BLOC: Balancing Load with Overload Control In the Microservices Architecture

Ratnadeep Bhattacharya
Department of Computer Science
George Washington University
ratnadeepb@gwu.edu

Timothy Wood
Department of Computer Science
George Washington University
timwood@gwu.edu

***Abstract*—The microservices architecture has become ubiquitous in the cloud environment. It simplifies application development by breaking monolithic applications into manageable micro services that can be developed and deployed independently of the whole. However, the move from a monolithic or simple multi-tier architecture to a distributed microservice "service mesh" leads to new challenges due to the more complex application topology.**

A particular problem when automatically managing the performance of microservices is that since each service component scales up and down independently, it can easily create load imbalance problems on shared backend services accessed by multiple components. Traditional load balancing algorithms were designed for centralized load balancers sitting between a group of clients and a server farm. These algorithms, however, do not port over well to a distributed microservice architecture where load balancers are deployed client-side. In this paper 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.

We show that different service layers scaling independently can create unacceptably wide response time distributions and long tails, hurting client experience. This is because popular microservice load balancing algorithms, like Least Connection, only use a single component's view of the backend load to guide decisions. This limited perspective leads to an unevenly balanced system and the potential for incast problems where a large number of frontend components can easily overload a shared backend. BLOC uses overload control approaches like rate limiting, active queue management and backpressure to provide feedback to the load balancers. The load balancers react to this feedback with techniques like backoff and retries. We show that this performs significantly better in solving the incast problem in microservice architectures.

Evaluating this framework, we found that BLOC improves the response time distribution range, between the 10^{th} and 90^{th} percentiles, by 2 to 4 times and the tail, 99^{th} percentile, latency by two times.

***Index Terms*—microservices, loadbalancing, overload control**

I. 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, it also brings 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 [1]. A pattern related to container clusters that has 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.

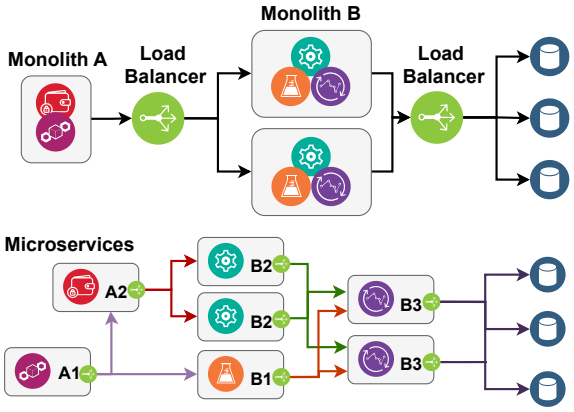


Fig. 1. A multi-tier application built from Monolithic services (top) can be decomposed into microservice components (bottom), potentially improving development practices, but complicating the application topology. Sidecar load balancers (green circles) are deployed adjacent to each microservice component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

Many microservice deployments are managed by service meshes like Istio [2]. Istio uses Envoyproxy [3] for load balancing, which uses a power of two random choices (P2C) [4] version of the Least Connection algorithm. Least Connection 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 which currently has the smallest queue of work [5]. 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 Least Connection, 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 that 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 SLO (Service Level Objective). We then send each upstream node that 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 to reject requests that push the number of active requests over its capacity. In response to such rejections, the upstream nodes backoff for a predefined amount of time. Upstream nodes also use power of two random choices to reduce the likelihood of immediately selecting a downstream node that just rejected a request.

We make the following contributions in this paper:

- The design of BLOC, a distributed load balancing system that uses admission control, backpressure, and piggy-backed server information to effectively balance load, 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 10^{th} to 90^{th} percentile by 2 - 4 times and the 99^{th} percentile tail latency by two times.

II. Background and Motivation

Microservices and Sidecars: Micorservices 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.

[1]Source code available at [6]

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 muliple 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 Least Connection: As an example of industry deployments of microservices networking we cite Istio [2] 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, like Envoyproxy 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, etc. 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 Power of 2 choices (P2C) version of Least Connection. 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.

