aggressive in reaching the desired operating point but less stable with large fluctuations. We choose small values for a and d, preferring stability. We set a as 0.1% of the number of the client sessions and d as 2%, which leads to good performance in incast scenarios [9].

Workload-specific parameters. The target delay specifies

[1] We found that four RTTs allows for accurate measurement of all parameters while allowing for fast reaction to changes in the workload.

the maximum delay allowed in a single server. Its value is calculated as the SLO divided by the expected number of attempts that a request can make before it succeeds.

4 Implementation

We implemented Protego as a library that uses Shenango [25] and builds upon the RPC-layer implementation of Breakwater [9]. Furthermore, Protego extends Shenango's synchronization library to implement ASQM, facilitating the adoption of Protego to Shenango applications.

Performance measurement. Protego adjusts the credit pool size, once every iteration, based on five measures of efficiency and drop rate: in_{cur}, out_{cur}, $drop_{cur}$, in_{last} and out_{last}. The measures are updated (i.e., current measures are reset after their values are assigned to the last measures) after one end-to-end RTT from the time the credit pool size is updated to accurately reflect performance during an iteration. This period is selected because the incoming load changes in correspondence to the new pool size after at least one end-to-end RTT.

Dispatcher threading model. Protego assigns a queueing delay budget per request, deducting from it after a request is serviced from a queue. This operation requires accurately tracking the time a request spends in various queues, avoiding any variability that might be introduced due to the operating system or the network stack. Thus, we implement Protego with a dispatcher threading model where a dispatcher thread parses the network payloads into requests, spawning a new thread for each incoming request. This approach minimizes the delay requests face in the network stack because packets are parsed quickly by the dispatcher thread, out of the critical path of request processing. When a new thread is created by a dispatcher, it's assigned a queueing delay budget by subtracting the 99th percentile network latency and the 99th percentile service time from the target delay.

Latency-aware Active Synchronization Queue Management (ASQM) API. Protego provides the following latency-aware APIs to enable ASQM:

```
bool mutex_lock_if_uncongested(mutex_t *);
bool condvar_wait_if_uncongested(condvar_t *,
                                 mutex_t *);
```

These interfaces are similar to those of a `try_lock()`, but their behavior is different. If the queueing delay of a blocking critical section exceeds a request's queueing delay budget, it returns false without waiting. Otherwise, it returns true after successfully acquiring the lock. An application developer can leverage the existing synchronization API provided by Shenango, including `mutex_lock()` and `condvar_wait()` for parts of the program that cannot handle dropping. For example, a maintenance thread running in the background may need to acquire a lock no matter how long it has to wait.

Queueing delay measurement. Protego needs to measure instantaneous queueing delay to compare it against a request's remaining budget. We instrument the waiter queue for mutexes and conditional variables to measure the queueing delay.When a thread is enqueued to the waiter queue, Protego timestamps the request. When the blocking synchronization is queried for the queueing delay, it returns the difference between the current timestamp and the enqueue timestamp of the oldest thread in the waiter queue. Using an efficient hardware timestamp read function, Protego can measure the queueing delay of blocking synchronization with little overhead.

Identifying contended locks. In order to get the full performance benefits of Protego, developers must identify all the contended locks to replace with Protego's ASQM APIs. A developer needs to hypothesize which locks are likely to be contended based on the application-specific knowledge and run experiments to verify which locks introduce a large queueing delay with per-lock queueing delay measurements. This process requires iterating multiple times until all the contended locks are identified and their code is modified to use the Protego API. Alternatively, a developer can use high-resolution latency profilers [16] to identify contended locks.

Application modification. Enabling Protego requires replacing blocking synchronization primitives with the ones provided in the Protego API. Further, Protego allows requests to be dropped after they have been partially processed by the server, potentially modifying some states or reserving some resources. Thus, enabling Protego requires the application to perform all necessary clean-up after a request is dropped (e.g., freeing memory it allocated to the request and releasing other locks the request currently holds). However, the complexity of handling request drops can be significantly reduced by utilizing features of modern programming languages, such as RAII in C++ with smart pointers and scoped locks.

