Dynamically Balancing Load with Overload Control for Microservices

RATNADEEP BHATTACHARYA, YUAN GAO, and TIMOTHY WOOD, George Washington University, Washington, DC, USA

The microservices architecture simplifies application development by breaking monolithic applications into manageable microservices. However, this distributed microservice "service mesh" leads to new challenges due to the more complex application topology. Particularly, each service component scales up and down independently creating load imbalance problems on shared backend services accessed by multiple components. Traditional load balancing algorithms do not port over well to a distributed microservice architecture where load balancers are deployed client-side. In this article, we propose a self-managing load balancing system, BLOC, which provides consistent response times to users without using a centralized metadata store or explicit messaging between nodes. BLOC uses overload control approaches to provide feedback to the load balancers. We show that this performs significantly better in solving the incast problem in microservice architectures. A critical component of BLOC is the dynamic capacity estimation algorithm. We show that a well-tuned capacity estimate can outperform even join-the-shortest-queue, a nearly optimal algorithm, while a reasonable dynamic estimate still outperforms Least Connection, a distributed implementation of join-the-shortest-queue. Evaluating this framework, we found that BLOC improves the response time distribution range, between the 10th and 90th percentiles, by 2 –4 times and the tail, 99th percentile, latency by 2 times.

CCS Concepts: • **General and reference** → **Performance**; • **Computing methodologies** → **Self-organization**;

Additional Key Words and Phrases: load balancing for microservices, closed-loop load balancers, performance optimization for synchronous services

ACM Reference format:
Ratnadeep Bhattacharya, Yuan Gao, and Timothy Wood. 2024. Dynamically Balancing Load with Overload Control for Microservices. *ACM Trans. Autonom. Adapt. Syst.* 19, 4, Article 22 (November 2024), 23 pages.
https://doi.org/10.1145/3676167

1 Introduction

Microservices have become increasingly popular due to a variety of advantages they provide like ease of deployment, continuous integration, independent development, and others. However, they also bring the network inside the architecture as the monolith is broken into multiple independently deployed pieces. In most current scenarios, microservices are deployed as containers in clusters managed by an orchestrator like Kubernetes [5]. A pattern related to container clusters that have

This work was supported in part by NSF Grant CNS-1837382.
Authors' Contact Information: Ratnadeep Bhattacharya (corresponding author), George Washington University, Washington, DC, USA; e-mail: ratnadeepb@gwu.edu; Yuan Gao, George Washington University, Washington, DC, USA; e-mail: ygao1@gwu.edu; Timothy Wood, George Washington University, Washington, DC, USA; e-mail: timwood@gwu.edu.

ACM 1556-4703/2024/11-ART22
https://doi.org/10.1145/3676167

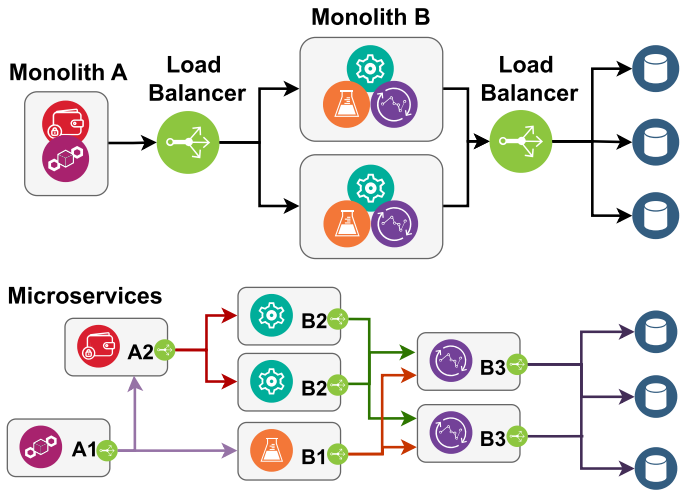


Fig. 1. A multi-tier application built from monolithic services (top) can be decomposed into microservice components (bottom), potentially improving development practices, but complicating the application topology. Sidecar load balancers (green circles) are deployed adjacent to each microservice component to route requests to downstream nodes.

also become popular is a move away from single-node centralized load balancers. Instead, client-side load balancers are deployed alongside each upstream service container as a "sidecar," as illustrated in Figure 1. An advantage of using this pattern is that the load balancer is removed as a single point of failure or performance bottleneck.

Many microservice deployments are managed by service meshes like Istio [4]. Istio uses Envoyproxy [3] for load balancing, which uses a **power of two random choices (P2C)** [23] version of the **Least Connection (LC)** algorithm. LC is based on **join-the-shortest-queue (JSQ)**, which has been proven to closely approximate the best possible load balancing algorithm by greedily selecting the backend that currently has the smallest queue of work [20]. However, JSQ's optimality depends on it being deployed in a centralized environment where all requests flow through a single load balancer, giving it a global view of backends' queues. In LC, a sidecar-based load balancer lacks this perspective, so it selects the backend to which it currently has the smallest number of open connections as the target for a request. In this case, the selected backend may not necessarily have the smallest queue since the policy only accounts for requests coming from the node attached to the sidecar.

In a microservice deployment, it is common for backend services to be shared by multiple upstream components, each of which may be replicated. In such a scenario, each upstream node sends only a small fraction of the total requests that each downstream node receives. This leads to a divergence between the actual load of the downstream nodes and the estimate of the load the upstream nodes have. As a result, the performance of the application can deteriorate quickly due to bad decisions made by such "local" algorithms.

In this work, we present BLOC,[1] which makes the downstream nodes a part of the decision-making without requiring expensive coordination. We compute the capacity of each service in terms of the number of requests one node of that service can handle while keeping end-to-end response times within the **service level objective (SLO)**. We then send each upstream node that

[1]This article extends our prior publication at ACSOS 2022 [12]. Source code available at [6]

we are currently interacting with "confidence chips" that will enable them to send requests in the future. The scheme also maintains some capacity for upstream nodes that the downstream is not interacting with at the moment but might still send a request. Downstream nodes use **active queue management (AQM)** to reject requests that push the number of active requests over their capacity. In response to such rejections, the upstream nodes back off for a predefined amount of time. Upstream nodes also use the P2C to reduce the likelihood of immediately selecting a downstream node that just rejected a request.

We make the following contributions in this article:

- —The design of BLOC, a distributed load balancing system that uses admission control, backpressure, and piggybacked server information to effectively balance loads, particularly in overload scenarios.
- —BLOC's architecture is fully distributed, requiring no coordination between replicas or centralized load balancers that can be a bottleneck or single point of failure.
- —BLOC's implementation uses ingress and egress proxies deployed as container sidecars, allowing its load balancing and admission control algorithms to be seamlessly integrated with existing applications without code modifications.

We implement BLOC as a Go-based proxy and deploy it in a Kubernetes cluster. Our evaluations show that BLOC can improve the response time distribution, from 10th to 90th percentile, by 2–4 times and the 99th percentile tail latency by 2 times.

2 Background and Motivation

Microservices and Sidecars: Microservices are a popular architecture pattern that breaks a monolithic application into multiple smaller services. It allows for shorter development time, faster deployment cycles, usage of different technology stacks for different parts of the application, swapping entire parts of an application, and continuous integration without any impact on the operation of the overall system.

Microservices are typically deployed in containers with an orchestrator framework like Kubernetes. Just as microservices are the smaller parts of a decomposed monolithic service, container orchestration frameworks take this a step further and allow each microservice to be decomposed into several containers, e.g., one container might hold the application business logic, while others run monitoring components and load-balancing proxies. These auxiliary containers are typically referred to as "sidecars," due to the way they are deployed adjacent to an application container and often process their incoming or outgoing requests. A group of application-specific and auxiliary containers that together form a logical service are grouped into a single namespace known as a "pod" by Kubernetes.

Since each pod can be replicated multiple times to scale up and down a microservice component, it is necessary to have load balancers that help route requests to the appropriate downstream node. The ability to easily glue together functional components has allowed for the move away from single-node centralized load balancers to distributed sidecar load balancers deployed as part of each pod. Each proxy sidecar thus handles load balancing all outgoing requests from the microservice component they are attached to across multiple downstream replicas. This distributes the load balancing work, giving a more scalable system, but it also means that each load balancer lacks the global view of a centralized approach.

