L3: Latency-aware Load Balancing in Multi-Cluster Service Mesh

Olivier Michaelis
TU Berlin
Berlin, Germany

Stefan Schmid
TU Berlin & Fraunhofer SIT
Berlin, Germany

Habib Mostafaei
Eindhoven University of Technology
Eindhoven, Netherlands

ABSTRACT

Microservice architectures and service meshes have become highly popular and face increasingly stringent scalability and dependability requirements. To achieve low-latency service execution and maximize performance, service providers of large-scale distributed systems deploy microservices geographically closer to their users in multi-cluster service mesh environments. However, inter-cluster service dependencies introduce additional latency, and effective load balancing across multiple replicas distributed across clusters is crucial. Addressing this challenge, we present L3, an adaptive latency-aware load-balancing mechanism for multi-cluster service meshes. We conduct extensive simulations on Amazon EC2, and our results of using the microservices of the DeathStarBench suite for three clusters show that L3 reduces the 99th percentile latency by 26% and 22% compared with round-robin and C3.

CCS CONCEPTS

• **Networks** → **Cloud computing**; **Network architectures**; • **Computer systems organization** → **Distributed architectures**.

KEYWORDS

Multi-cluster Service Mesh, Load Balancing, Latency, Microservices

ACM Reference Format:
Olivier Michaelis, Stefan Schmid, and Habib Mostafaei. 2024. L3: Latency-aware Load Balancing in Multi-Cluster Service Mesh. In *25th International Middleware Conference (Middleware '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3652892.3654793

1 INTRODUCTION

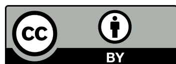
Service meshes are a particularly attractive part of microservice architectures for addressing critical communication challenges, including service location, secure connections, and communication failures of loosely coupled microservices in cloud applications [21, 35, 39]. They offer advanced features like rate limiting, load balancing, and telemetry [53]. To realize and simplify service implementations meeting the business or application requirements [33, 36, 40–44, 48, 50], service meshes were introduced as an infrastructure layer abstraction to proxy network traffic and handle the connectivity challenges in microservice architectures [32]. This abstraction allows developers to focus on writing business logic rather than getting involved in infrastructure-related complexities. Service meshes also empower developers to manage and monitor their microservices easily and add new security, observability, and traffic management features to their applications without significant changes to their code [53]. In contrast, container orchestration tools, such as Kubernetes [23], prioritize orchestrating compute units without offering additional functionalities.

As network applications evolve, there are often growing requirements to deploy them across multiple clusters, occasionally even across geographically distributed locations. This shift may originate from diverse considerations, including privacy, governance constraints, or the desire to enhance application availability and reliability by separating failure domains. While existing service mesh implementations like Linkerd [10] and Istio [17], alongside related research endeavors [40, 44, 48], have effectively optimized service execution latency within individual clusters, they often lack comprehensive solutions for service meshes extending across multiple clusters. For instance, Istio's *clusterLocal* approach [26] lacks dynamic feedback-based load balancing distribution adjustment.

Despite the availability of geolocation- and latency-based load balancing services from major cloud providers like AWS [28], Azure [29], and GCP [31], these solutions exhibit significant drawbacks. They typically rely on a Domain Name System (DNS)-based approach considering the incoming source IP and pre-existing latency measurements from associated IP address ranges to direct clients to presumed low-latency servers within their datacenter network. However, in microservice environments, where TCP connections can be long-lived, clients may continue sending HTTP requests over these connections regardless of whether the initially selected server remains the optimal choice. To address this issue, integrating a latency-aware layer into the service mesh architecture enables more granular Layer 7 load balancing.

While multi-cluster deployments of service mesh clusters can significantly reduce user-facing latency by content caching and content localization, they introduce several challenges. For instance, traffic among various service instances may traverse the wide-area network (WAN), and latency optimization becomes complex due to dependencies between services residing in different clusters [46]. This inter-cluster dependency introduces additional latency to the execution of services.

When multiple replicas of a microservice exist, load balancing is crucial in ensuring efficient resource utilization and optimal service performance. Selecting replicas, however, is challenging as the latency is difficult to predict and can vary. In particular, the geographically closest replica may often not be the one providing the best latency. We can identify three main reasons, all affecting latency and its variability: **First**, often WAN links are characterized by varying latency over time [38], which makes the system more challenging to optimize for latency. **Second**, the routing paths among the different clusters of the service mesh can change every couple of seconds [45], further increasing latency variability. **Finally**, the



This work is licensed under a Creative Commons Attribution International 4.0 License.

Middleware '24, December 2–6, 2024, Hong Kong, Hong Kong
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0623-3/24/12
https://doi.org/10.1145/3652892.3654793

combination of executing different services in microservice architectures can contribute to additional latency. For example, a slow database service can negatively add latency to the execution time. The latency penalty from a slow database can often be an order of magnitude higher than the network delay induced by geographical distribution. Therefore, balancing the load of service replicas is crucial and has a determinant role in reducing tail latency.

This paper explores latency-aware load balancing for services replicated in multiple clusters within a distributed service mesh. We introduce L3, a dynamic multi-cluster load balancer designed for Linkerd [16], a widely adopted service mesh. L3 adjusts traffic between multi-cluster service replicas (referred to as backends) based on data plane metrics such as latency, success rate, and requests per second (RPS). By prioritizing replicas with lower latency, guided by latency distribution metrics, L3 effectively reduces overall latency in the service mesh. Additionally, we use the success rate metric to guide traffic away from replicas with lower availability. We also introduce a rate control algorithm to distribute traffic evenly across all replicas during significant load increases. Leveraging exponentially weighted moving averages (EWMA) and PeakEWMA, L3 filters request metrics and assigns weights to service replicas across various clusters. We implement L3 using the standard interface within Linkerd, which can be easily adapted to other service mesh implementations, such as Istio. We present the results of an empirical evaluation of L3 on Kubernetes clusters deployed on AWS, utilizing the DeathStarBench suite [37] and report up to 26% and 22% reduction in tail latency compared with round-robin and C3 [49].

The remainder of this paper is organized as follows. We introduce the challenges of offering microservices in §2. The design of L3 is presented in §3. In §4, we explain the proof-of-concept implementation of L3 and its challenges. §5 reports on the performance of our evaluation under different scenarios. After putting our work into perspective with existing works in §6, we conclude in §7.

2 CHALLENGES OF SERVICE SELECTION

This section discusses challenges in offering microservices in geographically distributed systems. We first discuss issues related to the variation of latency and service demand when running a service in geo-distributed settings. Then, we underline the need for latency-aware load balancing in a multi-cluster service mesh and to perform clever service selection.

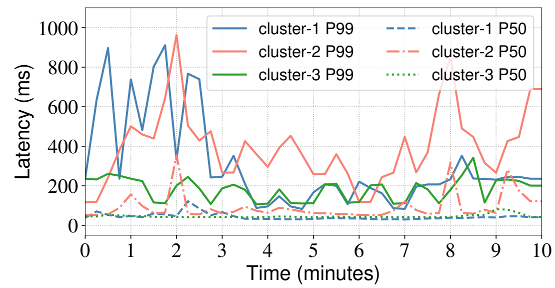
We further provide empirical evidence in the form of traffic captures of microservices from TIER Mobility's [18] production environment with over 200 microservices. The traffic captures consist of a randomly selected 10-minute period of a service deployed in several clusters, namely, *cluster-1*, *cluster-2*, and *cluster-3*.