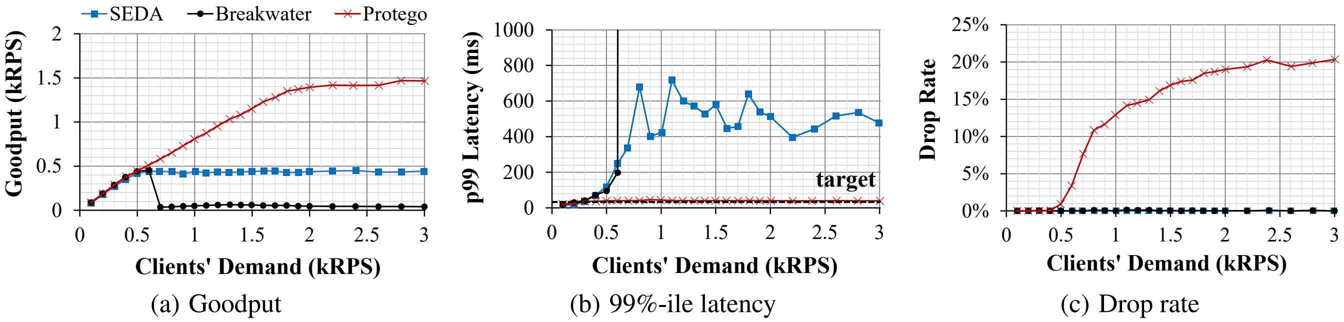
5 Evaluation

Our evaluation answers the following key questions:

1. Can Protego balance high throughput and low latency for real-world applications?
2. How much code change is required to enable Protego?
3. Does Protego maintain its benefits for different workloads?
4. Can requests maintain their SLO in the presence of drops?
5. How much does each component of Protego contribute to its overall performance?
6. How sensitive is the performance of Protego to its parameter values?
7. What are the limitations of Protego?

5.1 Evaluation Setup

Testbed: We use eleven xl170 nodes in Cloudlab [13]. Each node has a ten-core (20 hyper-threads) Intel E5-2640v4 2.4GHz CPU, 64GB ECC RAM, and a Mellanox ConnectX-4 25GbE NIC. Nodes are connected through a single Mellanox 2410 switch. The average and 99th percentile network RTT between any pair of two nodes are 10 µs and 20 µs, respectively. We use one node as an RPC server and the other ten nodes



(a) Goodput

(b) 99%-ile latency

(c) Drop rate

Figure 6: Performance of SEDA, Breakwater, and Protego for Lucene

as RPC clients. The server application uses up to ten hyper-threads for real-world applications and four hyper-threads for synthetic application. Each client machine simulates one hundred RPC client connections with sixteen dedicated, spinning hyper-threads. Requests are generated following an open-loop Poisson arrival process.

Workloads: We evaluate Protego using three workloads: 1) Lucene, a search application with significant lock contention overhead, 2) Memcached, a latency-sensitive in-memory key-value store that exhibits both locking bottlenecks and CPU bottlenecks, and 3) a synthetic workload with its execution time drawn from an exponential distribution.

Baseline: We compare Protego to SEDA, a latency-based overload control system, and Breakwater, a queueing delay-based one. SEDA controls the load at the server by rate-limiting the clients. Each SEDA client adjusts its request sending rate based on the 99th percentile end-to-end latency faced by requests. Breakwater controls the load at the server through a credit-based mechanism, adjusting the credit pool size based on the sum of packet queueing delay and CPU thread queueing delay. To ensure low latency, Breakwater drops a request if the queueing delay exceeds a workload-based threshold.

Evaluation metrics: To incorporate throughput, latency, and the target latency into one single metric, we compute goodput as the throughput of the requests whose latency is below the target delay. For Breakwater and Protego, we report the drop rate as the ratio of the number of dropped requests to the number of requests received by the server during an experiment. SEDA does not drop the request at the server. We run the experiments for 8 seconds and collect the data for the last 4 seconds to capture the steady-state behavior.