Istio, Envoy, and LC: As an example of industry deployments of microservices networking we cite Istio [4] and Envoy proxy [3]. Istio is a popular example of what is known as a service mesh. A service mesh is a control plane that works with Kubernetes to deploy networking infrastructure throughout Kubernetes clusters. Typically, this is done through deploying a mesh of sidecar proxies,

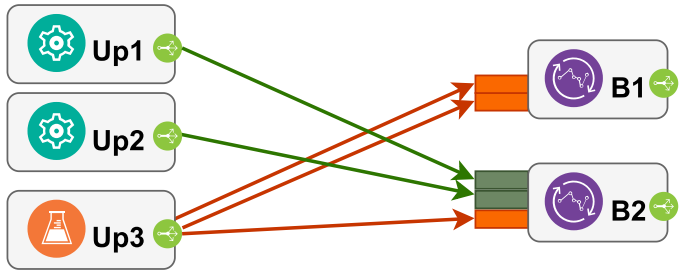


Fig. 2. LeastConn only has information about outgoing requests leaving a sidecar, not the actual queue lengths at the backend nodes.

like Envoy proxy with Istio, that provides the networking data plane and implements components like load balancing, service discovery, backpressure, and much more.

Envoy acts as both an ingress and egress proxy. The egress proxy implements load balancing and routing for any requests generated by the attached microservice component to downstream services. The ingress proxy intercepts all incoming requests from upstream services and is used for monitoring, security management, and so on. In our work we leverage this architecture so that downstream ingress proxies can provide feedback to upstream egress proxies, improving load-balancing decisions. Since our changes are only within the proxy, no modifications need to be made to the microservice applications themselves.

Sidecar proxies typically use traditional load balancing algorithms like the P2C version of LC. In this algorithm, the proxy randomly considers two possible downstream nodes and selects the one that has the least number of outstanding requests from the current node. Unfortunately, the node being picked might actually be more heavily loaded than others since the proxy is unaware of requests forwarded by the proxies in other pods.

LC and similar algorithms that rely only on a sidecar load balancer's local state can perform well when the number of service replicas is relatively low and workloads are evenly distributed across the upstream nodes. Yet in a microservice deployment, this may not be the case. For example, the applications provided by Deathstarbench, an open source collection of microservices, each contain between 21 and 41 unique microservices, each of which may be replicated multiple times [17]. Netflix, an early adopter of microservice architectures, was reported to have over 700 different microservices deployed over tens of thousands of virtual machines as of 2015 [8]. These massive arrays of microservices form complex topologies with shared services being accessed by many different types of upstream components. Further, there might be geographical constraints in large clusters leading to different client pods sending requests at different rates to the backends. In such a dynamic environment, workloads can easily become skewed, leading to an inaccurate local view of downstream node load levels.

LC Limitations: To see the intuition for why LC can perform poorly, consider the situation in Figure 2, Upstream Node Up3 has two outstanding requests to Backend B1 and one to Backend B2. The other two upstream nodes each have one outstanding request to B2. Thus the total number of outstanding requests at B1 is two while that on B2 is three. If a fourth request arrives at the load balancer of Up3, then the LC algorithm on Up3's LB will send the request to B2 instead of B1, which would have been the optimal solution. If a centralized load balancer was being used, this issue would not arise.

Generally, with a small number of servers and clients where the clients are all receiving roughly the same number of requests, this is not an issue since the relative equivalence in the number of clients and servers means that these discrepancies will be small so each sidecar's local view is a

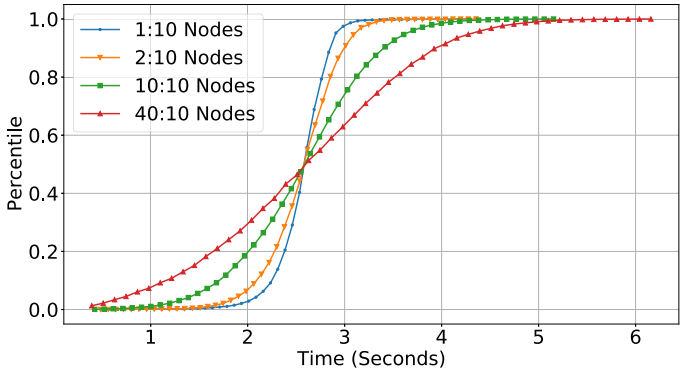


Fig. 3. Changing from 1 to 40 frontends causes a significant increase in the range of response times and tail latencies.

similar match to the global one. However, that may no longer be true when there is a large number of upstream nodes with different request characteristics than the downstream nodes.

To empirically measure this phenomenon, we deploy a frontend (upstream) and a backend (downstream) microservice and adjust the number of upstream nodes accessing a fixed set of 10 downstream replicas. In order to focus on the impact of load balancing across the downstream nodes, we configure the upstream service to be very lightweight and make the downstream service expensive (consuming a 250 ms service time). We deploy a custom sidecar load balancer, similar to Envoy, running the LC algorithm and using an HTTP load generator to stress test the system.

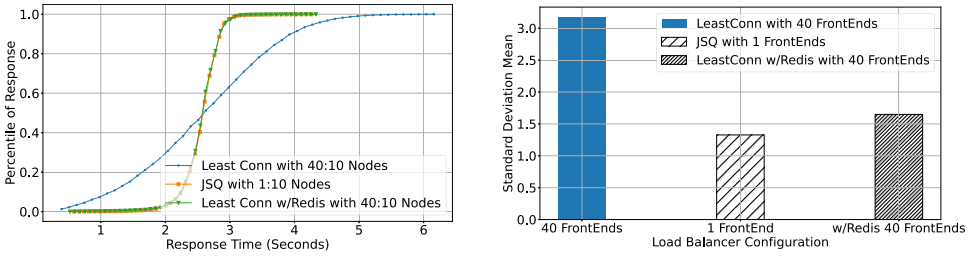
Figure 3 shows that as the number of upstream nodes increases, the response time distribution widens significantly. The case with only a single frontend (1:10) is representative of a traditional monolithic service deployment where a centralized load balancer sits between tiers of the application, while the other lines can represent distributed microservices. Interestingly, the median response time remains similar (about 2.5 seconds), but changing from 1 frontend to 40 frontends causes significant changes at the head and tail of the distribution. This result is somewhat unintuitive: one would typically expect adding more frontends to *improve* performance, not hurt it!

The explanation for these results is that the sidecar load balancers are making conflicting decisions due to lack of coordination—some requests are sent to very lightly loaded servers which are able to respond very quickly, while others queue up at overloaded servers, causing long delays. The impact can be quite large: the range between the 10th and 90th percentile increases by almost five times and tail latency degrades by more than 40% when going from 1 to 40 upstream nodes.

Diagnosing LC: We determined that there are two factors that cause the response times of the system to degrade by such a large amount:

(1) The metadata that each sidecar load balancer holds locally becomes stale much faster as the number of upstream nodes increases making the load-balancing decisions progressively worse, and
(2) A larger number of upstream nodes accessing backends with heavy requests can easily overload them, similar to the **transmission control protocol (TCP)** incast problem [14].

In orer to prove the first point, we deployed a Redis service in our Kubernetes cluster to provide a global view of the backend load. The Redis cache stored the active queue length of each downstream node. Before routing a request, a sidecar load balancer would fetch queue length data for all nodes from the caching service. The load balancer then updated the cache to increment the queue length



(a) Redis Load Balancer Performance

(b) Least Connection Mean of Standard Deviation

Fig. 4. Using Redis to provide a global view of the backend state makes the response time distribution nearly identical to having a single centralized load balancer (green and orange lines overlap) and similarly reduces the variation in load across backends.

for the selected downstream node. When receiving a response, the sidecar load balancer subtracted 1 for the downstream node that sent the response. This made the Redis service a global source of true backend queue lengths for all load balancers. With this simple addition of a caching service, we found that the overall performance of 40 upstream nodes is indistinguishable from that of using a single upstream node (Figure 4(a)).

To show the level of imbalance between downstream nodes when the number of upstream nodes increases, we measured the total number of requests sent to each downstream node at 2 second intervals. With this data, we plotted (Figure 4(b)) to show the standard deviation across the 10 backends during each interval, averaged over the entire experiment. We can see that the mean of the standard deviation of new requests received every sampling interval for the 40 frontend case is much higher than the 1 frontend node case.

We conclude that response time degradation is caused by the burstiness in the request profile which in turn is caused by inaccurate local data. This is exacerbated when backend requests are expensive (which is often the case) since even if all frontends send just one request to the same backend, they will cause it to be completely overloaded. Thus we must combine load balancing in the upstream nodes and overload control solutions in the downstream nodes to solve this problem.