2.1 Latency Variation

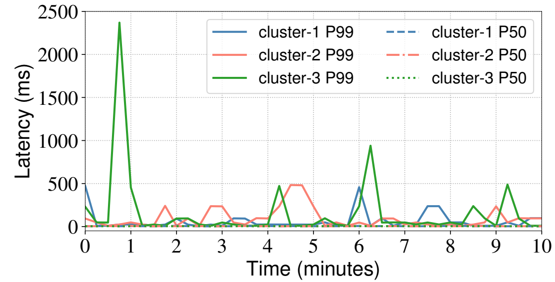
Service meshes typically permit interconnecting multiple sites, allowing for multi-cluster deployments in geographically distributed locations. The links connecting multiple locations can have transcontinental delays, and even the latency can vary over time [45]. Multi-tier microservices often have complex chains of dependencies on databases, caches, and other microservices, which can significantly influence the resulting latency [47]. Herein, the latency is the aggregation of both the WAN link latency and the service execution time on the servers of a cluster. In such scenarios, even infrequent drops in performance affect a significant proportion of all requests in large-scale distributed systems [34]. Furthermore, a slow microservice or a service that is on a critical path, i.e., a path with the longest chains of services— can degrade the overall performance of the system [52].

Our paper is motivated by the observation that instead of eliminating all sources of latency variability in large-scale distributed systems, it is more practical to use a latency-tolerant approach resilient to slow microservices. This can be achieved by steering traffic away from microservice replicas that experience performance degradation to replicas that perform well.

Figure 1 shows two scenarios from our traffic captures, namely, `scenario-1` and `scenario-2`, with the respective median and 99th percentile latency over a 10 minute period. We can observe that our services can experience tail latency that deviates considerably from the median latency. The median latency of `scenario-1`, for example, varies from 50 to 100ms most of the time (see Figure 1a), with some intermittent peaks up to 350ms for the service replicas in `cluster-2`. For the service of `scenario-2` shown in Figure 1b, this variation is much lower, with around 3 to 9ms. In both scenarios, the 99th percentile latencies of the services experience very high fluctuation– from 100 to 950ms in Figure 1a and from 10 to 2400ms in Figure 1b.



(a) Latency variations in `scenario-1`



(b) Latency variations in `scenario-2`

Figure 1: `Scenario-1` has a median latency of around *50ms*, with some spikes for `cluster-2` up to *300ms*. The 99th percentile latency is around *100ms* to *750ms*, with a very stable RPS of around *300* RPS. `Scenario-2` has a low 50th percentile latency with a 99th percentile latency around *10ms* to *100ms*, with some intermittent spikes up to over *2000ms* and fluctuating RPS between *50* and *200*.

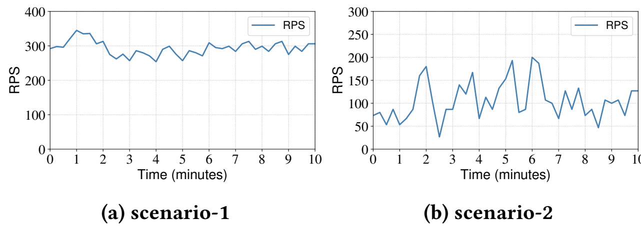


Figure 2: The RPS variation of `scenario-1` and `scenario-2`. RPSs slightly vary in `scenario-1` while this fluctuation is between 50 and 200 in `scenario-2`.

Given such a variance, solutions should dynamically adapt to latency variations in the geographically distributed environment to improve the system's overall performance. We investigate how load balancing can reduce the tail latency of service execution in a multi-cluster environment by exploiting insights from the latency of service execution.

2.2 Service Demand Variation

The number of services of a geographically distributed service mesh invoked by a service request propagating through the system can vary for several reasons. For example, the number of active users or their behavioral patterns. Therefore, it is crucial that the system can adapt gracefully to the demand variation without hindering performance, such as tail latency [47].

We now provide more insights from the service demand variation of our scenarios mentioned previously in Figure 2. We can observe that RPS slightly varies over time for `scenario-1` in Figure 2a. However, Figure 2b shows the service demand variation of `scenario-2` in which the number of RPS varies from 45 to 200 requests. We observe a high RPS variation in many time intervals, for example, between minutes 2 and 3. Therefore, designing an adaptive solution for an environment with such a dynamically changing number of RPS remains challenging.

3 L3 DESIGN

L3 is an adaptive load-balancing system designed to minimize the latency of microservices in a geographically distributed service mesh. Our design relies on the various traffic metrics offered by the data plane proxies, such as from Linkerd [14] and Istio [17]. These proxies provide a rich set of transport and application-layer metrics, such as the latency distribution of traffic destined for a specific service replica or the number of unsuccessful HTTP responses. We aim to leverage these metrics for the service selection in a service mesh to minimize latency. L3 directs the requests to the fastest replicas of a multi-cluster deployment. At the core of L3, there are three key components:

- **Metrics collector**: gathers the transport and application layer metrics when the traffic crosses the proxies and feeds them to the `weight assigner` component.
- **Weight assigner**: leverages the performance metrics, i.e., latency, RPS, and success rate, to assign a weight to each service replica.
- **Rate controller**: adjusts the weights given to each service replica to handle high RPS changes without overloading a specific one.

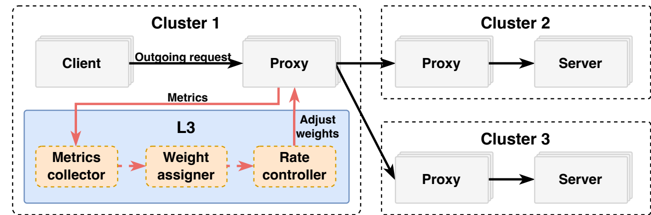


Figure 3: A possible architecture of L3 with three geographically distributed clusters over a WAN. Each cluster provides its performance metrics using a service mesh proxy.

Figure 3 depicts a simplified version of an L3 architecture with three clusters by putting the key components inside cluster 1. The clusters are interconnected through the service mesh, with the data plane consisting of many micro-proxies which are co-located with the microservice replicas and share the same kernel network namespace [25]. These proxies are injected into the network path and handle all TCP traffic transparently for the microservice. We note that in production deployments, L3 would most likely run on all clusters, and clients residing on different clusters do not transfer traffic to a central place for processing.

A set of clients in cluster 1 of Figure 3 generates requests for the microservice located in cluster 2 and cluster 3. The outgoing requests pass through their corresponding proxy in cluster 1, where they are forwarded to cluster 2 or cluster 3 depending on the weight distribution determined by L3. L3 collects the application layer metrics from the proxies and uses them to balance the cluster load, aiming to minimize tail latency. We provide the detailed integration of `metrics collector` in § 4. We now explain the `weight assigner` and `rate controller` components of L3 in detail.

3.1 Weighting Algorithm

With the `weight assigner` component, L3 assigns weights to the backends according to a scoring function by evaluating the collected performance metrics and converting them into weights. L3 leverages these weights to direct the requests to the backends that serve requests fast. These weights are then passed to the `rate controller` to determine the final weights by which the traffic is proportionally forwarded to the backends to optimize latency.

To reduce the tail latency, L3 considers the parameters that can affect the execution of requests, including the success rate, latency, RPS, and the number of in-flight requests of a backend. We now explain the role of each parameter of the `weight assigner`.

Accounting for the success rate. The success rate of a backend is the ratio of successful requests compared to failed requests and plays a critical role in load balancing for two reasons. First, it is generally preferred to forward a request to a backend that is more likely to process it successfully. Second, even a failed request can increase the tail latency in microservice architectures [34]. Thus, higher failure rates are penalized so that L3 can prioritize steering requests to healthier backends.

Symbol	Description
λ	Default latency
β	Decay coefficient
E_{now}	Current EWMA value
E_{prev}	Previous EWMA value
P_{now}	Current PeakEWMA value
P_{prev}	Previous PeakEWMA value
Y_{now}	Latest sample value
R_{s}	Success rate of a backend
R_{i}	Normalized number of in-flight requests
L_{s}	Latency of successful requests
L_{est}	Estimated latency
P	Penalty factor
c	Relative change
w_{b}	Weight of backend b
w_{μ}	Average weight of all backends

Table 1: List of Symbols

Accounting for the latency. The latency of successful requests is an important metric to consider when performing latency-aware load balancing, as it directly correlates with the user experience. To reduce the noise, L3 filters the request latency metrics with an EWMA. The latency of failed requests needs to be considered differently, as failed requests can often cause further problems for other services that were invoked as part of the request. Including the failure latency together with the success latency might result in misleading results.