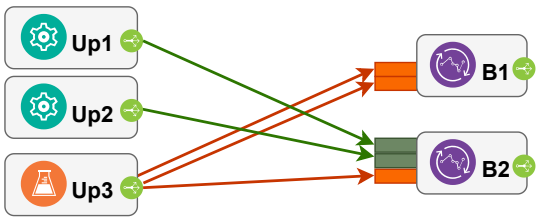


Fig. 2. LeastConn only has information about outgoing requests leaving a sidecar, not the actual queue lengths at the backend nodes.

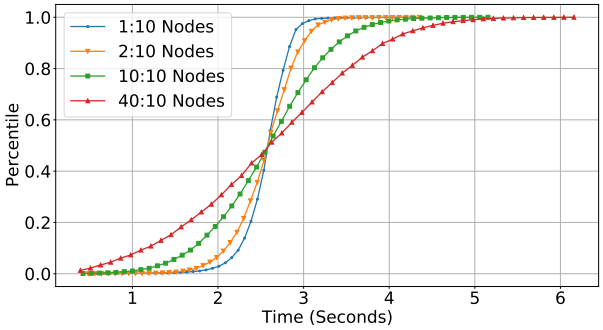


Fig. 3. Changing from 1 to 40 frontends causes a significant increase in the range of response times and tail latencies.

LeastConnection and similar algorithms that rely only on a sidecar loadbalancer'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 [7]. Netflix, an early adopter of microservice architectures, was reported to have over seven hundred 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.

Least Connection Limitations: To see the intuition for why Least Connection can perform poorly, consider the situation in Fig 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 now a fourth request arrives at the load balancer of Up3, then the Least Connection 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 mean that these discrepancies will be small so each sidecar's local view is a 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 to the downstream nodes.

To empirically measure this phenomenon, we deploy a pair of microservices and adjust the number of upstream nodes accessing a set of ten 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 msec service time). We deploy a custom sidecar load balancer similar to Envoy running the Least Connection algorithm and use an HTTP load generator to stress test the system.

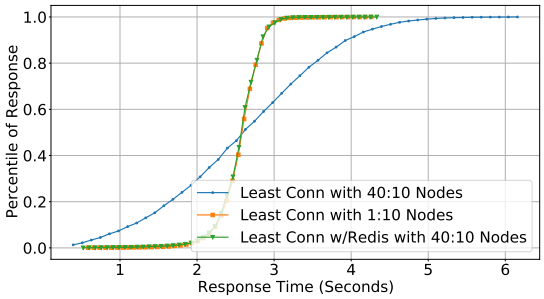
Figure 3 shows that as the number of upstream nodes increases, the response time distribution widens significantly. The case with only a single frontend (1:10) is representative of a traditional monolithic service deployment where a centralized load balancer sits between tiers of the application, while 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 10^{th} and 90^{th} percentile increases by almost 5 times and tail latency degrades by more than 40% when going from 1 to 40 upstream nodes.

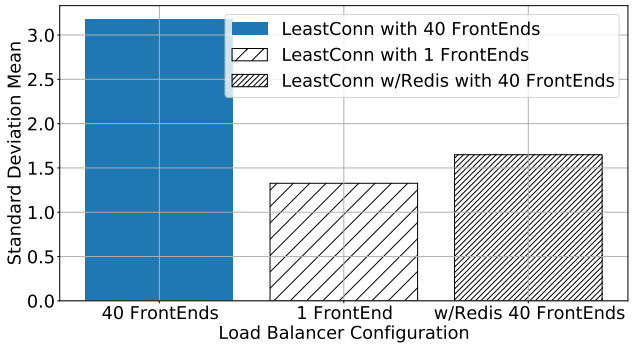
Diagnosing Least Connection: We determined that there are two factors that cause the response times of the system to degrade by such a large amount:

1) the metadata that each sidecar load balancer holds locally becomes stale much faster as the number of upstream nodes increase 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 TCP incast problem [9].

In order to prove the first point, we deployed a Redis service in our Kubernetes cluster to provide a global view of 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 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



(a) Redis Load Balancer Performance



(b) Least Connection Mean of Standard Deviation

Fig. 4. Using Redis to provide a global view of 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.