Overload Control Approaches: Two general techniques to implement overload control are:

—Rate limiting, where upstream nodes purposefully slow their requests to prevent backends from getting overloaded; often this is guided by backpressure algorithms where the server lets the client know that the server is overloaded.
—Admission control, where downstream nodes preemptively drop requests to avoid excessive queueing; AQM algorithms try to intelligently drop requests or network packets to do this in a graceful way.

Unfortunately, naively applying backpressure has been shown to lead to system-wide hotspots and trick the system into upsizing or penalizing the wrong service [17]. Admission control, on the other hand, is extremely useful in controlling the number of requests on the server, but it does so at the expense of "goodput" [17] directly affecting user experience.

To see the impact of an AQM approach that drops requests once they exceed a response time bound, we repeat our experiment with 40 upstream nodes and 10 backend nodes. In Figure 5 we show the impact of setting a 3 second timeout vs. the default system with a 20 second timeout. Setting the timeout to a low value is similar to having an admission control system that will not allow any request into the queue if it takes longer than the timeout value. The results show that while the 3 second timeout puts a hard cap on the tail latency, it doesn't have much effect on the

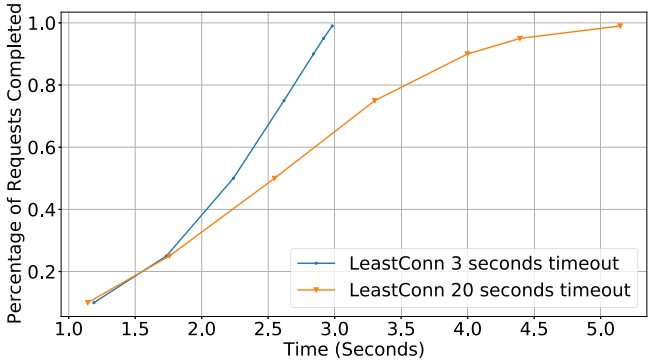


Fig. 5. Using AQM to drop requests early helps the tail, but not the head of the distribution, suggesting backends are still not evenly utilized.

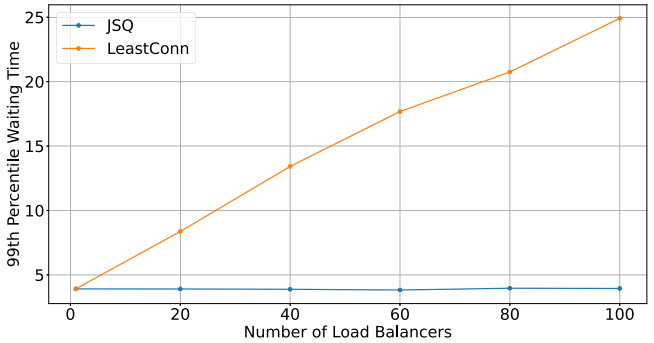


Fig. 6. Simulating JSQ vs. LC shows how waiting time rises with the number of load balancers.

head latency, indicating that the load is still not evenly distributed. Even worse, we find that the timeout-based system drops nearly 50% of the requests entering the system in order to achieve this and that the load variation across backends is not significantly improved.

3 LC Analysis

JSQ has been proven to be a nearly optimal load balancing algorithm [20]. In JSQ, the arrival rate into each queue is dependent on the length of the queue at that point in time, with the probability of a queue receiving a new request decreasing as the length of the queue increases. In the ideal case where all requests have the same cost, JSQ can guarantee that the most loaded server will have at most one more request in its queue than the least loaded server. For more complex scenarios, [11] provides an upper bound to the load imbalance between any two queues in the system. These properties ensure that load imbalances will be automatically corrected by JSQ by preferring backends with lower queue lengths. However, the JSQ algorithm assumes the load balancer has a perfect view of the backend queue lengths, which is only feasible in a centralized situation with a single load balancer. LC, on the other hand, operates in distributed client-side load balancers with incomplete knowledge of the backend queue lengths. In this section, we argue that LC cannot hold up the same guarantees as provided by JSQ and show why its performance degrades as the number of load balancers rises.

Simulation Analysis. We use a discrete event simulator to evaluate JSQ and LC performance. In Figure 6 we show how the 99th percentile of waiting time changes when simulating different

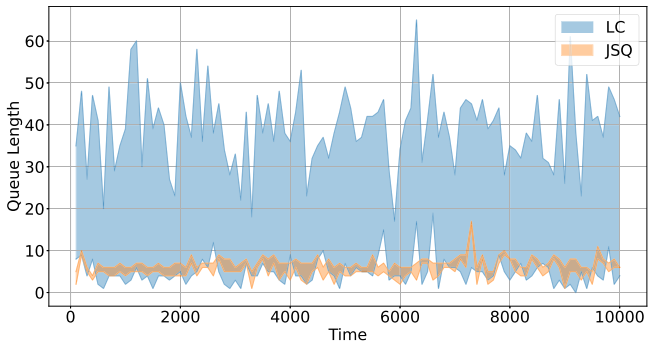


Fig. 7. LC (blue) sees both higher and more variable queue lengths than JSQ (orange) over time.

numbers of load balancers with 10 backends. For the JSQ case, we assume that all load balancers have perfect information about the backend server queue lengths, so performance is steady with no impact as we adjust the number of load balancers. On the other hand, LC sees a continual increase in tail response time as the number of load balancers rises.

LC performs poorly for two reasons. First, each LC load balancer only has visibility into the subset of requests that it receives. On average, we expect N load balancers to each receive $1/N$ fraction of the total load. If $N = 1$, then LC is equivalent to JSQ, but as N rises, each load balancer is only seeing a smaller fraction of the total load. Intuitively, we can expect LC to make worse decisions as N rises since it has less useful information to base decisions on. Even if each load balancer were to try to account for this by multiplying the load it is aware of by a factor of N, we cannot expect this to be perfectly accurate due to the randomness in how requests reach each load balancer. Balls and Bins analysis can be used to show that as N rises, the difference between the most loaded and least loaded load balancer (and thus the imprecision of their estimates) will increase based on $log(N)$ [25]. Using additional data from our experiment presented in Figure 6, we measure the difference in queue length between the server selected by our LC load balancer for each request and the server that would have been selected if we had used JSQ. We find that with 20 load balancers, the median queue length difference is 3, and this rises to 9 when we increase to 100 load balancers.

The second challenge for LC is that each load balancer may only have information about a subset of the backends. As N rises relative to the number of backends, it becomes more likely that a load balancer will have some backends for which it currently has no active requests, but this does not mean that the queue for those backends is empty. In these cases, LC simply picks a random backend with no active requests, leading it to behave more similarly to a random load balancer. To illustrate this challenge, we graph the queue length of each server over time under each load balancing scheme in Figure 7 when there are 100 load balancers. We plot the queue length on the most and least loaded server and fill in the area between them for each algorithm. The graphs show that LC (blue area) tends to have significantly higher queue lengths and that the spread between the most and least loaded backend is very large. In contrast, JSQ (orange area) keeps a much lower average queue length, and the queue lengths for all servers are correlated, implying that they are changing based on the burstiness of the incoming workload, not based on random decision-making. Note that here there are 100 load balancers, yet the queue lengths on each server are on average around 30 for LC, implying that most servers have zero active connections from most of the load balancers. As a result, LC's behavior is tending towards being uniformly random instead of being able to intelligently use queue lengths.

Information Theoretic Analysis. To understand why LC does worse than JSQ, we also perform an information theoretic analysis using the concept of Shannon entropy. We compare the entropy equations for JSQ and LC to show that LC has significantly higher entropy, meaning we expect greater randomness in its output which will lead to poorer decision-making. Let us consider the following situation with M backend servers, L frontend load balancers, X requests arriving at the system in total, and at most Y requests being seen by any of the L load balancers.

- When JSQ is used, the load balancers are aware of the exact queue size in each of the backend servers. This state can be represented as a vector, $S[M]$, where each entry $S[j]$ represents the queue size of the backend j, with the maximum value being X. Assuming all requests can be directed to any of the backends, the number of states in this case can be, $(X+1)^M$. If we assume, the probability of the system being in any state is uniformly distributed then:
  $H[JSQ] = \log_2((X+1)^M) \implies H[JSQ] = M * \log_2(X+1)$
- When LC is used, each of the L load balancers has its own view of the backends. The state in this case can be represented by a matrix, $S[L][M]$, where $S[i][j]$ is the ith load balancer's view of the jth backend. Considering each load balancer can only see at most Y requests, the number of states for each load balancer is $(Y+1)^M$. Across L load balancers then, the size of the state is $((Y+1)^M)^L \implies (Y+1)^{ML}$. Hence, the entropy of the system in this case is
  $H[LC] = \log_2((Y+1)^{ML}) \implies H[LC] = ML * \log_2(Y+1)$.