Moving averages to filter metrics. L3 relies on metrics such as the request latency, RPS, and success rate. The metrics are filtered with EWMA and peak EWMAs to smooth out fluctuations in the sample data. Table 1 summarizes the symbols used in designing L3.

EWMA. The EWMA filter gives less weight to older samples by exponentially averaging the latest sample with all previous samples. We use EWMAs as a filter for the metric samples as shown in Equation 1, where t_{prev} is the timestamp of the previous sample and t_{now} is the timestamp of the current sample. We define Δt as the difference between t_{now} and t_{prev}. β is the decay coefficient and represents the degree of weight decrement of the moving average.

$$E_{\text{now}} = \begin{cases} \lambda, & \text{if } E_{\text{prev}} = \varnothing \\ Y_{\text{now}} \times (1 - e^{(\frac{-(\Delta t)}{\beta})}) + E_{\text{prev}} \times e^{(\frac{-(\Delta t)}{\beta})}, & \text{otherwise} \end{cases} \quad (1)$$

L3 maintains EWMAs of metrics such as the latency or the success rate for each replica of a service.

PeakEWMA. The PeakEWMA filter is an extension to the previously mentioned EWMA filter and originates from Finagle [13], a remote procedure call system by Twitter. PeakEWMA is designed to react quickly to sample spikes and recover cautiously. PeakEWMA can be used in place of EWMA within the algorithms, as its sensitivity can be helpful if a special focus is to be placed on worst-case samples.

Equation 2 shows how L3 computes PeakEWMA. If a new sample value is larger than the PeakEWMA value, the PeakEWMA value

Algorithm 1: Weighting Algorithm

```
input  : Array of backends B, EWMA for success latency,
         success rate, RPS, and in-flight requests for each
         backend
output : weight w_b for each backend b ∈ B to propagate to
         the service mesh
1  P ← a latency penalty factor for failed requests
2  foreach b ∈ B do
3      L_s ← b.EWMAP99
4      R_s ← b.EWMASuccessRate()
5      R_rps ← b.EWMARequestsPerSecond()
6      R_i ← 0
7      if R_rps ≠ 0 then
8          R_i ← b.EWMAInflightRequests() / R_rps
9      end
10     if R_s = 0 then
11         L_est ← L_s   /* Prevent division by zero */
12     else
           /* 1/R_s is the expected value for the number
              of tries until receiving a successful
              response.                               */
13         L_est ← L_s + P × (1/R_s − 1)
14     end
15     w_b ← 1 / ((R_i+1)^2 × L_est)
16     if w_b < 1 then
17         w_b ← 1
18     end
19 end
```

is reset to the new sample value. Otherwise, the new sample value is incorporated similarly to Equation 1.

$$P_{\text{now}} = \begin{cases} \lambda, & \text{if } P_{\text{prev}} = \varnothing \\ Y_{\text{now}}, & \text{if } Y_{\text{now}} > P_{\text{prev}} \\ Y_{\text{now}} \times (1 - e^{(\frac{-(\Delta t)}{\beta})}) + P_{\text{prev}} \times e^{(\frac{-(\Delta t)}{\beta})}, & \text{otherwise} \end{cases} \quad (2)$$

Weighting. Algorithm 1 shows how L3 assigns a weight for each backend $b \in B$. We consider a backend b and propose to use its 99th percentile latency to represent the tail latency of the latency distribution. As the latency of network communication can often be characterized by a log-normal distribution, we found the 99th percentile latency to be a good indicator representing outliers or extremely high latency values. To meet different requirements, L3 can be configured to utilize other percentiles, such as the 98th or the 99.9th percentile, as necessary.
We take the EWMA of the backend's 99th percentile latency for successful requests and assign it to L_s.

To estimate the latency of a microservice, we take the EWMA value of the backends success rate R_s, the RPS R_{rps}, and the number of in-flight requests divided by R_{rps} to receive the normalized number of in-flight requests R_i. In-flight requests are those that are

still in transit and for which a response has yet to be received. Then, we calculate the estimated latency L_{est} using a method proposed by Spotify [1] as follows.

$$L_{est} = L_s + P \times (\frac{1}{R_s} - 1), \quad (3)$$

where L_s is the 99th percentile latency latency of successful requests. The constant penalty factor, denoted as P, serves as a parameter to represent the impact a failed request has on the client. To accurately capture this impact, P should be set to a value corresponding to the round-trip time of failed requests, as perceived from the client's perspective. The round-trip time describes the duration until the client becomes aware of the failure and can issue a retry. Then, P can be multiplied by the expected value $\frac{1}{R_s}$ of the geometrically distributed number of requests a client has to send until a successful response is received. These two terms are then added together to obtain a latency estimate depending on the success rate.

We calculate w_b in Equation 4 as the weight for each backend based on the estimated latency L_{est} and the normalized value for the number of in-flight requests R_i. As the weight should get smaller for increasing latency, we chose the reciprocal function $\frac{1}{x}$ as a basis to model this inverse relationship. In the denominator, R_i is multiplied with L_{est} to encompass both the signal from in-flight requests and feedback from already completed requests. To account for cases without in-flight requests, R_i is incremented by one, resulting in $R_i + 1$. As in-flight or queued requests mostly influence the tail latency [34], $R_i + 1$ is squared to increase its significance. We used squaring, which presents a good trade-off between swiftly diverting traffic away from backends experiencing increasing latency and ensuring stability without causing excessive fluctuation.

$$w_b = \frac{1}{(R_i + 1)^2 \times L_{est}} \quad (4)$$

If L3 assigns a low weight to a backend, we run into the risk that the backend will not receive any traffic. This situation can result in information about the backend's performance being lost, as the system cannot collect metrics. To prevent this situation, we assign a weight to that backend that is high enough to direct enough traffic to the backend for the metric collection. A backend replica that becomes unable to serve traffic in a reasonable time frame, e.g., by being severely CPU throttled, should be handled on a different system layer. One approach is to use periodic health checks within the container orchestrator, which can temporarily take the replica out of the load balancing rotation until the service quality has sufficiently recovered.

3.2 Rate Control Algorithm

The `weight assigner` component of L3 prefers backends that can serve requests fast. However, when the number of requests significantly increases, it cannot ensure that the combined requests from all the clusters remain within the capacity of the fastest backends. Exceeding the capacity of the backends will result in poor performance and increase the tail latency [34]. Thus, we introduce the `rate control` component to address such situations to avoid directing a majority of the requests to a small number of backends.

Algorithm 2: Rate Control

input : Set of weights $w_b \in W$ for each backend b, EWMA of total RPS across all backend RPS_{EWMA}, and the last RPS sample RPS_{last}

output: Set of Weights W to propagate to the service mesh

1 $c \leftarrow$relativeChange(RPS_{EWMA}, RPS_{last}) `/* Relative change from EWMA to the latest sample value */`
2 $w_\mu \leftarrow W$.averageWeight()
3 **foreach** $w_b \in W$ **do**
4   **if** $c > 0$ **then**
5     $w_b \leftarrow w_\mu - \frac{w_\mu}{(1+c^2)^{\frac{3}{2}}} + \frac{w_b}{(1+c^2)^{\frac{3}{2}}}$
6   **else**
7   **if** $w_b \leq w_\mu$ **then**
8       $w_b \leftarrow \frac{w_b}{(1+2c^2)^{\frac{3}{2}}}$
9   **else**
10       $w_b \leftarrow 2w_b - w_\mu - \frac{w_b - w_\mu}{(1+3c^2)^{\frac{3}{2}}}$
11   **end**
12   **end**
13   **if** $w_b < 1$ **then**
14     $w_b \leftarrow 1$
15   **end**
16 **end**

L3 needs to adjust the traffic distribution resulting from the latency-based `weight assigner` to account for RPS fluctuations for two different scenarios with 1) increasing number of RPS and 2) decreasing number of RPSs. We now explain the reaction of L3 for each scenario.