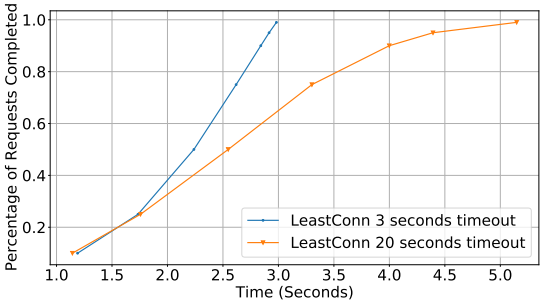


Fig. 5. Using AQM to drop requests early helps the tail, but not the head of the distribution, suggesting backends are still not evenly utilized.

queue lengths for all load balancers. With this simple addition of a caching service we found that the overall performance of a 40 upstream nodes is indistinguishable from that of using a single upstream node (Fig. 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 (Fig 4(b)) to show the standard deviation across the ten 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; Active Queue Management (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 [7]. 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" [7] 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 forty upstream nodes and ten backend nodes. In Figure 5 we show the impact of setting a 3 second timeout versus 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 they would take 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 head latency, indicating that 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.

III. BLOC: Design Principles

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 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. The first 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 backoff from this server for some time and to retry the request on a different server.

Secondly, we use confidence chips as a form of load information to make the load balancers' decisions smarter. Confidence chips flow upstream 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.

A final key design consideration is that we seek to avoid adding complexity to the overall system deployment or adding centralized services which 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.

IV. System Design

A. Confidence Chips

BLOC uses "confidence chips" as a way for upstream nodes to easily 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 amount of confidence 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 choose to 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 in the near future. Also, since downstream nodes do not track chips granted, the probabilistic distribution protects the downstream from becoming too highly oversubscribed.

Every node has a "capacity" value defined at the service level which represents the number of requests it can have in its queue while meeting a target SLO. When responding to a request, the nodes decide whether or not to issue a chip to the upstream node based on the following formula:

$$r = \text{uniform random number} \in (0, 1]$$

$$chip = \begin{cases} 0, \text{ if } r < \frac{q}{0.8 * cap} \\ 1, \text{ otherwise} \end{cases}$$

where q is the number of requests currently queued in the downstream node.

In this case, we guarantee that at least 20% of the downstream node's maximum capacity will always be reserved for the upstream nodes that are either not interacting with the downstream at the moment or have already been granted a chip. The probabilistic nature of the equation means that the number of chips generated during any given period of time will depend on the load on the downstream node during that time. Hence, when the server is lightly loaded it is more likely to send out chips whereas during times of higher loads it is sending lesser number of chips.

B. Active and Inactive Lists

Every upstream node individually maintains two lists of Active and Inactive downstream nodes and a predefined "probe" timer. Requests are preferentially routed to downstream nodes that are in the Active list using the P2C version of LeastConnection. The upstream nodes do not synchronize information on the lists with each other.

A downstream node is on the Inactive list if:

- the upstream node does not have chips for it, or
- the downstream node recently rejected a request from this node. In that case, the upstream node drops all chips it has for that downstream.

When an upstream receives a chip from a downstream, it adds that to the total chips for that downstream and moves it to the Active list if necessary. The upstream can then use those chips to make requests. If a downstream is already in the Active list, then receiving more chips does not affect its status.

If there are no entries in the Active list, then an upstream will send a request to a downstream on the Inactive list in order to update its metadata. This can only be done if the probe reset timer has expired for that downstream, as described below. However, probing a downstream does not immediately move it into the Active list. If there are no nodes in the Inactive list with a complete reset timer, then the request is dropped immediately rather than enter a queue which would violate its SLO. Thus in effect, the confidence scheme implements a form of AQM.

C. Client Side Backoff, Retries and Probes

If a node receives a request which would push its queue length above its capacity, then it will reject the request and return an HTTP 429 - "TooManyRequests" - status. This causes the upstream node to place the downstream node into

the Inactive list so it will back off the offending server for a predefined "reset" interval.

Rather than completely drop a rejected request, the BLOC load balancer will automatically attempt to retry the request on a different Active list server if possible. Each client can retry a request a predefined number of times. Requests that run through all retries at each layer are dropped.