Comparing the above values, we can see that $H[LC] \geq H[JSQ]$ since $ML * \log_2(Y+1) \geq M * \log_2(X+1)$, and that the uncertainty of LC grows linearly with N. While our model of entropy is imperfect since not all states will be uniformly distributed, the significant difference in entropy suggests LC will tend to make decisions based on greater randomness than JSQ. Our model of entropy cannot be directly related to metrics such as the variability of response time seen by clients, but it helps explain why we should expect poorer choices in algorithms like LC. The lower entropy for JSQ means that JSQ considers much fewer states (estimates of backend load) than LC and all of these states are accurate. LC not only has to deal with a larger number of states, many of those states contain inaccurate information. This uncertainty leads to the higher entropy for LC, which increases the chance of picking a backend that is already heavily loaded, which in turn widens its response time distribution.

Fundamentally, BLOC alleviates the issue raised here by realizing that the number of requests actually pending on a backend, n_{actual} is almost always different than number of requests outstanding from any particular frontend/client to that backend, $n_{outstanding}$. BLOC simply introduces an upper bound, N, for each backend. The backend then sends a binary hint to the client indicating the potential availability of bandwidth to process further requests. In other words, the indicator says that $P[n_{actual} < N] > 0$. Since this indicator is dependent on the difference $N - n_{actual}$, the number of indicators sent out by the backend reduces as the outstanding requests in its queue rises.

4 System Design

In this work, our goal is to show that better load balancing can be done by combining AQM, backpressure, and a novel "confidence chip" distribution scheme that allows upstream load balancers to perform rate limiting in a self-organizing manner. The simple idea is that as requests flow downstream, server metadata flows upstream to inform better load balancing. We wish to keep each server under its maximum capacity, distributing the load evenly through the system, without incurring overheads from explicit messaging or requiring global coordination which cannot scale to large microservice deployments.

Our framework is divided into two logical parts. First, we use "confidence chips" as a form of load information to make the load balancers' decisions smarter. Confidence chips flow upstream

Algorithm 1: Backend Request Processing

```
1:  function HANDLE(w: http.ResponseWriter, r: http.Request)
2:      if Capacity and QueueLength + 1 > Capacity then
            Respond(TooManyRequests)
            Exit
3:      end if
4:      ProcessRequest(r)
5:      if Random() < QueueLength/(0.8 * Capacity) then
6:          chip ← 0
7:      else
8:          chip ← 1
9:      end if
10:     Return chip bit with response
11: end function
```

from backend nodes, piggybacked in the response headers of successful requests. Rather than just use local information like the number of active connections, the load balancers use the number of confidence chips they have received from different backends as an indication of how likely they are to be able to handle additional requests at this time. This allows the backend to help load balancers coordinate request rates, without requiring any direct communication.

Second, BLOC uses overload control to restrict the number of active requests on the downstream nodes. Downstream nodes preemptively reject incoming requests if they will cause them to become overloaded. However, rather than simply dropping the requests, the upstream load balancer takes this as a hint both to back off from this server for some time and to retry the request on a different server.

A final key design consideration is that we seek to avoid adding complexity to the overall system deployment or adding centralized services that cannot scale well to large systems. Thus we eschew approaches such as the Redis-based global coordinator described previously. A centralized approach would be difficult to deploy in practice and could incur high overhead in terms of latency and resource cost if every request needed to access it in a large-scale system. Just as importantly, we seek to support legacy code by incorporating BLOC into the sidecar proxies deployed alongside applications. This allows us to seamlessly add this functionality without any code modifications to the actual applications.

4.1 Confidence Chips

BLOC uses "confidence chips" as a way for upstream nodes to quickly learn which downstream nodes are above or below capacity. Each downstream node probabilistically returns a chip to upstream nodes piggybacked with the response header. An upstream node views the availability of a chip for a downstream as an indication that the particular node will have enough capacity to fulfill a request. The upstream spends a chip to make a request.

The probability of a downstream node returning a chip is related to how loaded the server is currently. This probabilistic distribution also serves as a hedge against requests from upstream nodes that the downstream is not talking to currently. We can reserve some capacity for upstream nodes for whom we do not have an active request right now but who might send a request to us soon. Also, since downstream nodes do not track chips granted, the probabilistic distribution protects the downstream from becoming oversubscribed.

Algorithm 1 details how BLOC processes each request. BLOC allows users to provide a capacity value in the configuration for the backends. This capacity value is used to limit the queue size in

Algorithm 2: Retry and Backoff

```
1:  function ProcessRequest(request)
2:      retry_num ← 0
3:      while retry_num ≤ total_retries do
4:          server = PickServer(servers)
5:          response = server.Handle(request)
6:          if response == TooManyRequests then                ▷ Server becomes "inactive"
7:              retry++
8:              server.chip ← 0                                              ▷ Reset "chips"
9:              server.RcvTime ← GetCurrentTime()   ▷ Calculate ResetInterval (line 5 and 9, algorithm 3)
    from here
10:         else
11:             break
12:         end if
13:     end while
14:     return response
15: end function
```

each individual backend. Lines 2–3 show that if adding another request would push the current queue size of the backend over the capacity configured then the request is rejected. Otherwise the request gets processed (line 4).

Lines 5–9 of Algorithm 1 show our "confidence chips" calculation. This is a binary valued indicator sent along with each response to inform upstream clients whether this backend has capacity available to handle more requests. In order to determine the value of the confidence chip, a random number, between 0 and 1, is compared against the ratio of capacity currently being used on the backend (line 5). A confidence chip returned with the response has a value of 1 (line 8) if the capacity ratio is lower than the random number. Otherwise the confidence chip has a value of 0 (line 6). Furthermore, the actual calculations are made against 80% of the defined capacity value to provide some protection against overloading. This means that as the queue length on the backend starts to approach 80% of its defined capacity, the backend will start sending less and less chips with a value of 1. It will send no chips with the value of 1 once queue length exceeds 80% of the defined capacity.

4.2 Client-Side Backoff and Retries

Algorithm 2 describes how each client sends requests to backends. We saw in Algorithm 1 that any request that pushes a backend's queue size beyond the defined capacity is rejected. In order to handle such cases, we also define the number of times each request can be retried. The while loop, lines 3–13 runs the overall algorithm. In line 4, we use the "PickServer" algorithm, described in the next section, to choose a possible backend to route the request.

If the server rejects the request, then it is reflected in the response from the server, line 5. When this happens, the upstream node increments the number of retries for that request (line 7), resets all chips for that backend (line 8) and sets the probe timer (line 9). At this point, that particular backend is considered to be "inactive."

If the request is successful or we have exhausted the number of allowed retries for the request, then the while loop (lines 3–13) is exited and the response is returned. Irrespective of the status of the response, we also note when the server received the last response. A request is dropped only if all retries have been exhausted.

Algorithm 3: BLOC for Server Selection

```
1:  function PickServer(servers)
2:      server1 ← RandomSelect(servers)
3:      // Either "ResetInterval" has passed since last response.        ▷ This backend can be probed.
4:      // Or the backend has a chip.                                    ▷ Backend is "active"
5:      while ElapsedTime(server1.RcvTime) ≤ ResetInterval and server1.chip == 0 do
6:          server1 ← RandomSelect(servers)
7:      end while
8:      server2 ← RandomSelect(servers)
9:      while ElapsedTime(server2.RcvTime) ≤ ResetInterval and server2.chip == 0 do
10:         server2 ← RandomSelect(servers)
11:     end while
12:     if server1.Reqs < server2.Reqs then   ▷ Select server with lower active requests from this upstream
13:         selectedServer ← server1
14:     else
15:         selectedServer ← server2
16:     end if
17:     if selectedServer.chip ≤ 0 and ElapsedTime(selectedServer.RcvTime) > ResetInterval then
18:         selectedServer.RcvTime ← GetCurrentTime()        ▷ Ensure this server is not probed too soon
19:     end if
20:     return selectedServer
21: end function
```

4.3 Server Selection

BLOC's algorithm for server selection, Algorithm 3, starts by picking two backends at random, lines 2–9. However, the algorithm takes care to ensure that the servers selected are considered "active" by the upstream, or that a reset interval has passed since an overloaded server was last contacted. An upstream considers a backend "active" if and only if the upstream has a chip for that backend. Lines 5 and 9 of Algorithm 3 test for this condition. Alternatively, lines 5 and 9 also test if the predefined reset interval has passed since a response was last received from that backend. If this condition is satisfied, then the upstream may select that server despite it not being "active." This is because we consider the state of the backend as unknown if at least the reset interval has passed since a response was last received. This backend, despite being inactive, is chosen because we want to probe inactive servers at regular intervals in order to update the upstream's metadata about them. However, when this happens we do not want to send another probe to the same backend too quickly. This is ensured by updating this backend's "RcvTime" to the current time (lines 17–19). Unless a chip for this backend is received, it will be rejected for all future iterations that run in the next "ResetInterval" units of time (lines 5 and 9).