To deal with an increase in RPS, L3 `rate controller` effectively distributes the incoming requests among multiple backends within a short time frame to prevent overwhelming any individual backend. This distribution strategy enables the cluster's autoscaling mechanisms to promptly scale up the faster backends in response. Once the autoscaling mechanisms have adjusted, the traffic share to the fastest backends can be increased again.

A decrease in RPS frees up backend capacity, allowing them to serve more requests. This situation enables L3's `rate controller` to shift proportionally more traffic to the faster backends opportunistically. Autoscaling mechanisms then have the chance to scale down the slower backends to increase resource efficiency.

When the RPS remains unchanged, the traffic distribution remains untouched by the `rate controller`.

Rate control. Algorithm 2 shows the pseudocode of the rating component of L3. We use W as the set of weights w_b of each backend, RPS_{EWMA} as the EWMA value of the total RPS across all backends, and RPS_{last} as the latest sample of RPS across all backends.

The algorithm initially determines the relative change c between RPS_{EWMA} and RPS_{last}. The relative change c indicates how much the RPS has increased or decreased in a recent time window, as RPS_{EWMA} suffers a time lag before it reflects a change in trend. Then, we iterate over each weight w_b and modify it depending on

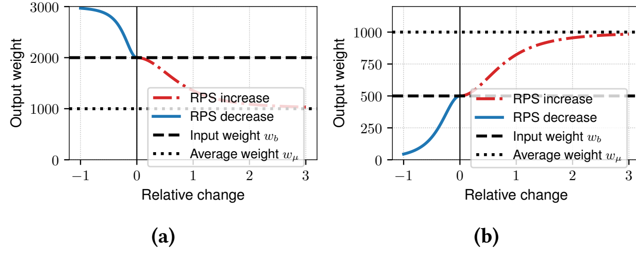


Figure 4: Weight adjustments made by the rate control algorithm of L3 for different situations (a) $w_b > w_\mu$ and (b) $w_b < w_\mu$.

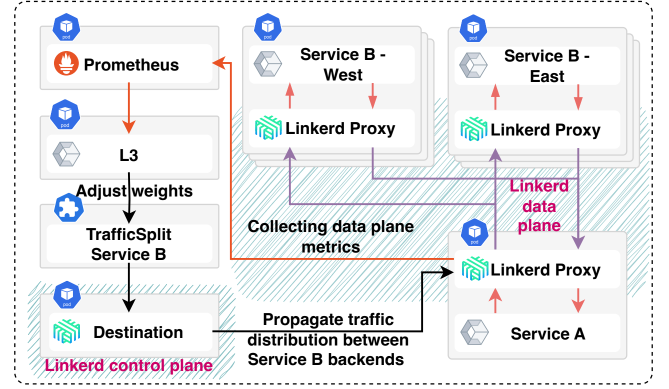


Figure 5: The integration of L3 with Kubernetes. Traffic from Service A is distributed between the East and West deployments of Service B according to the weights of the TrafficSplit. L3 adjusts the weights of the TrafficSplit based on the data plane metrics it retrieves via Prometheus. Linkerd's control plane propagates the weight changes to the Linkerd proxy of Service A.

whether: 1) the RPS has increased or decreased and 2) w_b is above or below the average weight of all backends w_μ. If the relative change c of the RPS is greater than 0 and the traffic has thus increased, the traffic has to be distributed more evenly across all backends. Thus, Equation 5 converges towards the average weight w_μ for larger positive relative changes c. The adjustment becomes smaller when w_b trends towards 0.

$$w(c) = w_\mu - \frac{w_\mu}{\left(1 + c^2\right)^{\frac{3}{2}}} + \frac{w_b}{\left(1 + c^2\right)^{\frac{3}{2}}} \tag{5}$$

Considering the case where the RPS decreases, and c is less than 0, w_b is decreased if it is less than w_μ and increased otherwise.

Figure 4a shows the weight adjustment for a backend weight $w_b = 2000$ and an average weight $w_\mu = 1000$. If the RPS decreases, the relative change c is below 0, then the weight increases opportunistically. For example, if the RPS is halved and therefore $c = -0.5$, the weight increases from 2000 to over 2800. The weight w_b is adjusted asymptotically towards w_μ for positive relative changes.

Figure 4b shows the weight adjustment as a function of relative change c for $w_b = 500$ and $w_\mu = 1000$. As $w_b < w_\mu$ applies, w_b is decreased for negative c and increased asymptotically towards w_μ for positive c.
Similar to algorithm 1, if w_b is under the threshold where the backend will continue receiving traffic, a higher weight is assigned to ensure continuous metric collection.

4 PROOF OF CONCEPT

We implement L3 as a microservice in 1800 lines of Go code, including tests. [1] It is built to run as a containerized workload in a Kubernetes cluster, managing user-defined objects declaring desired latency optimizations. L3 can be deployed with multiple replicas in a high-availability mode. Only a single replica acts as the leader and changes weights through a lease-based locking leader election mechanism. Information about the internal state of the controller and algorithm is exposed through Prometheus or OpenTelemetry metrics, respectively. Metrics can be collected with various open-source observability tools such as Prometheus, enabling human operators and other systems to infer the internal state at any point in time.

Figure 5 shows the integrated architecture of L3 with the Linkerd service mesh deployed on top of Kubernetes. The reference implementation of L3 follows the Kubernetes operator design pattern [3], which extends Kubernetes by providing custom dynamic management capabilities. L3 leverages the controller pattern [8] to implement control loops that continuously monitor the state of TrafficSplits and carry out changes as necessary.

TrafficSplits are part of the Service Mesh Interface standard [20] and allow users to define the traffic distribution between multiple target services in which each target service is called *backend*. The weight ratios between the weights of the backends define the distribution of traffic: a backend with twice the weight receives twice as much traffic. Weights can be any non-negative integer number. The role of TrafficSplits in L3 is to steer arbitrary portions of traffic destined for a Kubernetes service from one cluster to another. By building on top of the Service Mesh Interface, L3 can be used with Linkerd and easily adapted to other service meshes supporting the standard, such as Istio or Kuma [24]

We use one control loop to monitor the state of the TrafficSplits and handle the addition and removal of TrafficSplits and their target services. Another control loop fetches data plane metrics of the TrafficSplits every 5 seconds and updates the operator-internal state, such as EWMAs, with the latest metric samples. We apply the algorithms in §3 to update the weights of the TrafficSplits.

Metric collection. We collect the data plane metrics with `Prometheus` [22], a time series database, and L3 's implementation leverages these metrics for balancing the load. The Prometheus instance shown in Figure 5 periodically scrapes data plane metrics from all Linkerd proxies, by default every 5 seconds. Linkerd's metrics are represented with monotonically increasing counters, for example, with a counter for the total number of requests observed [14]. To convert these metrics into rate metrics like RPS or success rate, we need to calculate the per-second average of the metrics by calculating the rate of increase of the counters between two points in time. As we need to ensure that the period between

[1] The source code is available at https://github.com/oliviermichaelis/l3

these two points in time contains at least two time-series samples with a scrape interval of 5 seconds, we used a time window of 10 seconds.

The timing choice for collecting the telemetry information from Prometheus results in some limitations regarding the data freshness. The per-second averaged metrics are based on extrapolated data from a 5 seconds period, which could be a problem for especially spiky workloads where it is necessary to react more quickly. It is possible to reduce the scrape interval of Prometheus and thus also the time window for L3's queries, resulting in a measurable improvement. However, that also increases the load on Prometheus, especially in large clusters with tens of thousands of Linkerd proxies.