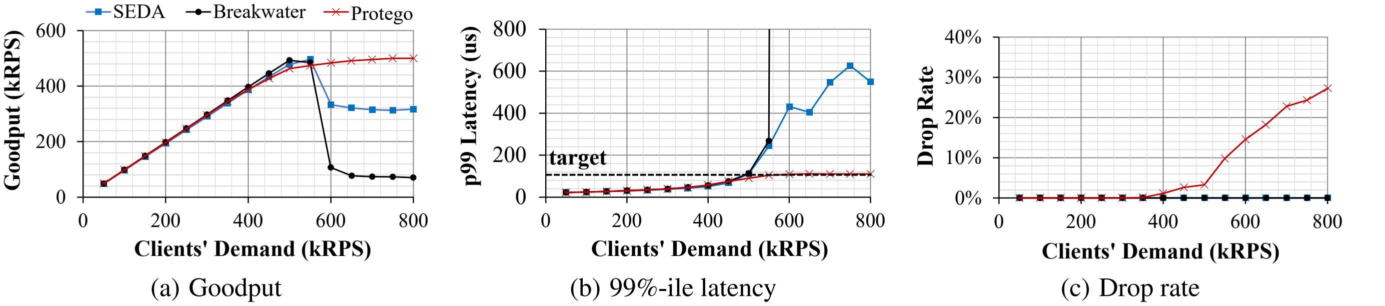
Parameter settings: We tune the parameters of all systems to allow each system to achieve its best goodput for each workload. For SEDA, we adjust *timeout* (request sending rate update interval), adj_i (rate increase factor), and adj_d (rate decrease factor). We use the default configuration from [32] for all other parameters. For Breakwater, we tune the target queueing delay and the drop threshold which we set to 40% and 80% of Protego's target delay, respectively, for all workloads. We use the default configuration from [9] for all other parameters. For Protego, we use an efficiency threshold (t_e) of 10%, a maximum drop rate (t_d) of 100%, an increment step size (a) of 1, and a decrement factor (d) of 2% for all workloads. We determine the queueing delay budget for ASQM by deducting 99th percentile service time and 99th percentile network delay (20 µs) from the target delay for each workload. We determine the target delay as the maximum value between 10 × the sum of average network RTT (10 µs) plus the average service time, and 2 × the sum of 99th percentile network RTT (20 µs) plus the 99th percentile service time. For example, for the exponential service time distribution with 10 µs average whose 99th percentile is 46 µs, we set the target delay to 200 µs because $10 \cdot (10+10) = 200$ µs is higher than $2 \cdot (20+46) = 132$ µs. The way we set the target delay is comparable to how the SLO is calculated in recent proposals [9, 10, 26]. We set the SLO as twice the target delay, assuming that a request fails at most once.

5.2 Mutex-intensive Application: Lucene

Lock contention inside Lucene: Lucene is a search engine library that maintains two main types of structures: 1) inverted indices, called `Segments`, and 2) per-term scores of all indexed documents, called `TermDocs`. Every `Segment` and `TermDocs` is protected by its own mutex. Every request performs a binary search over all `Segments` to find the documents corresponding to its search query. Then, documents are ranked based on the information found in the `TermDocs` corresponding to the identified documents.

As load increases on the server, the per-`Segment` lock becomes contended because every request needs to search over all the `Segments`. `Segments` containing more entries are more likely to be contended because it takes more time to perform a binary search over their entries. Further, if a specific document becomes popular, the per-`TermDocs` lock protecting its data becomes contended.

Application modification: We ported the C++ version of Lucene, Lucene++ [31], to Shenango and built a simple in-memory search application, where all the data is stored in memory with `RAMDirectory`. We replaced the per-`Segment` lock and per-`TermDocs` lock with Protego's latency-aware synchronization API to allow request drops. In total, we modified 40 LOC of Lucene++ after porting it to Shenango. Note that, while Lucene allows for reporting partial search results,



(a) Goodput (b) 99%-ile latency (c) Drop rate

Figure 7: Performance of SEDA, Breakwater, and Protego for Memcached with VAR workload

we don't allow that to provide a fair comparison between overload control schemes that don't drop requests. The response contains either the complete search result or a failure notification.

Workload and configuration: We populate the server with a dataset of 403,619 COVID-19-related tweets [6] in English posted between 27th and 29th November 2021. The clients generate single-term search queries. The search term (or word) is sampled from the word distribution in the data set excluding stop words like "a", "the", "and", etc. All the tweets are loaded to the server before serving clients, and tweets are not modified or deleted during an experiment. This workload yields an average processing time of 1.7 ms and a 99th percentile latency of 20 ms on a lightly-loaded server. Thus, we set the target delay to 40 ms. For SEDA, we set $timeout = 1$ s, $adj_i = 0.1$, and $adj_d = 1.3$. For Protego, we use an initial queueing delay budget of 20 ms.

Overall performance: Figure 6 shows the goodput, 99th percentile latency, and drop rate for all three overload control schemes. Note that Lucene does not suffer from any CPU congestion. Thus, Breakwater's admission control and AQM are never triggered, leading to congestion collapse as mutexes become congested with demand exceeding 600 RPS. SEDA reduces clients' request sending rate as soon as it measures high latency due to a mutex congestion, reacting to the most congested data path, which limits the system's goodput to 500 RPS. SEDA's tail latency is bounded but more than 10 times higher than the target latency because of incast. By better utilizing uncongested data paths and dropping the excess load, Protego achieves up to 3.3 times higher goodput and 17 times lower 99th percentile latency than SEDA.