When we have found two randomly selected servers that satisfy one of the conditions being tested in lines 5 and 9, we pick the one with the lower number of pending requests, as known to this particular client, (lines 12–16). Finally, this server is returned in line 4 of Algorithm 2. This is similar to the LC algorithm but BLOC ensures that the servers selected are unlikely to be overloaded.

4.4 Server Capacity

The capacity parameter plays an important role in determining the performance of a system. This parameter forms an upper bound on the size of the active queue of any upstream node in the cluster.

In the simplest case, a system administrator can specify a fixed Capacity value for each microservice based on its expected service time and SLO. The Capacity value times the service time gives an

upper bound on request queueing time. For simple services, this may be feasible, but for large-scale applications with many microservices, or deployments on heterogeneous hardware with different service costs, it may not be practical. Alternatively, BLOC has a simple method to dynamically estimate the capacity. This makes the system compute a cumulative average of the number of active requests in its queue for 30 seconds. The system then uses this average as the capacity value. We reset and recompute this average every 30 seconds. It admits all requests by default in the first 30 seconds where the capacity value is not defined yet. Improving this capacity estimate is an area needing further research.

5 Implementation and Experimental Setup

5.1 Customizable Microservice Generation

Our experiment testbed has been inspired by the Deathstarbench [17], which provides a set of premade microservice applications for system benchmarking. However, Deathstarbench is limited in its flexibility to only support its predefined applications. For BLOC, we built a customizable microservice generator that can define arbitrary microservice topologies [6]. Each microservice component is generated as a Python Flask service with a customizable request processing time and can optionally drive the input of many other services (fanout). The fanout is simulated by making parallel requests to each downstream service. Configuration files are generated to deploy the services in a Kubernetes cluster and automatically interconnect them to form the service mesh.

5.2 Sidecar Proxies

We also built the BLOCProxy reverse proxy framework [6] from the ground up to enable us to implement our algorithms with ease. The proxy handles ingress and egress traffic, allowing it to implement both admission control and load balancing. We redirect all incoming and outgoing traffic, except traffic to and from the proxy itself, to the proxy input and output ports, respectively. The proxy maintains a local directory mapping pods to service types as well as the Active and Inactive lists. During each request, the proxy can select the next endpoint by using a load-balancing algorithm defined through an environment variable along with other metadata.

Currently, the system implements the following load-balancing algorithms:

—Random
—P2C LC
—BLOC

The BLOC egress proxy modifies the HTTP headers generated by the microservice application to add a field indicating if confidence chips were generated. This is then interpreted and stripped out by the BLOC ingress proxy on the upstream node that generated the request. As of now, our implementation only supports HTTP1.1-based applications. However, our approach could easily be extended to support other protocols, such as gRPC, broadening the types of applications that can make use of our design without any code modifications to the applications themselves.

5.3 Control Plane

We also developed a simple control plane that uses the Kubernetes API to monitor live endpoints for each service that has been deployed. The proxies make REST API calls to the control plane pods, which run as a daemonSet in the Kubernetes cluster to populate their local service directories.

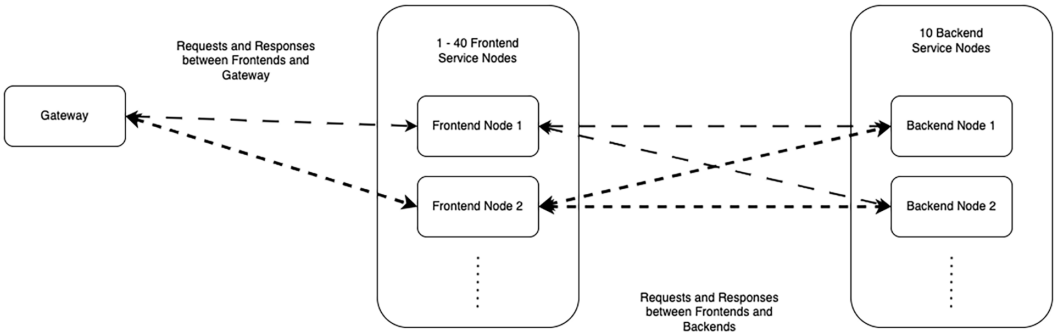


Fig. 8. Experimental setup.

5.4 Testbed Setup

In order to focus on load balancing between a pair of microservices, we use BLOC to run an application consisting of three layers of services (Figure 8) with the total number of pods ranging between 12 and 51 in a Kubernetes cluster running on 4 physical nodes. The Gateway layer consists of a single pod that acts as the ingress gateway. All requests to the cluster are forwarded to this gateway and are distributed to the frontend layer. The frontend layer, in turn, is variably sized. It scales between 1 and 40 pods. This layer sends all requests to the backend layer. The backend layer has a constant size of 10 pods. We overprovision the gateway and frontend layers so they will not become the bottleneck.

5.5 Workload

Most of our experiments have been conducted with a basic backend service that simply sleeps for 250 ms. However, we also test BLOC with backend service costs between 100 and 500 ms and there is provision for a variable service cost, which randomly selects a service cost uniformly in a range configurable through the environment variables.

For load generation, we use two open-source tools:

—hey [16], which is a closed-loop load generation tool that allows us to configure a concurrency for the requests we make
—a custom version of load test [1], that lets us define mean requests per second and generates load according to a Poisson distribution with this configured mean.

6 Evaluation

6.1 Experimental Setup

We ran our experiments on cloudlab [2] servers. A Kubernetes cluster was created with four Intel Xeon servers, each with 20 cores and 196 GB of memory. We then deployed our control plane that ran a pod on each of the servers. These pods form the service that is queried to get information about the backends of services running in the cluster. We create an affinity between our services and the physical nodes, such that the gateway and frontend services run on three of the four physical nodes. The fourth physical node can only schedule pods of the backend service. This was done to ensure that the performance of the backend pods was not interfered with. We use hey [16] as our closed-loop generator. We use the tool to send requests to the gateway for a fixed amount of time (5 minutes) where every request starts a new TCP connection.

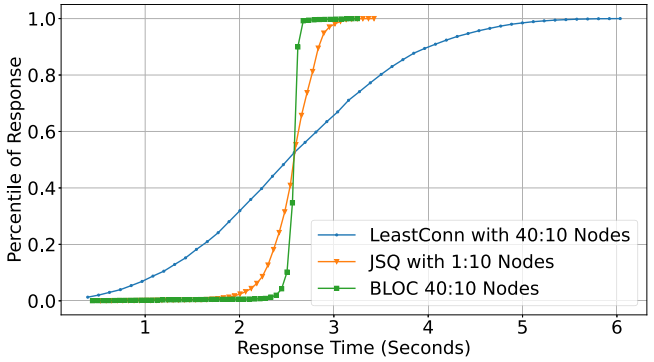


Fig. 9. BLOC (Cap = 10) provides a substantially tighter response time distribution by avoiding incast problems and applying careful admission control.

In most of our experiments we compare BLOC against LC with 40 frontends and a centralized JSQ approach. We implement JSQ by using a single node in the frontend tier running the LC algorithm (since LC with 1 node is equivalent to JSQ). This allows us to see how BLOC compares against an algorithm with perfect load balancing information as a nearly optimal baseline. However, in some of our experiments, the single frontend node for the JSQ case can become a bottleneck leading to worse performance than BLOC; we note these cases as they arise.