As shown in Figure 5, L3 queries Prometheus every 5 seconds for aggregated metrics of the TrafficSplit, more precisely: RPS, success rate, and latency, and adds them to the internal EWMAs, among others. The 5-second choice balances data freshness without overloading both Prometheus and Linkerd at a larger scale. Linkerd needs to push a new configuration to the affected sidecar proxies whenever the weights of the TrafficSplit are updated, so too frequent updates should be avoided at a larger scale.

EWMA default values. When initializing EWMAs, we assign a default value for each performance metric: 5 seconds for latency-related EWMAs, 100% for success rate EWMAs, and 0 for RPS-related EWMAs. We choose the initial values to prevent a new target service from overloading before establishing a meaningful baseline. Whenever L3 cannot retrieve metrics for a TrafficSplit — which happens after at least 10 seconds without any traffic — it starts converging toward the initial value of the EWMAs in small increments until new samples come in or the initial state is reached.

The decay coefficient β is configured such that the EWMAs have a half-life of 5 seconds for latency EWMAs and number of in-flight request EWMAs, and a half-life of 10 seconds for success rate and RPS EWMAs.

Resource usage. The L3 microservice itself is quite resource-efficient. It utilized less than 1.5% of a vCPU core during our benchmarks and maintained memory usage below 32MB. The Linkerd data plane proxies exhibit modest resource requirements, typically consuming memory in the low double-digit megabyte range and utilizing only a small percentage of the available CPU time of a vCPU core. An in-depth benchmark study [30] indicates that the proxy introduces latency increases of approximately 6*ms* for the median and 50*ms* for the 99th percentile at a rate of 2000 RPS. It is important to approach these figures cautiously, recognizing that they offer a rough estimate of resource consumption magnitude. Exact values depend on various factors, including the underlying hardware capabilities, software versions, and the requests per second (RPS). The Prometheus instance used approximately 0.5 GB of memory and up to 30% of a vCPU core. The scalability of this Prometheus setup could become a bottleneck in clusters with tens or hundreds of thousands of Linkerd proxies, especially paired with low scrape intervals. A sharded Prometheus setup with low data retention might be more suitable for such use cases. However, this choice was unnecessary for our scenarios.

5 EVALUATION

This section first explains the setup of our test environments and introduces our implementation of C3 [49] as one of the state-of-the-art solutions for comparison. Then, we perform some validation and robustness tests before evaluating the performance impact. In particular, we examine the influence of L3 on the latency and success rate.

5.1 Test Environment

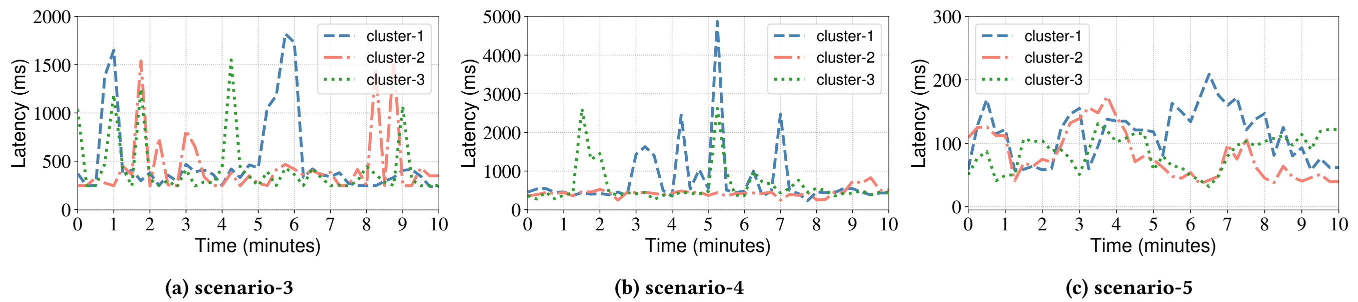
To measure the performance of our implementation, we built a geo-distributed system on public cloud infrastructure services from Amazon Web Services (AWS). The test environment consists of three Kubernetes clusters with cross-cluster communication enabled through a multi-cluster installation of the Linkerd service mesh. AWS Elastic Kubernetes Service [6] was used for the Kubernetes control plane, while the worker nodes were provisioned on AWS EC2 [5] `m6.xlarge` instances with 4 vCPU, 16GB of memory and up to 12.5 Gbps network speed.

The geo-distributed locations of the clusters were chosen so that the network delay between the clusters is as equal as possible and generally relatively low. Thus, a heavy bias towards a single location can be avoided and ensures that the network delay doesn't overpower the variability of the service latency. Choosing locations with a large network delay, e.g. from different continents, would most likely imply a heavy bias for the local cluster. In such a situation, a circuit-breaker-based failover mechanism triggered by outlier detection could be more suitable.

`Cluster-1` was provisioned in the `eu-central-1` (Frankfurt) region, while `cluster-2` and `cluster-3` were in `eu-west-3` (Paris), and `eu-south-1` (Milan), respectively. The network delay from `cluster-1` to both `cluster-2` and `cluster-3` is approximately 10*ms*.

Comparison algorithms. We compare the performance of L3 with the round-robin load balancing of Linkerd and C3 [49] as the state-of-the-art. Minor changes to the C3 algorithm had to be made to make it suitable for service meshes. In the following, we name noteworthy differences between the original C3 implementation and our adaptation. C3 takes load-balancing decisions at the level of individual requests, whereas L3 changes the aggregated traffic distribution of all requests within multiple clusters. We use aggregated metrics instead of per-request metrics to reduce the implementation effort of adapting C3 to our test environment. In scenarios where the success rate is suboptimal, L3 accounts for possibly necessary retry attempts and incorporates them into its weighting mechanism. While accounting for failed requests is critical in the context of HTTP requests, it is less relevant in C3's context of data stores. Since C3 lacks success rate optimizations, our adaptation of it similarly does not support it, which is relevant for the comparison with L3 in §5.3.2.

One further difference is C3's congestion-control-inspired backpressure mechanism. This mechanism allows clients to retain requests in a backlog queue in case a server's rate limit has been exceeded. Requests are kept in the queue until a server has regained the capacity to handle them. Due to the dynamic and heterogeneous nature of cloud server hardware, and the inherent uncertainty about the capabilities of their underlying hosts, most microservices lack



(a) scenario-3 (b) scenario-4 (c) scenario-5

Figure 6: The 99th percentile latency of running requests on remaining three different clusters, i.e., *scenario-3, scenario-4*, and *scenario-5*, of our production environment.

self-awareness of their capacity. This inability to consider the capacity of servers is also reflected in service meshes and, therefore, also in L3.

DeathStarBench. To measure the impact of L3 on the tail latency and compare it to other load balancing strategies such as round-robin and C3, we ran extensive benchmarks with the hotel-reservation application of the DeathStarBench [37] suite. This application consists of eight microservices and corresponding caches and databases, which offer users the functionality to reserve hotels. We deployed the entire application on each Kubernetes cluster, with a constant-throughput HTTP benchmarking client [27] deployed in one of the clusters. The benchmarking client then sends HTTP requests to the cluster-local frontend microservice to simulate multiple users logging in, reserving, and booking hotels. Outgoing requests from any of the microservices to other microservices are then distributed within all clusters according to the load balancing algorithm.

Given that the microservices were not initially designed with geo-distribution in mind and rely on stateful databases, distributing traffic across different clusters may not consistently yield correct responses. Despite these challenges, the system architecture remains valuable for assessing latency.