If no Active servers are available, then the load balancer will consider servers from the Inactive list if their predefined reset interval has passed. However, if the client sends a request to a server in this manner, then that server is retained in the Inactive list and the probe timer reset. Only if the request is successful and the server returns a confidence chip will it actually be moved back into the Active list.

D. 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 can attempt to dynamically determine 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.

V. Implementation and Experimental Setup

A. Customizable Microservice Generation

Our experiment testbed has been inspired by the Deathstarbench [7], which provides a set of premade microservice applications for system benchmarking. However, Deathstarbench is limited in its flexibility to only supporting 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. Figure 6 shows an example of the type of service which the tool can deploy.

B. Sidecar Proxies

We also built the BLOCProxy reverse proxy framework [6] from ground up to enable us to implement our algorithms with

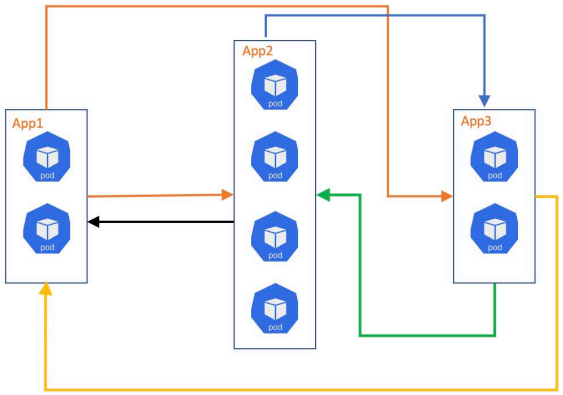


Fig. 6. Architecture Example with BLOC microservice generator

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 Least Connection
- 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 which can make use of our design without any code modifications to the applications themselves.

C. 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.

D. Test Bed 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 (Fig 7) 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

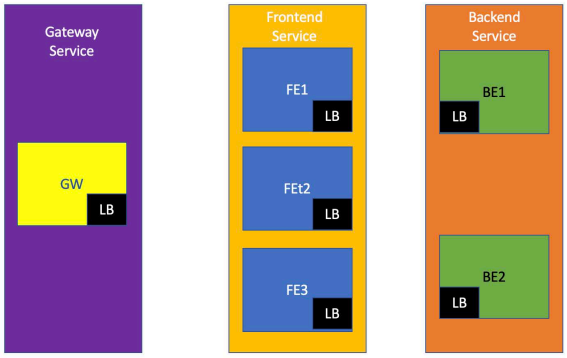


Fig. 7. Experimental Setup

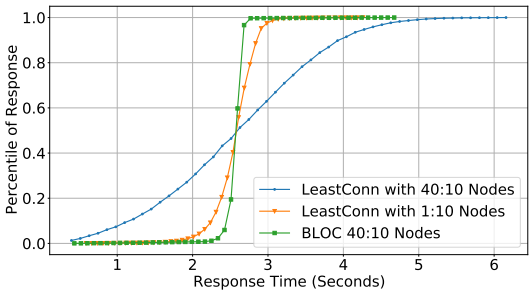


Fig. 8. BLOC (Cap=10) provides a substantially tighter response time distribution by avoiding incast problems and applying careful admission control.

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.

E. Workload

Most of our experiments have been conducted with a basic backend service that simply sleeps for 250ms. However, we also test BLOC with backend service costs between 100 to 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 [10], which is a closed loop load generation tool that allows us to configure a concurrency for the requests we make
- a custom version of loadtest [11], that lets us define a mean requests per second and generates load according to a Poisson distribution with this configured mean.

VI. Evaluation

A. Experimental Setup

We ran our experiments on cloudlab [12] servers. A Kubernetes cluster was created with four Intel Xeon servers, each with 20 cores and 196GB 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 3 of the 4 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 are not interfered with.

We use hey [10] 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 requests starts a new TCP connection.