6.2 BLOC Overall Performance

We first compare BLOC, LC, and JSQ to evaluate our approach's impact on response time distribution. Figure 9 shows the response time **cumulative distribution function (CDF)** of each approach when the upstream nodes are accessing a shared pool of 10 backends. We compare 40 BLOC upstream nodes with a fixed Capacity parameter of 10 against LC with either 40 or 1 upstream node (which is equivalent to a centralized JSQ algorithm). While the median response times of all approaches are similar, there are dramatic differences in their response time distributions. When there are 40 upstream nodes, LC sees a very wide response time distribution, with the fastest 10% of requests finishing within 1 second and the slowest 10% of requests taking about 4 seconds, giving a 10–90%ile range of 2.77 seconds. On the other hand, BLOC maintains a very narrow response time window, with a range of 0.97 seconds. In fact, BLOC achieves a tighter window than LC running with a single upstream node (we ensure that the frontend is not the bottleneck in these experiments by using downstream backends with expensive service costs of 250 ms). Thus BLOC's distributed sidecars are able to effectively determine the relative loads on different servers, improving overall system utilization and providing very consistent response times.

Next, we vary the service cost of the backend nodes to understand the impact on load balancer performance. Figure 10 shows the improvement of BLOC over LC with forty frontends and 10 backends with a per request service cost ranging from 0.1 to 0.5 seconds. We use boxplots to show the median (black line), upper and lower quartiles (box edges), and 90%tile tail latency (whiskers). The response distribution of BLOC with 40 frontends is much closer to that of LC with a single frontend compared to LC with 40 frontends. The presence of outliers beyond the upper whiskers indicates a noteworthy observation: the 99th percentile tail latency of the BLOC when configured with 40 frontends, closely parallels that of the LC when it operates with only a single frontend, whereas that of LC with 40 frontends diverges from LC with 1 frontend as the service cost increases. BLOC's performance is similar to a centralized JSQ load balancer, although since it lacks perfect information it cannot maintain as tight a response time distribution. However, with a very low

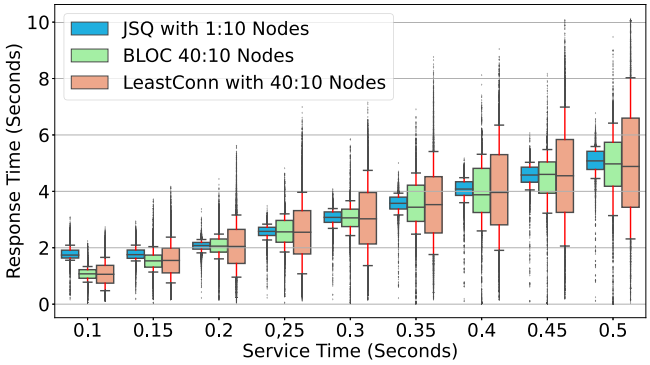
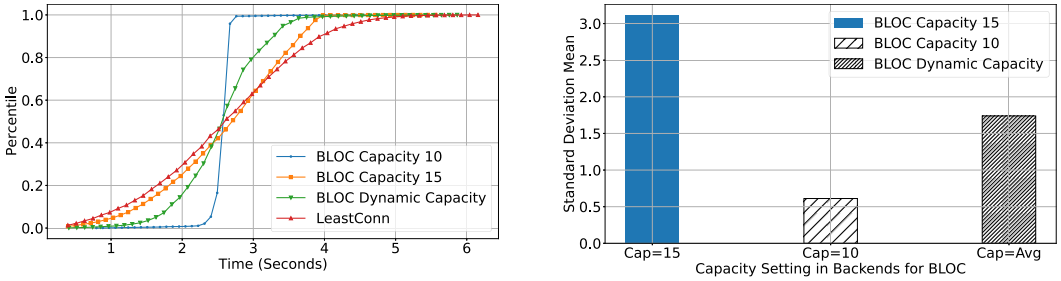


Fig. 10. Performance improvement with BLOC over JSQ and LeastConn for different service costs.



(a) Sensitivity of BLOC to Capacity

(b) Capacity Impact on Downstream Load Imbalance

Fig. 11. Sensitivity to capacity and impact on load imbalance.

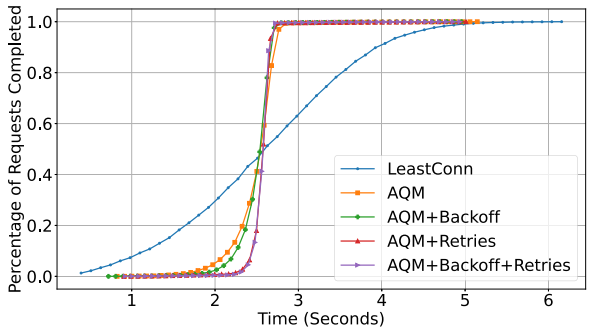
backend service time (0.15 and below), we see that the single JSQ load balancer is the bottleneck, resulting in worse performance than BLOC.

BLOC relies on its estimate of downstream capacity to control its AQM algorithm and allocation of confidence chips. To illustrate the impact of the Capacity parameter, we evaluate several fixed settings and BLOC's dynamic capacity estimation technique. Figure 11 shows the difference in performance when we used a Capacity of 10 (which our tests suggest is optimal for this configuration) Capacity of 15 (which tends to too aggressively overload servers), and our dynamic Capacity value based on the observed average. All of these approaches provide an improvement over LC, but setting an appropriate value gives a tighter bound.

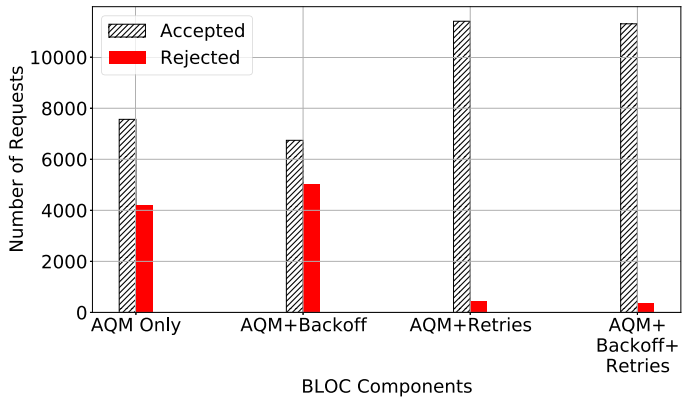
To further analyze the impact of Capacity, Figure 11(b) shows the level of imbalance on the downstream servers. This is measured by looking at the number of requests served by each node over time and calculating the standard deviation between them during each time interval; we then plot the mean of this variability. The results show that our hand-tuned Cap = 10 setting provides the greatest benefit, but that using the dynamic averaging approach also keeps the variance relatively low.

6.3 Benefits of Different BLOC Components

BLOC employs several techniques to avoid overload and keep downstream nodes balanced, so in this experiment, we quantify the benefits of each approach. In Figure 12(a), we show the CDF of AQM, AQM with BLOC techniques and LC. We can see that using AQM to drop requests that exceed the downstream node's capacity (without the rest of BLOC's functionality), provides a



(a) Impact on Response Time



(b) Impact of Admission Control

Fig. 12. The combination of all BLOC components ensures a tight response time distribution while minimizing request drops.

substantial improvement in response time. However, this only shows the performance of requests that are successfully processed, and as shown in Figure 12(a), AQM drops about 4,000 of the 12,000 requests sent during the experiment. Adding BLOC's backoff technique provides a further benefit to response time by reducing the chance that requests will be added to a long queue, however, it leads to an even higher drop rate. Adding support for Retries substantially improves the system, eliminating most of the drops and also providing a further reduction in the interquartile range. The final BLOC system that supports AQM, Retries, and Backoff provides a significant improvement to response times over LeastConn and reduces the number of failed requests by 22% (from 446 to 346) compared to the system with only AQM and Retries.

6.4 BLOC under Bursty Workloads

The prior experiments used the Hey benchmarking tool, which is a closed-loop load generator that seeks to continuously saturate the system. While this is an effective way to test the system on the brink of overload, it may not be representative of real web workloads which tend to have bursty periods of light and heavy load. In this experiment we use a customized version of loadtest [1], which is an open loop generator that can send requests at variable rates. While the official loadtest distribution follows a uniform distribution, our modified version sends requests following a Poisson distribution which gives a more realistic bursty arrival process.

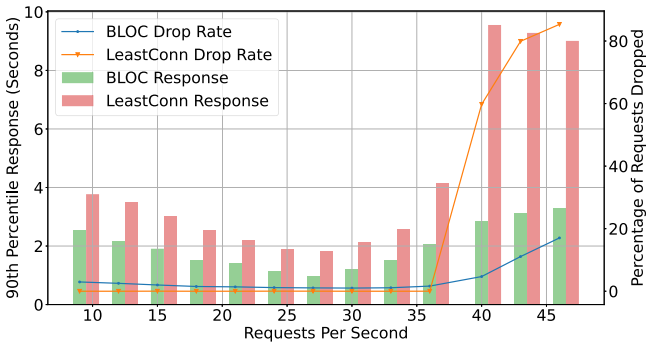


Fig. 13. 90th percentile response time (left-axis bars) and dropped requests (right-axis lines) with Poisson load generated at different rates.