TIER Mobility. In contrast to the test environment of the hotel-reservation application, we need an environment to test scenarios such as with decreased success rates or vastly different latency variations. We choose five scenarios based on randomly selected time periods of 10 minutes of real-world latency and success rate data taken from more than 200 microservices from TIER Mobility's [18] production environment. Our scenarios involve requests exchanged between microservices, deliberately chosen to encompass a broad diversity of latency and request volume patterns. To measure request latency and construct test scenarios, we gathered latency traces generated via distributed tracing. We recognized that these traces encompass network delay, which could potentially skew results in our test environment due to existing topology-dependent network delay, so we excluded network delay spans from the traces. As a result, we focus solely on extracting service execution latency data. Nevertheless, we present representative results of L3 for the selected scenarios. Figure 6 reports the 99th percentile latency of our traces for *scenario-3*, *scenario-4*, and *scenario-5*. As the microservices from which the production data was taken have very high availability and no significant failure rate except during outages, we created scenarios `failure-1` and `failure-2` by injecting artificial failure into the previously mentioned scenarios.

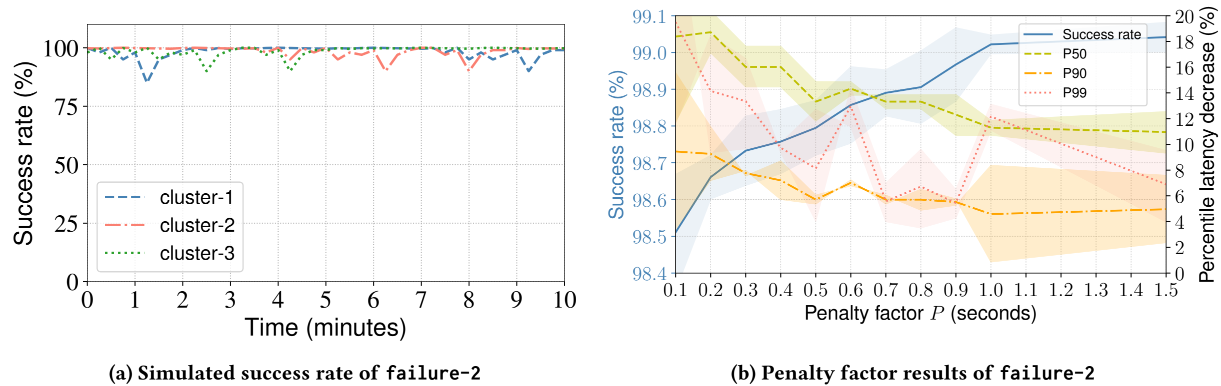
For the benchmark, a benchmark coordinator and an HTTP load generator are deployed alongside L3 in `cluster-1`. In each cluster, an HTTP/2 REST API workload is deployed with three replicas per cluster. The benchmark coordinator instructs the load generator to generate requests according to the scenario's request volume. The load balancing algorithm distributes these requests to the API workload. The benchmark coordinator instructs the API workloads via a RabbitMQ message queue to artificially delay HTTP responses according to the scenario's latency distribution and simulate failed requests.

Before starting a scenario, the coordinator performs a short warm-up period to populate caches and establish baselines for all the internal EWMAs of L3. After each benchmark scenario has run for 10 minutes, the coordinator retrieves the request latency and HTTP status code of each request and calculates the resulting success rate. Additionally, the coordinator retrieves metrics for percentile latencies, RPS, and internal components of L3, like EWMA values with a one-second granularity. This allows us to reason about the internal state of L3 at any point in time and explain behavior observed for certain scenarios.

5.2 Validation and Robustness Testing

In this section, we perform benchmarks to validate design decisions of L3. First, we determine a penalty factor in §5.2.1, which is used for all subsequent benchmarks. Then, we compare the performance of the EWMA with the PeakEWMA algorithm in §5.2.2 and validate that EWMA is the right choice for L3's use case.

5.2.1 Penalty Factor. To highlight the effect of the constant penalty factor P introduced in §3.1 on success rate and latency decrease, we run a series of benchmarks with the `failure-2` scenario. `Failure-2`'s latency is shown in Figure 1b and success rate is depicted in Figure 7a. The success rate of this scenario is mostly around 99% for all three clusters, with a few rare spikes down to 90%. We vary the value of P for each benchmark run, starting from $P = 100ms$ up to $P = 1000ms$. Additionally, we perform another run with $P = 1500ms$ to verify the trend for larger P. We calculate the



(a) Simulated success rate of `failure-2`

(b) Penalty factor results of `failure-2`

Figure 7: Understanding the impact of penalty factor on scenario `failure-2`. a) the success rate of the scenario. b) the penalty factor P impacts the success rate and percentile latency decrease. We chose $P = 0.6s$ as a good compromise between success rate and percentile latency decrease

relative percentage decrease of the percentile latency of L3 in comparison to the round-robin algorithm. We repeat each experiment twice to increase our sample size.

Figure 7b shows the impact of different P on the success rate and the latency percentile reduction. With increasing P, the latency percentile reduction diminishes. The reason for such behavior lies in Equation 3 for the latency estimate L_{est}. By increasing P, the TrafficSplit's backend weights w_b change due to the failure rate. Thus, the ratios among the values of w_b change more extremely. In such situations, individual backends are brought closer to their saturation point, which is accompanied by higher latency.

L_{est} is designed on the assumption that clients perform retries on failed requests. Accordingly, we see the RPS increase when the failure rate increases. In our benchmark, however, we did not perform retries for simplicity. We argue that the effect of P on the latency percentile decrease might not be as strong with retries as in our benchmark.

Furthermore, we analyzed the impact of P on the success rate. A larger P increases the latency estimate L_{est}, which lowers w_b and the proportion of traffic sent to backends with a lower success rate. Thus, the success rate increases with larger P, although the increase of success rate diminishes with larger P and eventually converges towards a ceiling. The ceiling is largely determined by the backend with the highest success rate, as its success rate is the best that can be theoretically achieved. The average success rate of the backend with the highest success rate at any point in time is 99.8% in scenario `failure-2` used for this benchmark. Thus a ceiling around 99.0% seems reasonable for our algorithm in this scenario. Measuring the round-robin success rate resulted in an average of 98.59%, similar to what was achieved with $P = 100ms$.

An additional improvement could involve L3 dynamically determining P for each workload based on historical latency data from failed requests. However, in its current state, we adopt a static approach, setting $P = 600ms$ for all benchmarks. This choice aims to strike a balance between latency and success rate across varying workloads.

5.2.2 Comparing EWMA with PeakEWMA. For L3, we considered using PeakEWMA, an EWMA algorithm that is especially sensitive to sample peaks. We compare L3 with either EWMA or PeakEWMA with the round-robin algorithm. For this benchmark, we use `scenario-4` with its 99th percentile latency shown in Figure 6b since the tail latency of executing request in this scenario has the highest fluctuation among the five traces we obtained from the Tier Mobility network. Each benchmark was performed three times.

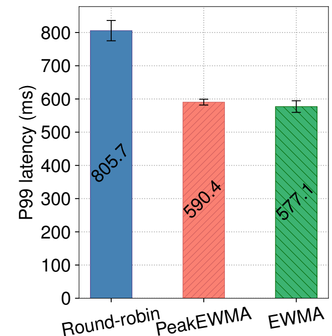


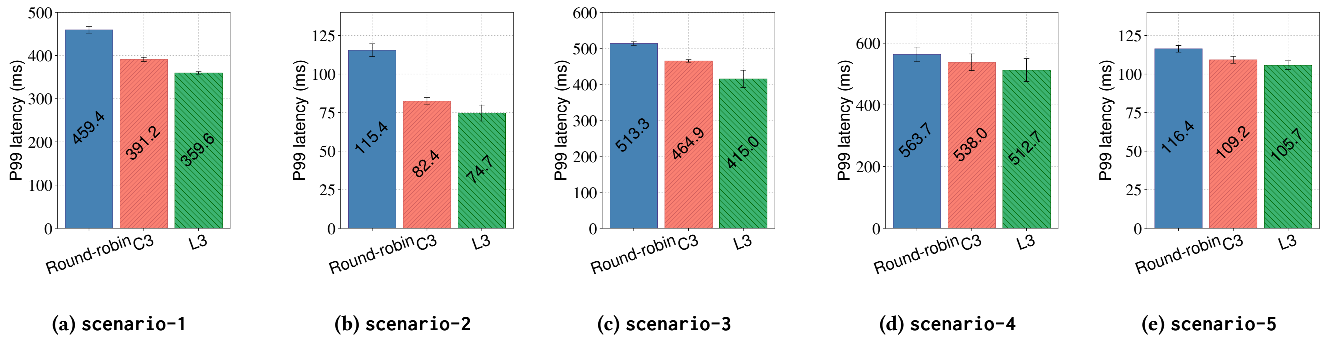
Figure 8: The P99 latency of round-robin, and L3 with PeakEWMA or EWMA.