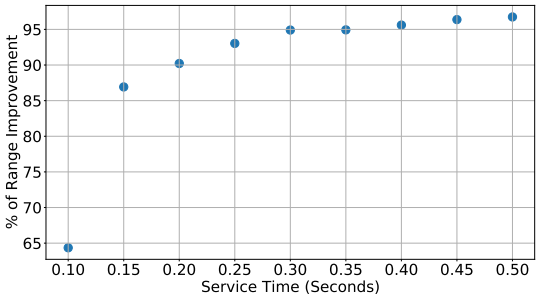
B. BLOC Overall Performance

We first compare BLOC and Least Connection to evaluate our approach's impact on response time distribution. Figure 8 shows the response time CDF of each approach when the upstream nodes are accessing a shared pool of ten backends. We compare forty BLOC upstream nodes with a fixed Capacity parameter of 10 against LeastConnection with either forty or ten upstream nodes. While the median response times of all approaches are similar, there are dramatic differences in their response time distributions. When there are forty upstream nodes, Least Connection 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 Least Connection running with a single upstream node (we ensure that the front-end is not the bottleneck in these experiments by using downstream backends with an expensive service costs of 250 msec). In addition to significantly improving response time, we find that BLOC slightly improves overall throughput of the system, successfully completing 0.93% more requests during the experiment compared to Least Connection. 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.

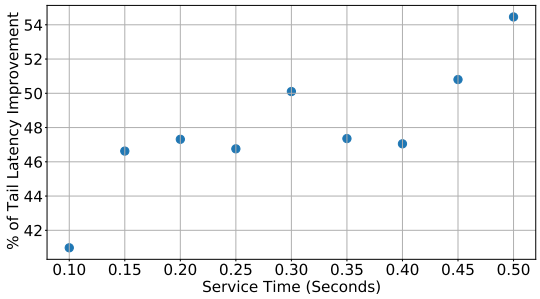
Next we vary the service cost of the backend nodes to understand the impact on load balancer performance. Figure 9 shows the improvement of BLOC over LC with forty frontends and 10 backends with a per request service cost ranging from 0.1 to 0.5 seconds. Even for the less expensive requests, BLOC provides a nearly 65% improvement in 10-90%ile Range, and a 41% improvement in 99%ile latency. For more expensive requests—where the impact of bad choices leads to even longer queuing time and front-ends must avoid incast issues—BLOC provides an even greater improvement.

C. Dynamic Capacity Estimation

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. Fig 10 shows the difference in performance when we used a Capacity of 10 (which our tests suggest is optimal



(a) Response Time Distribution Improvement



(b) Tail Latency (99 percentile) improvement

Fig. 9. Performance Improvement with BLOC over LeastConn for Different Service Costs

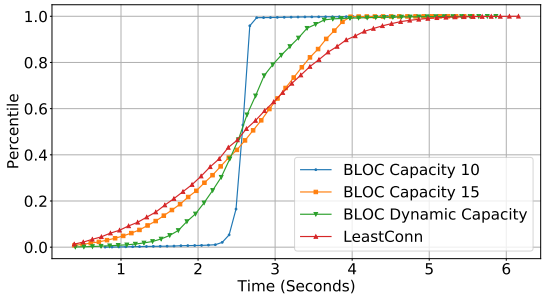


Fig. 10. Sensitivity of BLOC to Capacity

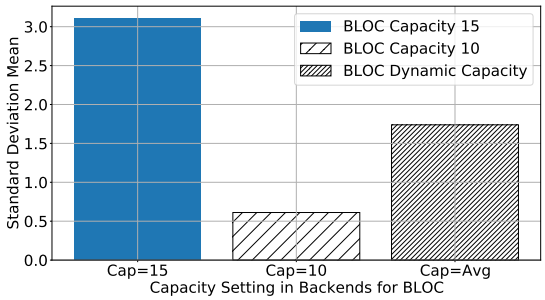


Fig. 11. Capacity Impact on Downstream Load Imbalance

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 LeastConnection, but setting an appropriate value gives a tighter bound.

To further analyze the impact of Capacity, Figure 11 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.