In Figure 13 (left-axis bars), we show the performance of BLOC relative to LC with 40 upstream nodes and 10 downstream nodes under increasingly intense request rates. The results show that BLOC provides a substantially better 90th percentile latency, allowing it to support a much larger incoming request rate than LC. LC becomes overloaded with very poor performance after a workload of 35 req/sec, whereas BLOC is able to gracefully handle loads as high as 47 req/sec.

While BLOC provides a dramatic improvement in response time distribution at high load, it is in part due to its preference to drop requests that will cause excessive queuing. To evaluate this, Figure 13 (right-axis lines) shows the percent of requests dropped at each request rate for LC and BLOC. At lower request rates, BLOC still drops a small fraction of requests due to the bursty arrival pattern which can cause spikes in queue length.[2] Nevertheless, BLOC's drop rate is reasonably low, and even when facing an overloaded system at the highest request rate, BLOC drops only 16% of requests compared to LC dropping more than 80%.

6.5 Handling of New Resources

In this experiment, we start out by sending requests with loadtest [1] (open loop) to a cluster with 40 frontend services and 10 backend service instances. We increase the scale of the backend service by one pod a minute into the experiment (total 5 minutes) and plot the response times in Figure 14. The load being sent to the cluster makes the cluster slightly higher than the capacity of the cluster. We see that under this amount of load, LC's response times rise quite high very fast. Both LC and BLOC have a mean response time of 5,000 ms in the first minute. But even after adding a new pod, LC is still overloaded because it cannot effectively rebalance the queues on its servers. This would likely lead to LC needing to add another pod to stabilize the load. On the other hand, BLOC is able to utilize the newly added pod better and quickly reduces the mean response time to 3,400 ms, a reduction of 32% reduction in mean response time. This illustrates the importance of load balancing not just in reducing response times, but in reducing the total resources that must be assigned to the system.

6.6 A Real Variable Cost Backend Application

We use a real application that resizes images in response to incoming requests. This resizing application has a significant variance in its service cost, between 40 and 650 ms. Figure 15(a) shows

[2]In fact, we believe BLOC's drops may be due to a bug causing the gateway node to incorrectly drop requests even though the downstream nodes are not full.

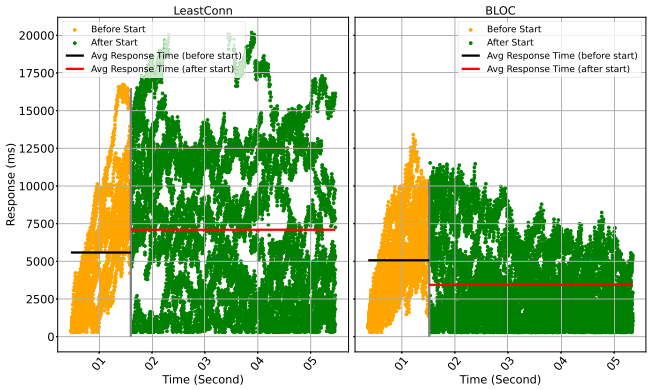
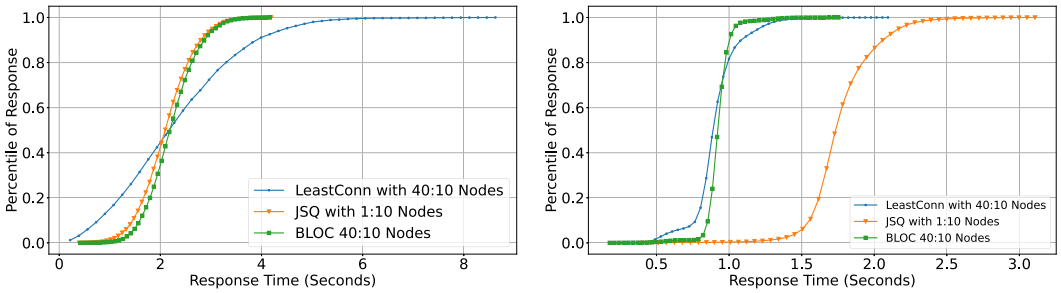


Fig. 14. BLOC and LC behavior when adding new resources to the cluster.



(a) High, Variable Cost Application (b) Low Service Cost Application

Fig. 15. BLOC, JSQ vs. LC for applications with different service costs.

that BLOC is still able to make good decisions to optimize the response time of each request, despite the large variance, matching the performance of a single LC frontend load balancer.

6.7 Low Backend Service Cost

When the service cost of the backend layer is low, the frontend layer becomes the bottleneck. Scaling up the number of frontend layers improves performance. In this experiment, we have an app that rotates images in response to incoming requests and the service cost for this backend varies between 10 and 80 ms. Figure 15(b) shows us that while both LeastConnection-with-40-frontends (LC40) and LeastConnection-with-1-frontend (LC01) have a narrow response time distribution, LC40 has a significantly better performance. Further, we see that BLOC-with-40-frontends has a response time distribution that is even narrower than LC40 and has a shorter tail than LC40.

6.8 BLOC vs. LC for a Complete Microservices Chain

In this experiment, we combine image resizing (highly variable service cost) and image rotation (low service cost) components into a single backend application. Our experience dictates that estimating the capacity of such a variable and multi-layered backend is extremely difficult. Figure 16 shows that BLOC can easily outperform LC in such scenarios.

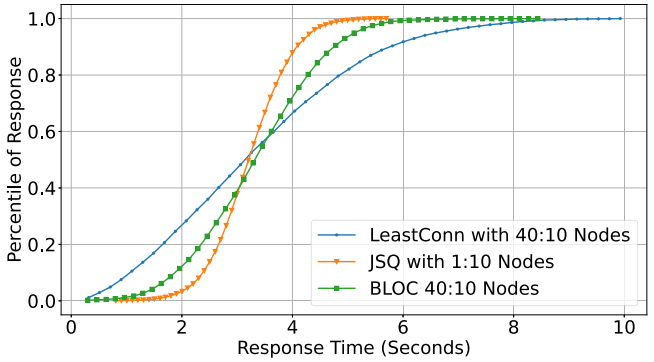


Fig. 16. BLOC, JSQ vs. LC for a complete chain.

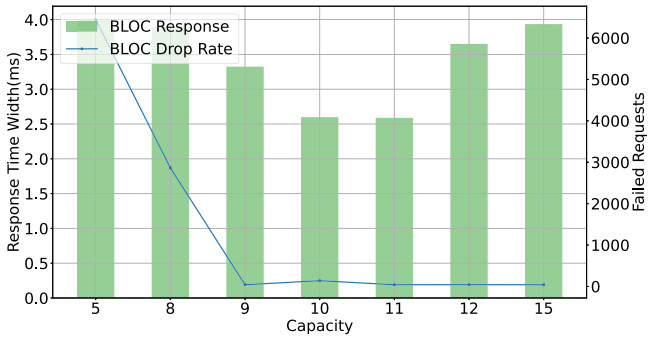


Fig. 17. Impact of different values of capacity 90th percentile response time (left-axis bars) and dropped requests (right-axis lines) with poisson load generated at different rates

6.9 Impact of BLOC Parameters

In Figure 17, we measured the width of the response time, the difference between 10 and 90 percentile responses, as we changed the static capacity values (left-axis bars). This result indicates that results deteriorate quickly as we move away from the optimal capacity value. We also measured how the number of failed requests changed as we changed the capacity value (right-axis lines). This result is quite intuitive in that as we increase the capacity values, the number of errors decreases.

6.10 BLOC Performance with Variable Service Cost

Finally, we present a measure of the performance of BLOC vs. LC under variable service cost. The service cost was chosen to be normally distributed with a mean of 250 ms and a 125 ms standard deviation, mimicking the parameters of the image processing application. In these experiments, refer to Figure 18, we saw that BLOC outperform LC.

7 Related Work

In this work, we have combined load balancing with overload control:

—*Load Balancing* approaches typically attempt to solve issues related to heterogeneity, performance, and uniform load distribution.
—*Overload Control* are admission control schemes that let servers control the rate at which clients can send requests.