Figure 8 depicts that L3 with EWMA or PeakEWMA both outperform the round-robin algorithm. More specifically, we can observe from the 99th percentile latency results that PeakEWMA and EWMA reduce the latency by 26.7% and 28.4% compared with the round-robin approach. As EWMA slightly decreases the latency by 2.3% compared to PeakEWMA, we use it for all subsequent benchmarks.

5.3 Latency and Success Rate

For general performance evaluation, we run a series of benchmarks to analyze both the latency in §5.3.1 and the effect of introducing failure rate on latency and success rate in §5.3.2.

5.3.1 Latency. In this section, we first report the 99th percentile execution latency of the hotel-reservation application of DeathStarBench benchmark [37] for the round-robin, C3 [49], and L3, and then show the performance of different systems when applying them on TIER Mobility's traces.



(a) `scenario-1` (b) `scenario-2` (c) `scenario-3` (d) `scenario-4` (e) `scenario-5`

Figure 10: The 99th percentile latency with L3 decreases in comparison to round-robin and C3 by 22% and 8% for `scenario-1`, 35% and 9% for `scenario-2`, 19% and 11% for `scenario-3`, 9% and 5% for `scenario-4`, and by 9% and 3% for `scenario-5`, respectively.

DeathStarBench. We conducted experiments using the hotel reservation application from the DeathStarBench suite, generating requests with a 100% success rate over a 20-minute duration. We ran the benchmark with different RPS with little to no changes in the results. At around 1000 RPS we approached the saturation points of some of the microservices at our test environment's scale, which led to an increase in latency. To prevent individual workloads from saturating due to a lack of autoscaling mechanisms, we chose to run the experiments at 200 RPS. Each benchmark configuration was repeated three times in alternating order. Figure 9 shows that L3 achieves a 26% and 22% reduction in 99th percentile latency as experienced by the user compared with round-robin and C3 strategies. The results confirm the effectiveness of including latency, success rate, and RPS in selecting the replicas in a multi-cluster environment.

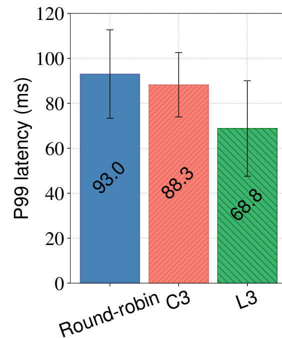


Figure 9: The execution latency of the algorithms when running the *hotel reservation* application of DeathStarBench benchmark in [37].

Tier Mobility. We run L3 on five different scenarios, i.e., `scenario-1` to `scenario-5`, where their trace information is shown in Figure 1 and Figure 6. We report the performance of L3 for each scenario in Figure 10 and compare them with the round-robin and C3 algorithms. We repeat each scenario three times to increase the sample size for each load balancing algorithm.

Out of the five scenarios, `scenario-1` shown in Figure 10a and `scenario-2` shown in Figure 10b have widely fluctuating 99th percentile latencies, with the median latency of one backend more often worse than the 99th percentile latency of the other backends. This creates quite favorable conditions for L3, as large performance differences between backends make it relatively easy to favor one backend over the others. In these scenarios, L3 improves the 99th percentile latency by 21.7% and 8% over round-robin and C3 for `scenario-1`, and 35% and 9% for `scenario-2`, respectively.

The three scenarios out of five, i.e., `scenario-3`, `scenario-4`, and `scenario-5`, shown in Figure 10c, Figure 10d, and Figure 10e have a stable median latency with some irregular peaks in the 99th percentile latency. For example, the backends' median latency in `scenario-1` has an average standard deviation of $\sigma = 30.5ms$, while the backends in `scenario-5` have a standard deviation of only $\sigma = 6.3ms$. Due to the relatively constant median latency, the 99th percentile latency plays a decisive role in the C3 and L3 algorithms. Since the 99th percentile latency fluctuates much more, it is beneficial to use PeakEWMA over EWMA to catch up with sudden peaks in latency quickly enough.

5.3.2 Success Rate. We evaluate the performance of L3 on success rate and latency by benchmarking two scenarios with different failure rate characteristics, comparing the results to round-robin and C3. As the microservices — from which the production data for `scenario-1` to `scenario-5` were taken — have very high availability and therefore no significant failure rate except during outages, we converted two of these scenarios into `failure-1` and `failure-2` by injecting artificial failure. `failure-1` has an average success rate of 91.4% with intermittent drops of success rate for single clusters down to 30%, representing heavy impact on availability. The average success rate of `failure-2` is 98.5%, with a success rate below 100% for most of the time and short drops by a maximum of 5%.

Figure 11 shows that L3 improves the 99th percentile latency over round-robin by 18.5% for `failure-1` and by 35% for `failure-2` scenario. Figure 12a depicts that L3 improves on round-robin's success rate from 91.4% to 92.4%. Figure 12b shows the results for `failure-2`, in which the success rate is almost unchanged slightly above 98%. It should be noted that C3's replica ranking algorithm does not perform success rate optimizations. The success rate is, therefore, consistently worse compared to the other algorithms, as it doesn't compromise latency for success rate.

It is important to note that the results are not directly comparable with the results from §5.3.1. Although the benchmarks of the same scenarios have similar ratios of different percentiles, the absolute values are not comparable. The benchmarks were conducted on different days, and the test environment was destroyed after each

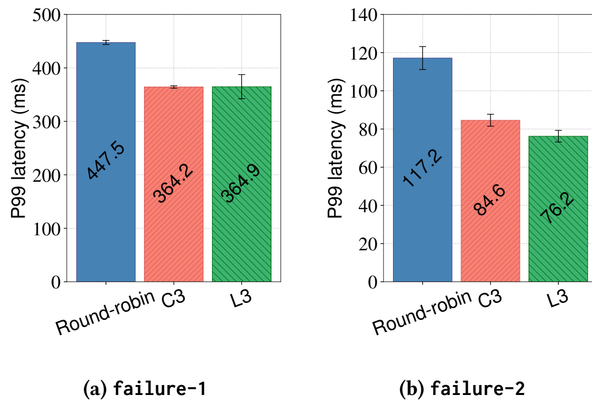


Figure 11: The 99th latency percentiles in failure-1 and failure-2 scenarios.

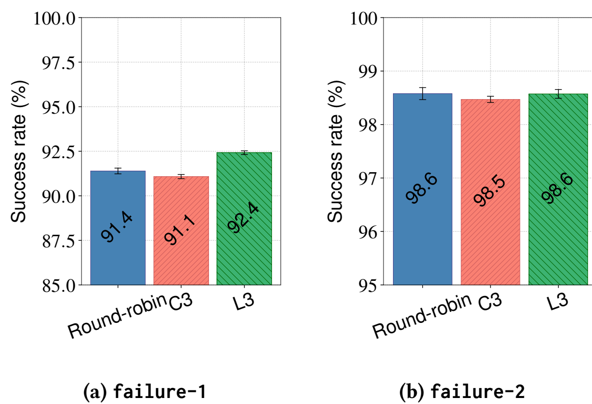


Figure 12: Improvement of success rate in failure-1 and failure-2 scenarios.

series of benchmarks to save costs. Therefore, the absolute values are not comparable due to different AWS EC2 instances and potentially a very different distribution of containers across data centers of availability zones. Nevertheless, we can observe the trend that L3's latency percentiles of the benchmarks in this subsection perform slightly less well than the benchmarks in the previous §5.3.1. The reason is that the weighting algorithm 1 makes trade-offs, as it optimizes for both low latency and high success rate.