5.3 Latency-critical Application: Memcached

Lock contention inside Memcached: The key-value pairs are stored in a giant hash table, composed of multiple hash buckets. Memcached has two main types of locks that may be contended. First, each hash bucket is protected by a mutex called `item_lock`, and this mutex may get contended not only by concurrent accesses (i.e., reads or rights) to the same key but also by accesses on different keys sharing the same key hash. Thus, it's difficult to predict which `item_lock` a request will need before executing it. Second, Memcached manages its memory by assigning items memory from a global pool, which is protected by a global lock called `slabs_lock`. Every `SET` and `UPDATE` request must grab the `slabs_lock` to allocate memory for the new value.

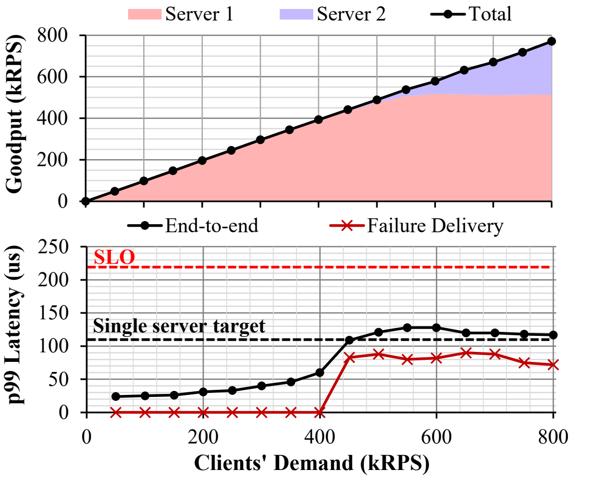
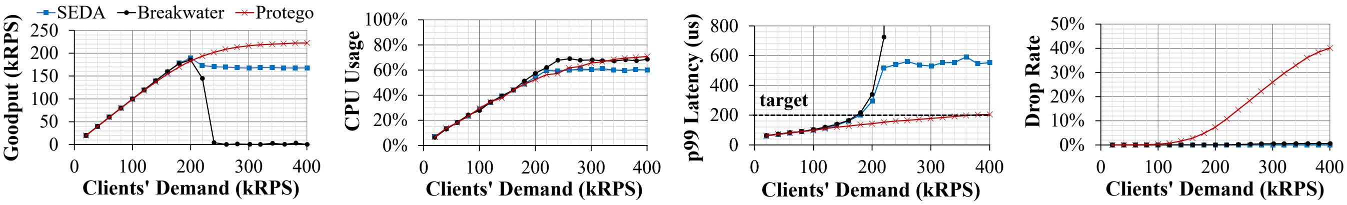


Figure 8: Service-level performance of Protego for the Memcached VAR workload with retransmission

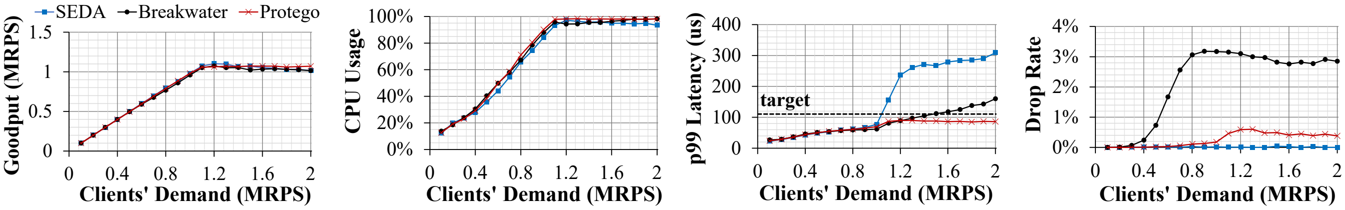
Application modification: We replaced the `item_locks` and `slabs_lock` with Protego's latency-aware mutexes. When a request is dropped, Protego delivers a failure message to the client immediately. Furthermore, it cleans up the intermediate state processed by the request, freeing up the chunk allocated to the request before the thread handling that request exits. We don't allow drop when a request tries to reacquire `slabs_lock` to free up the memory to avoid memory leaks. In total, we modified 50 LOC in Memcached [4], excluding the modifications to port it to Shenango.