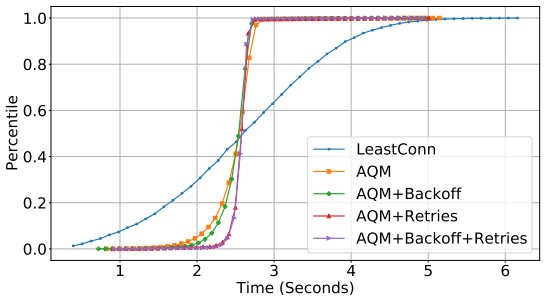
D. Benefits of Different BLOC Components

BLOC employs several techniques to avoid overload and keep downstream nodes balanced, so in this experiment we quantify the benefits of each approach. We use boxplots to show the median (red line), upper and lower quartiles (box edges) and 1.5*interquartile range (whiskers). In Figure 12(a), 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 substantial improvement in response time. However, this only shows the performance of requests that are successfully processed, and as shown in Figure 12(b), 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 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.

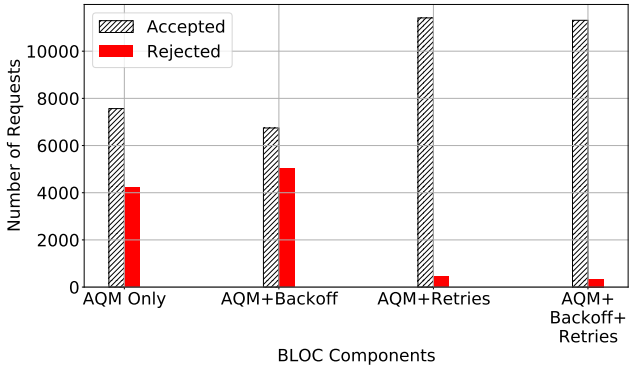
E. BLOC Under Bursty Workloads

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 [11], 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.

In Figure 13, we show the performance of BLOC relative to LeastConnection with forty upstream nodes and ten downstream nodes under increasingly intense request rates. The results show that BLOC provides a substantially better



(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

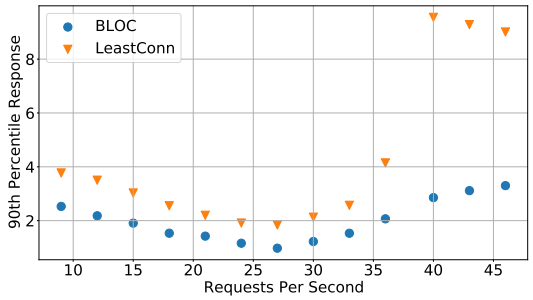


Fig. 13. BLOC Response Time (90th Percentile) Improvement with Poisson Load Generator

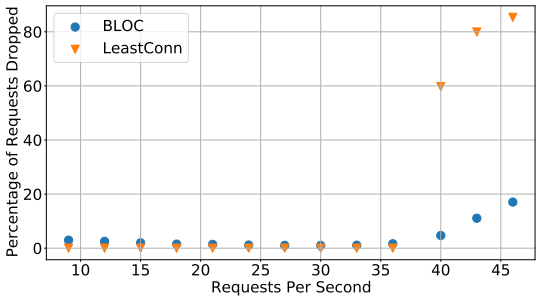


Fig. 14. BLOC Dropped Requests Percentage with Open Loop Poisson Generator

90th percentile latency, allowing it to support a much larger incoming request rate than LeastConnection. LeastConnection becomes overloaded with very poor performance after a workload of 35 req/sec, whereas BLOC is able to gracefully handle load 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 14 shows the percent of requests dropped at each request rate for LeastConnection 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 LeastConnection dropping more than 80%.

VII. 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.

A. Load Balancing

There is a wide range of work on load balancing for web [13] and cloud applications [14]. We have based our work on evaluating the LeastConnection algorithm, which is related to the JSQ algorithm [5], in the microservices environment.

Research on the performance of load balancers, recently, have generally looked at topics like handling heterogeneity [15], uniform load balancing with consistent connections [16] and so on. While it has been established that with centralized load balancing it is not possible to significantly improve JSQ [5], 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 are no other work that takes a look at the load balancing algorithms in microservices networks.

B. Overload Control

Overload on a system can cause catastrophic failures [17] and the idea behind overload control is to shed any excess

[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.

load before it consumes any resources [18]. In this work we primarily use active queue management (AQM) to shed extra load. However, we do not want to sacrifice "goodput" [7] and as such build overcommitment and retries into the system. To our knowledge there have been no prior attempts to use overload control towards load balancing in microservices.