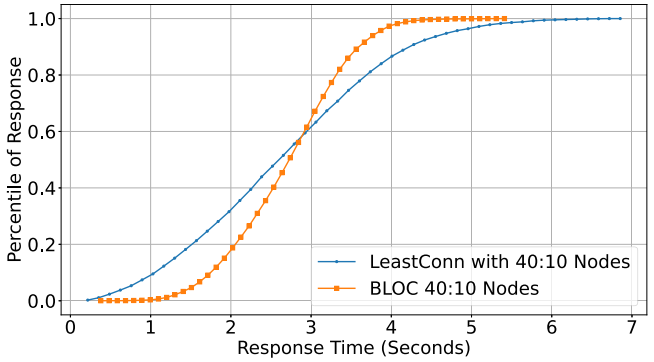


Fig. 18. Static BLOC vs. LC variable service cost.

7.1 Load Balancing

There is a wide range of work on load balancing for web [19] and cloud applications [21]. In microservices architectures, a load balancer plays an important role in terms of distributing workloads across multiple and various instances. There are two types of load balancers based on where the load balancer is placed, server-side and client-side. A study of tradeoffs between server-side and client-side load balancers is presented in [9]. We have based our work on evaluating the LC algorithm, which is a client-side approximation of the JSQ algorithm [20], often used in the microservices environment.

Research on the performance of load balancers, recently, has generally looked at topics like handling heterogeneity [18], uniform load balancing with consistent connections[10], and so on. While it has been established that with centralized load balancing it is not possible to significantly improve JSQ [20], we find that this result does not port over to distributed client-side load balancing. In this work, we tweak these load-balancing algorithms to be aware that their data might be stale and to take overcommitment into account. As far as we know, there is no other work that takes a look at the load-balancing algorithms in microservices networks.

7.2 Overload Control

Overload on a system can cause catastrophic failures[7] and the idea behind overload control is to shed any excess load before it consumes any resources[24]. In this work, we primarily use AQM to shed extra load. However, we do not want to sacrifice "goodput" [17] and as such build overcommitment and retries into the system. To our knowledge, there have been no prior attempts to use overload control toward load balancing in microservices.

7.3 Load Balancing with Server Feedback

There are two other systems [13, 15], that we know of, that incorporate feedback from the servers into how requests are distributed. In [13], the load balancer gets resource usage statistics from the servers to make its decisions. In our previous work [22], we have also used a similar feedback loop along with a mCeasure of the server capacities. In a distributed load-balancing architecture like microservices, however, this leads to convergence issues. In [15], the authors use overload controls to ensure no backends are overloaded. However, the low target service cost of [15] enables communication between all clients and all servers, in the form of registration messages, allowing for a complete flow of information. BLOC works with a much higher service cost which implies that it needs to load balance and protect against overload without any node-to-node messaging.

8 Conclusions

LC is a popular algorithm to balance load in microservices architecture and is based on JSQ, which has been proven to closely approximate optimal load balancing in a single node centralized load balancer. In the microservices world, the load balancer has moved from being a single centralized node to multiple instances each attached to a client service (upstream nodes). Here, LC finds it difficult to maintain the veracity of its metadata cache, which can atrophy quickly. This leads LC to make bad load-balancing decisions in aggregation. This in turn leads to a significant widening of the response time distribution and the lengthening of the tail.

In our framework, BLOC, we show that using overload controls judiciously overcomes this problem and is a far simpler solution than maintaining a distributed state. We also show that BLOC significantly improves overall performance. In our experiments, response time distribution improved by 2–4 times and tail latency did so by nearly 2 times. Overall, our results show that carefully combining overload controls with load balancing can lead to consistent response time despite the presence of a large number of frontends sending requests to a shared set of backends. BLOC is able to guarantee this performance consistency without sacrificing either user experience (by dropping requests) or adding to the overall load and complexity of the system (by sending metadata messages or using centralized caching services). We also show that BLOC can work with systems with a wide range of service costs and can handle variable service costs given an appropriate capacity estimator function. Our goal for BLOC is for it to become a more generic load-balancing algorithm able to support systems in different operating ranges and our work here indicates that possibility. Further work on BLOC can be directed at exploring various capacity estimator functions that would further BLOC towards this goal.

Dynamic capacity estimation is a significant area of research in both industry and academia using techniques from queuing theory, machine learning, and dynamic programming among others. In this article, we show that a simple running average method renders BLOC much more performant than state-of-the-art load-balancing algorithms. Thus investigating even more powerful methods in the context of BLOC is one research direction worth exploring.

We have created a repository to enable anyone to refer to and run the code to verify our results at https://github.com/MSrvComm/Experiments.

References

[1] [n. d.]. Custom Loadtest: Open Loop Poisson Load Generator. Retrieved from https://github.com/lyuxiaosu/loadtest

[2] [n. d.]. The Design and Operation of CloudLab: Flux Research Group. Retrieved from https://www.flux.utah.edu/paper/duplyakin-atc19

[3] [n. d.]. Envoy Proxy - Home. Retrieved from https://www.envoyproxy.io/

[4] [n. d.]. Istio. Retrieved from https://istio.io/latest/

[5] [n. d.]. Kubernetes. Retrieved from https://kubernetes.io/

[6] [n. d.]. Microservices Communication. Retrieved from https://github.com/MSrvComm/original-date:2021-11-05T02:23:34Z.

[7] Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, and Adam Stubblefield. 2020. *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems*. O'Reilly Media, 557.

[8] Amazon Web Services. 2015. AWS re:Invent 2015: A Day in the Life of a Netflix Engineer (DVO203). Retrieved from https://www.youtube.com/watch?v=-mL3zT1iIKw

[9] Marco Autili, Alexander Perucci, and Lorenzo De Lauretis. 2020. A hybrid approach to microservices load balancing. In *Microservices: Science and Engineering*, 249–269.

[10] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire, Panagiotis Papadimitratos, and Marco Chiesa. 2020. A high-speed load-balancer design with guaranteed per-connection-consistency. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation (NSDI' 20)*. USENIX Association, 667–684.

[11] Petra Berenbrink, Artur Czumaj, and Angelika Steger. 2000. Balanced allocations: The heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*. 745–754

[12] Ratnadeep Bhattacharya and Timothy Wood. 2022. BLOC: Balancing load with overload control in the microservices architecture. In *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS '22)*. 91–100. DOI: https://doi.org/10.1109/ACSOS55765.2022.00027

[13] Netflix Technology Blog. 2018. Rethinking Netflix's Edge Load Balancing. *Medium*. Retrieved from https://netflixtechblog.com/netflix-edge-load-balancing-695308b5548c

[14] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN '09)*. ACM, New York, NY, 73–82. DOI: https://doi.org/10.1145/1592681.1592693

[15] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µs-scale RPCs with Breakwater. Retrieved from https://www.usenix.org/conference/osdi20/presentation/cho

[16] Jaana Dogan. [n. d.]. hey - Open Loop Load Generator. Retrieved from https://github.com/rakyll/heyoriginal-date:2016-09-02T10:24:09Z

[17] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, 3–18. DOI: https://doi.org/10.1145/3297858.3304013

[18] Anshul Gandhi, Xi Zhang, and Naman Mittal. 2015. HALO: Heterogeneity-aware load balancing. In *Proceedings of the 2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 242–251. DOI: https://doi.org/10.1109/MASCOTS.2015.14

[19] Katja Gilly, Carlos Juiz, and Ramon Puigjaner. 2011. An up-to-date survey in web load balancing. *World Wide Web* 14, 2 (2011), 105–131. DOI: https://doi.org/10.1007/s11280-010-0101-5

[20] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. 2007. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation* 64, 9 (2007), 1062–1081. DOI: https://doi.org/10.1016/j.peva.2007.06.012

[21] Pawan Kumar and Rakesh Kumar. 2019. Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 120:1–120:35. DOI: https://doi.org/10.1145/3281010

[22] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2021. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. ACM, New York, NY, 168–181. DOI: https://doi.org/10.1145/3472883.3487014

[23] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104. DOI: https://doi.org/10.1109/71.963420

[24] Jeffrey C. Mogul and K. K. Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15, 3 (1997), 217–252. DOI: https://doi.org/10.1145/263326.263335

[25] Martin Raab and Angelika Steger. 1998. "Balls into Bins" - A simple and tight analysis. In *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '98)*. Springer, Berlin, 159–170.

Received 8 March 2023; revised 21 March 2024; accepted 18 May 2024