Workload and configuration: For Memcached experiments, we use the VAR workload from Facebook Memcached cluster [33]. VAR is a `SET`-heavy workload for server-side browser information where 82% of the requests are `SET` requests. The key distribution of the workload is skewed with 10% of the keys used by 90% of the requests. With a `SET`-heavy workload, `slabs_lock` becomes the bottleneck as all SET requests require `slabs_lock` to allocate memory region. We approximately follow the key and value size distribution for each workload as described in [33]. We generate 100,000 key-value pairs and use the hash power of 17, providing 131,072 buckets in the hash table, which is sufficient to avoid severe hash col-



(a) Goodput (b) CPU usage (c) 99%-ile latency (d) Drop rate

Figure 9: Performance of SEDA, Breakwater, and Protego for synthetic workload with $p = 50\%$ and 10 µs average service time



(a) Goodput (b) CPU usage (c) 99%-ile latency (d) Drop rate

Figure 10: Performance of SEDA, Breakwater, and Protego for synthetic workload with $p = 50\%$ and 1 µs average service time

lisions. Since SET requests complete within less than 1 µs on average, we set the target delay to 110 µs. For SEDA, we set *timeout* to 1 ms, adj_i to 100, and adj_d to 1.02. For Protego, we set the initial queueing delay budget to 70 µs.

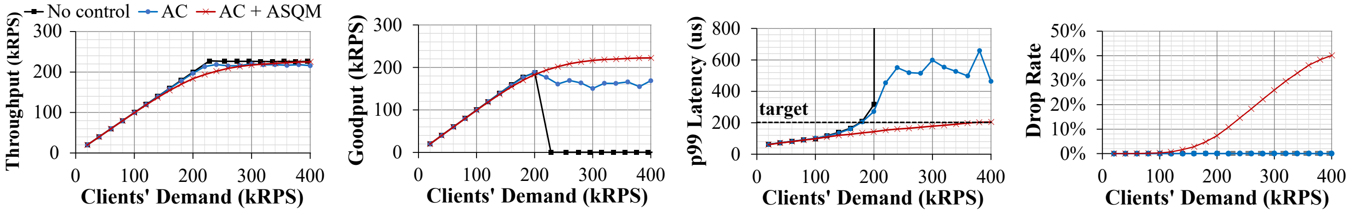
Performance with a global mutex bottleneck: Figure 7 demonstrates the performance of the three overload control schemes. When the slabs_lock becomes contended with clients' demand of more than 550 kRPS, both Breakwater and SEDA experience a goodput drop because of the increase in latency. As with Lucene, the admission control and AQM of Breakwater are not triggered because the CPU is not congested. On the other hand, SEDA suffers from incast. The goodput of Protego increases further by utilizing uncongested data paths with GET requests achieving 1.6 times higher goodput than SEDA and 7 times higher goodput than Breakwater. The increment in Protego's goodput is limited by the overhead of request drops. Most of the dropped requests are SET requests, and some of them require the slabs_lock to free the allocated memory. As more requests are dropped, the slabs_lock becomes more contended by new SET requests that need to allocate the memory as well as old and dropped requests that need to release their memory, resulting in lower throughput of SET requests at very high loads.

Maintaining the SLO under retransmissions: To better understand the impact of request drops on the overall SLO, we construct a simple scenario where Memcached has two replicas, but we otherwise use the same configuration as before. When a client makes a request, it sends the request to Server 1. If it is dropped, the client then retransmits it to Server 2 (after receiving a failure message from Server 1). This structure is similar to how Memcached is operated at Facebook [24] where they don't provide a strong consistency guarantee. Note that if both servers are overloaded, the problem ceases to be an overload control problem as the service operator needs to allocate more servers. Thus, our experiment captures the case where there is sufficient capacity to handle all requests, but retransmission may still be necessary. We anticipate up to one retransmission could happen, considering the capacity of the two servers and the demand the clients generate during the experiment, so we set the service-level objective (SLO) to two times the single server target delay, or 220 µs.

Figure 8 demonstrates the total goodput of both servers, the 99th percentile end-to-end latency, and failure message delay for the VAR workload. When the clients' demand exceeds 400 kRPS, Server 1 starts to drop requests. Protego drops the requests before they wait for the contended mutex if the delay at the mutex exceeds a request's budget. Thus, most of the failure messages are delivered within the target delay. Note that if a client doesn't receive a credit for a request within 10 µs from Server 1, it sends the request to Server 2 with the locally generated failure message. As clients' demand increases, the 99th percentile delay of failure messages decreases because more requests are retransmitted to Server 2 with local failure message. The overall 99th percentile end-to-end latency achieved by Protego is higher than the per-server target delay because some requests need to be retransmitted. However, it is still 1.7 × lower than the SLO.