C. Load Balancing with Server Feedback

There are two other systems [19], [20], that we know of, that incorporate feedback from the servers into how requests are distributed. In [19], the load balancer gets resource usage statistics from the servers to make its decisions. In our previous work [21], we have also used a similar feedback loop along with a measure of the server capacities. In a distributed load balancing architecture like microservices, however, this leads to convergence issues. In [20], the authors use overload controls to ensure no backends are overloaded. However, the low target service cost of [20] enables communication between all clients and all server, 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.

VIII. Conclusions

Least Connection is a popular algorithm to balance load in microservices architecture and is based on Join the Shortest Queue, 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, Least Connection finds it difficult to maintain the veracity of its metadata cache, which can atrophy quickly. This leads Least Connection 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 more simpler solution than maintaining distributed state. We also show that BLOC significantly improves overall performance. In our experiments, response time distribution improved by 2 to 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 have created a repository to enable anyone to refer to and run the code to verify our results at https://github.com/MSrvComm/Experiments.

Acknowledgements: This work was supported in part by NSF Grant CNS-1837382.

References

[1] Kubernetes. [Online]. Available: https://kubernetes.io/
[2] Istio. [Online]. Available: https://istio.io/latest/
[3] Envoy proxy - home. [Online]. Available: https://www.envoyproxy.io/
[4] M. Mitzenmacher, "The power of two choices in randomized load balancing," vol. 12, no. 10, pp. 1094–1104. [Online]. Available: http://ieeexplore.ieee.org/document/963420/
[5] V. Gupta, M. Harchol Balter, K. Sigman, and W. Whitt, "Analysis of join-the-shortest-queue routing for web server farms," vol. 64, no. 9, pp. 1062–1081. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0166531607000624
[6] "Micorservices communication," original-date: 2021-11-05T02:23:34Z. [Online]. Available: https://github.com/MSrvComm/
[7] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. Association for Computing Machinery, pp. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013
[8] Amazon Web Services, "AWS re:Invent 2015: A Day in the Life of a Netflix Engineer (DVO203)," Oct. 2015. [Online]. Available: https://www.youtube.com/watch?v=-mL3zT1iIKw
[9] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, ser. WREN '09. New York, NY, USA: Association for Computing Machinery, Aug. 2009, pp. 73–82. [Online]. Available: https://doi.org/10.1145/1592681.1592693
[10] J. Dogan, "hey - open loop load generator," original-date: 2016-09-02T10:24:09Z. [Online]. Available: https://github.com/rakyll/hey
[11] Custom loadtest: Open loop poisson load generator. [Online]. Available: https://github.com/lyuxiaosu/loadtest
[12] The design and operation of CloudLab : Flux research group. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19
[13] K. Gilly, C. Juiz, and R. Puigjaner, "An up-to-date survey in web load balancing," vol. 14, no. 2, pp. 105–131. [Online]. Available: https://doi.org/10.1007/s11280-010-0101-5
[14] P. Kumar and R. Kumar, "Issues and Challenges of Load Balancing Techniques in Cloud Computing: A Survey," vol. 51, no. 6, pp. 120:1–120:35. [Online]. Available: https://doi.org/10.1145/3281010
[15] A. Gandhi, X. Zhang, and N. Mittal, "HALO: Heterogeneity-aware load balancing," in *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 242–251, ISSN: 1526-7539.
[16] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire, P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'20. USENIX Association, pp. 667–684.
[17] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield, "Building secure and reliable systems," p. 557.
[18] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," vol. 15, no. 3, pp. 217–252. [Online]. Available: https://doi.org/10.1145/263326.263335
[19] N. T. Blog. Rethinking Netflix's Edge Load Balancing. Medium. [Online]. Available: https://netflixtechblog.com/netflix-edge-load-balancing-695308b5548c
[20] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload control for {µs-scale} {RPCs} with breakwater," pp. 299–314. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/cho
[21] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. K. Ramakrishnan, and T. Wood, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. Association for Computing Machinery, pp. 168–181. [Online]. Available: https://doi.org/10.1145/3472883.3487014