5.4 Microbenchmark

Workload and configuration: To further analyze Protego's performance, we run the synthetic application depicted in Figure 1. We choose the configuration $p = 50\%$, making both data paths equally likely to be congested, to provide a best-case scenario for SEDA. We use a workload with exponential service time distribution of 10 µs and 1 µs average. The target delay values are 200 µs and 110 µs, respectively for the two settings. For SEDA, we set $timeout = 1$ ms, $adj_i = 10$, and $adj_d = 1.04$ for the first setting, and $timeout = 1$ ms, $adj_i = 40$, and $adj_d = 1.04$ for the second setting. For Protego, we set the initial queueing delay budget to 134 µs and



(a) Throughput (b) Goodput (c) 99%-ile latency (d) Drop rate

Figure 11: Performance of Protego by incrementally applying performance-driven admission control (AC) and ASQM with the synthetic application with 10 µs average service time

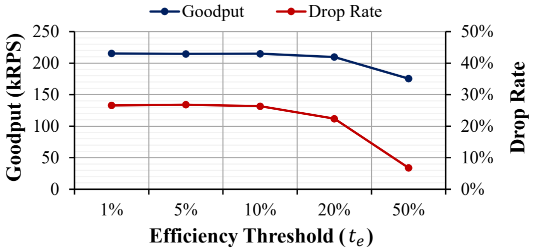
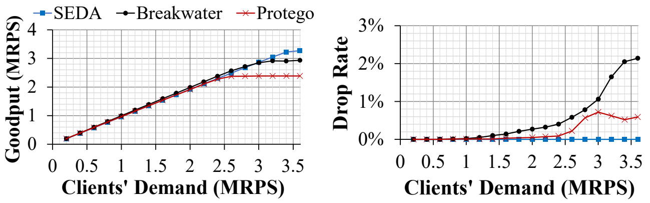


Figure 12: Protego parameter sensitivity (efficiency threshold, t_e)



(a) Goodput (b) Drop rate

Figure 13: Performance of SEDA, Breakwater, and Protego for Memcached with USR workload

85 µs, respectively, for the two settings.

Overall performance: Figure 9 shows the goodput, CPU usage, 99th percentile latency, and drop rate for a workload with 10 µs average service time. The performance is bottlenecked by the mutexes, leaving the CPU underutilized even with a high clients' demand. Thus, at high load, the admission control or AQM logic of Breakwater is not triggered, leading to congestion collapse. SEDA limits the sending rates of clients as soon as it measures high tail latency with a single temporarily congested data path. Thus, SEDA's goodput is limited to 168 kRPS leaving the other data path uncongested. With a larger clients' demand, SEDA suffers from incast because 1,000 clients are each running a control loop separately. As a result, it shows up to three times higher tail latency than the target delay. Protego improves goodput by up to 32% compared to SEDA, maintaining latency within the target delay by dropping up to 40% of incoming requests. Note that the performance benefits of Protego compared to SEDA increase as p deviates from 50%, making SEDA more conservative as it reacts to the most congested path.

Impact of average service time: We reduce the average service time to 1 µs, reducing the time requests can spend with the lock, allowing the CPU to become the bottleneck. The results are shown in Figure 10. As demand exceeds 1.1 million RPS, the CPU is saturated, triggering Breakwater mechanisms. However, it still suffers at high loads when the mutexes become contended. SEDA still suffers from high tail latency up to three times of the target delay because of the incast, but its impact on goodput is limited. Protego maintains the tail latency lower than the target delay while dropping less than 1% of the requests in a CPU-bounded scenario.

Performance breakdown: We measure the performance of Protego after incrementally activating its two components: the performance-driven admission control scheme (AC) and Active Synchronization Queue Management (ASQM). We run the experiments with the synthetic application with $p = 50\%$ and an average service time of 10 µs. Figure 11 shows the throughput, the goodput, the 99th percentile latency, and drop rate. With no overload control, goodput collapses as soon as one of the data paths becomes congested. Enabling admission control bounds the tail latency by limiting incoming load if there is no throughput improvement. However, when mutexes start to be congested, its goodput degrades with up to three times higher tail latency than the target because one of the mutexes can have a high queueing delay with the requests' probabilistic data path selection. By employing ASQM, Protego ensures the tail latency does not miss the target delay by dropping requests.

Parameter sensitivity: Protego balances goodput and drop rate using the efficiency threshold (t_e). To quantify the tradeoff between them, we repeat the experiment with the synthetic application with $p = 50\%$ and the average service time of 10 µs varying the t_e from 1% to 50%. Figure 12 shows the goodput and drop rate of Protego with different t_e values when the clients' demand is 300 kRPS, around 1.4 × of the capacity (consider Figure 11 as a reference). For all values of t_e smaller than 10%, the goodput and drop rate don't change because throughput improvements with a small t_e are always marginal. With larger t_e values, both the goodput and drop rate decrease as admission control targets to operate the server on the left side of the Phase II region in Figure 5. With $t_e = 50\%$, it achieves 23% less goodput and 4 × lower drop rate than $t_e = 1\%$, allowing server operators to navigate the tradeoff between goodput and drop rate.

5.5 Limitations of Protego

To demonstrate the limitations of Protego, we repeat the Memcached experiment in §5.3 with the USR workload, a `GET`-dominated workload for user account status information where 99.8% of the requests are `GET` requests and about 20% of the keys are used by 80% of the requests. With the USR workload, Memcached saturates the CPU when it's configured with a high enough hash power (i.e., a large number of buckets compared to the number of key-value pairs). However, some `item_locks` can still become congested intermittently because of the skewed key distribution. Figure 13 shows the goodput and drop rate, comparing Protego to Breakwater and SEDA. With clients' demand of 3.6 million RPS, Protego achieves 37% less goodput than SEDA and 23% less goodput than Breakwater.

The USR workload is CPU bottlenecked, allowing Breakwater mechanisms to be triggered. Protego achieves lower goodput than Breakwater due to the slow reaction of Protego's admission control. In particular, Protego changes its credit pool size every four end-to-end RTTs. On the other hand, Breakwater adjusts its credit pool size every network RTT. As a result, Protego reacts to both congestion and added capacity slowly, leading to a lower goodput. Breakwater and Protego achieve lower goodput than SEDA because of the overhead of credit management at the server. Specifically, SEDA doesn't add any extra logic at the server while Breakwater and Protego perform all their admission control and AQM calculations at the server. This overhead is significant when the request execution time is very small. Note that increasing the number of clients from 1,000 to 10k can lead to performance degradation in SEDA with a larger size of incast [9]. This experiment shows that Protego can lead to goodput degradation in some scenarios where the CPU is bottlenecked. However, if the setting has any significant likelihood of mutex congestion, Protego can introduce significant benefits even when the CPU is bottlenecked (Figure 10).

6 Discussion

Fairness. Protego does not provide any mechanisms to ensure fairness between clients. For example, a client issuing more requests that require contended locks will get more failure messages because it faces a higher drop rate. However, it does provide flexibility for clients in their selection of replicas. A client can choose to send requests to a replica with a lower drop rate or distribute requests to multiple replicas to lower its drop rate. In this paper, we assume that the system as a whole has enough capacity to handle requests, relying on elastic resource allocation schemes like auto-scaling.

Generalizing Protego for other in-application congestion. An evaluation of DeathStarBench [14] revealed a challenging overload scenario where the tail latency of an upstream service (NGINX) spiked more than 10× while its CPU usage remains low due to the blocking network socket call used in HTTP. The delay introduced by such calls cannot be detected with the overload signal used in DeathStarBench (i.e., CPU Usage). Thus, the auto-scaler is never triggered to launch a new instance, causing high tail latency. Protego can be used to handle such overload scenarios where blocking calls (e.g., network or storage system calls) are the bottleneck. More specifically, the performance-driven admission control can back-pressure upstream services when it observes that there is no throughput improvement as load increases due to blocking calls. If the invocation of blocking calls by requests is unpredictable, it would require editing those calls to support ASQM. Furthermore, in a multi-tier microservice architecture, upstream microservices might be able to abstract calls made to downstream microservices as blocking calls, allowing Protego to be used to perform overload control over the entire microservice chain.

7 Related Work

Overload control. To avoid congestion collapse with receive live lock, an overload control system tries to bound the incoming requests or drop the request to prevent overload. Overload can be detected using several metrics. Breakwater [9] and DAGOR [34] use thread and network queueing delay. SEDA [32] and ORCA [20] use response time as a congestion signal. The way a system controls the overload also differs across these systems. Breakwater utilizes credit-based admission control with AQM. DAGOR utilizes priority-based admission control with AQM. SEDA adjusts the request sending rate at the client side. ORCA uses TCP-like window-based approach at the client side.

Flow Control. In TCP and eRPC [18], flow control advertises the size of the available receive buffer to clients to prevent receiving more packets than the network stack can accommodate. Akka [1] Stream has a similar but more flexible flow control mechanism where a server signals the maximum number of requests it can handle to the clients based on the remaining buffer size, the amount of idle resources, etc. The clients do not send more requests than the demand signaled by the server. Flow control is useful to avoid high latency when the CPU is the bottleneck. However, when a blocking synchronization becomes the bottleneck, it achieves either low throughput by underutilizing uncongested data paths or high latency with long queueing delay.

Measurement-based network congestion control. BBR [7] and PCC [12] employ mechanisms similar to Protego's performance-driven admission control. BBR explores the maximum network bandwidth by measuring the throughput with increasing window size. It concludes that the network bandwidth has reached its maximum value if it observes less than 25% of bandwidth increase with doubled window size. Unlike Protego, BBR does not utilize a performance-based approach to detect network congestion but to determine a parameter used for congestion control. In PCC, the system operator defines a utility function (e.g., TCP friendliness, latency, or throughput). PCC conducts multiple micro-experiments with a randomized set of parameters to find the configuration that achieves the highest utility. PCC-like algorithms require multiple rounds to find the best configuration, which slows down the reaction of the algorithm to the congestion. Unlike PCC, Protego deterministically modifies the credit pool size based on the measurement, which makes its reaction to congestion faster.

Auto-scaling. Auto-scaling [2,5,15,21,27,28] dynamically changes the amount of resources allocated to a service based on various signals including CPU usage, estimated demand, or custom-defined signals. It ensures that a service has enough resources to serve requests by allocating more resources when the chosen signal indicates that a load has exceeded a configurable threshold. Some auto-scalers [15, 21] let service operators specify the signal (e.g., response time, SLO violation, cost, etc.). More recently, machine learning models are used for auto-scaling. Facebook [5] and Google Autopilot [28] auto-scale resources based on the estimated demand learned from historical data. FIRM [27] uses system-wide performance metrics (CPU, Memory, Disk I/O, Network usage, or arrival rate) to train and predict which microservices require how much additional resources not to violate SLO. Auto-scaling mechanisms are useful with consistent overload over a long time scale, but it does not handle transient bursts in a load that happen over small timescales. Such bursts can be handled by Protego. In addition, auto-scaling alone is not enough to achieve both high throughput and low latency in the presence of lock contention as it does not provide any way to drop requests in a congested data path.

8 Conclusion

In this paper, we presented Protego, an overload control system that handles overloaded blocking synchronizations with performance-driven admission control and Active Synchronization Queue Management (ASQM). Protego's admission control decisions are based on measured throughput, admitting more load only if it improves throughput, admitting less load otherwise. To ensure low latency even for congested data paths, Protego sheds load by dropping requests at contended blocking synchronization points using ASQM. Our extensive evaluation of Protego demonstrates that it can effectively handle overload when combined with lock contention, achieving high goodput and low latency for a wide range of conditions. In particular, Protego achieves up to 3.3× higher goodput with 12.2× lower 99th percentile latency than state-of-the-art overload control schemes when applied to Lucene, a realistic search workload.

Acknowledgments

We thank our shepherd Marios Kogias and the anonymous reviewers for their valuable feedback, and Cloudlab [13] for providing us with infrastructure for development and evaluation. This work was funded in part by NSF grants CNS-2104398, CNS-2212098, CNS-2104398, and CNS-2212099; DARPA FastNICs (HR0011-20-C-0089); VMware and Google.

References

[1] Akka. https://akka.io/.

[2] AWS Auto Scaling. https://aws.amazon.com/autoscaling/.

[3] Breakwater implementation on shenango. https://inhocho89.github.io/breakwater.

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

[5] Throughput autoscaling: Dynamic sizing for Facebook.com. https://engineering.fb.com/2020/09/14/networking-traffic/throughput-autoscaling/.

[6] J. M. Banda, R. Tekumalla, G. Wang, J. Yu, T. Liu, Y. Ding, K. Artemova, E. Tutubalina, and G. Chowell. A large-scale COVID-19 Twitter chatter dataset for open scientific research - an international collaboration, May 2020. https://doi.org/10.5281/zenodo.3723939.

[7] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 2016.

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

[9] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay. Overload control for μs-scale rpcs with breakwater. In *OSDI*, 2020.

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

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

[12] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting congestion control for consistent high performance. In *NSDI*, 2015.

[13] 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.

[14] 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.

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

[16] R. Haecki, R. N. Mysore, L. Suresh, G. Zellweger, B. Gan, T. Merrifield, S. Banerjee, and T. Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *NSDI*, 2022.

[17] 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.

[18] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *NSDI*, 2019.

[19] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[20] 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.

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

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

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

[24] 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.

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

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

[27] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *OSDI*, 2020.

[28] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, et al. Autopilot: workload autoscaling at google. In *EuroSys*, 2020.

[29] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. Martin, M. McLaren, P. Chandra, R. Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *SIGCOMM*, 2020.

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

[31] B. van Klinken. Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

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

[33] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Characterizing facebook's memcached workload. *IEEE Internet Computing*, 2013.

[34] 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.