6 RELATED WORK

Service meshes allow application developers to focus on the implementation of business logic instead of infrastructure concerns [32, 53]. We overview the attempts to improve the different aspects of offering microservices in geo-distributed scenarios. Within service-to-service load balancing, we distinguish between three load-balancing strategies: optimizing for availability, latency, and network transfer cost.

Optimizing for availability. Most service mesh implementations support multi-cluster failover functionality, which automatically routes traffic to an available replica of the service in another cluster in case of reduced availability of a service replica. This functionality is usually implemented based on health checks. Once a service replica is marked as unhealthy, the service mesh switches the traffic to an available replica. This is the case with locality load balancing for Istio [11], Linkerds failover extension [9], Traffic Director (a managed service mesh control plane by Google Cloud Platform, GCP) [19], and AWS' service mesh offering AppMesh [15]. However, the AWS AppMesh solution relies on a service mesh external DNS service, which means that in the worst-case scenario, the failover will take as long as it takes for the time to live (TTL) of cached DNS records to expire.

With L3, traffic can be quickly forwarded to other clusters without waiting a potentially long time for the fallback mechanism to kick in. If a failure due to overload in a part of the system is imminent, it may manifest in symptoms such as increased latency. In response, L3 proactively begins load balancing traffic to another cluster at an early stage.

Optimizing for latency. Service mesh implementations typically have mechanisms for application layer protocols to optimize latency within a cluster. Linkerd [10] maintains a moving average of the round-trip time of other services' replicas in the network proxy of each container. The number of outstanding requests weighs the moving average, and Linkerd distributes the traffic to the replicas for which this cost function is the smallest [2]. To the best of our knowledge, no service mesh implementation supports latency-based load balancing across multiple clusters, as L3 does. ServiceRouter [46] minimizes the remote-procedure call (RPC) latency for Meta's service mesh while considering the routing of the requests among different regions. Expected Latency Selector (ELS) is a load balancing algorithm from Spotify [1] with similarities to C3 proposed in [49]. The algorithm uses latency, success rate, and queue depth metrics of pending requests to balance traffic to Virtual Machines (VMs). However, ELS does not support service meshes, and the implementation is not publicly available. NetMarks [51] improves on the default Kubernetes scheduling by leveraging data plane metrics offered by Istio to reduce latency.

Optimizing for network transfer cost: Transferring data among different regions of public cloud providers can increase the cost of service execution in a geo-distributed cluster due to load balancing decisions [43]. The three most prominent public cloud vendors, AWS, Azure, and GCP, charge for all data transfer unless the transfer stays within the same data center [4, 7, 12]. They charge even for data transfer between a region's different availability zones, which is crucial for high-availability distributed system deployments. Linkerd, Istio, and GCP's Traffic Director support locality-aware load balancing, which strives to balance traffic to targets within the same locality zone to avoid additional costs.

At present, L3 lacks awareness of the network transfer costs and does not factor them into the load-balancing decisions. With the increasing maturity of service meshes and growing awareness of network costs, further optimizations could be possible in the future.

7 CONCLUSION

We proposed L3, a mechanism that builds on top of service meshes to minimize latency by directing traffic to the service replicas with the lowest latency. L3 accounts for the traffic distribution by reacting quickly to changes in latency, success rate, and RPS metrics of the replicas, thus providing service responsiveness and availability through latency- and fault tolerance. We performed a set of benchmarks to evaluate the performance of L3 in a multi-cluster service mesh environment, with a wide variety of latency behavior from different real-world microservices. We found that L3 significantly improves latency and can be used for arbitrary black-box microservices without fine-tuning algorithms or environments with high variability in their usage patterns. In future work, L3 could be extended with additional parameters to make it aware of data transmission costs from cloud vendors or energy availability. We further plan to determine the penalty factor P individually and dynamically for each workload. The continuous feedback about the response time of unsuccessful requests could be used. This would allow us to factor in the cost of failed requests more realistically.

ACKNOWLEDGMENT

This work was partially funded by BMBF (Federal Ministry of Education and Research) project, 6G-RIC: 6G Research and Innovation Cluster, grant 16KISK020K, 2021-2025

REFERENCES

[1] 2015. ELS: a latency-based load balancer. https://engineering.atspotify.com/2015/12/els-part-1/ https://engineering.atspotify.com/2015/12/els-part-2/. Accessed: 2022-11-23.

[2] 2016. Beyond Round Robin: Load Balancing for Latency. https://linkerd.io/2016/03/16/beyond-round-robin-load-balancing-for-latency/. Accessed: 2022-12-05.

[3] 2021. CNCF Operator White Paper. https://github.com/cncf/tag-app-delivery/blob/eece8f7307f2970f46f100f51932db106db46968/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md. Accessed: 2022-11-25.

[4] 2022. All networking pricing. https://cloud.google.com/vpc/network-pricing. Accessed: 2022-12-05.

[5] 2022. Amazon EC2. https://aws.amazon.com/ec2/. Accessed: 2022-11-28.

[6] 2022. Amazon Elastic Kubernetes Service (EKS). https://aws.amazon.com/eks/. Accessed: 2022-11-28.

[7] 2022. Bandwidth pricing. https://azure.microsoft.com/en-us/pricing/details/bandwidth/. Accessed: 2022-12-05.

[8] 2022. Controllers. https://v1-24.docs.kubernetes.io/docs/concepts/architecture/controller/. Accessed: 2022-11-25.

[9] 2022. linkerd-failover. https://github.com/linkerd/linkerd-failover. Accessed: 2022-12-03.

[10] 2022. Linkerd: The world's lightest, fastest service mesh. https://linkerd.io/. Accessed: 2022-12-16.

[11] 2022. Locality failover. https://istio.io/v1.15/docs/tasks/traffic-management/locality-load-balancing/failover/. Accessed: 2022-12-03.

[12] 2022. Overview of Data Transfer Costs for Common Architectures. https://aws.amazon.com/blogs/architecture/overview-of-data-transfer-costs-for-common-architectures/. Accessed: 2022-12-05.

[13] 2022. PeakEWMA from Finagle. https://github.com/twitter/finagle/blob/9cc08d15216497bb03a1cafda96b7266cfbbcff1/finagle-core/src/main/scala/com/twitter/finagle/loadbalancer/PeakEwma.scala. Accessed: 2022-11-26.

[14] 2022. Proxy Metrics. https://linkerd.io/2.12/reference/proxy-metrics/. Accessed: 2022-11-25.

[15] 2022. Run an active-active multi-region Kubernetes application with AppMesh and EKS. https://aws.amazon.com/blogs/containers/run-an-active-active-multi-region-kubernetes-application-with-appmesh-and-eks/. Accessed: 2022-12-03.

[16] 2022. Service meshes are on the rise — but greater understanding and experience are required. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf. Accessed: 2022-12-02.

[17] 2022. Simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io/. Accessed: 2022-12-16.

[18] 2022. TIER Mobility. https://tier.app. Accessed: 2022-11-30.

[19] 2022. Traffic Director load balancing. https://cloud.google.com/traffic-director/docs/load-balancing. Accessed: 2022-12-03.

[20] 2022. Traffic Split. https://github.com/servicemeshinterface/smi-spec/blob/main/apis/traffic-split/v1alpha4/traffic-split.md. Accessed: 2022-12-11.

[21] 2023. AWS for the Edge. https://aws.amazon.com/edge/services/. Accessed: 2022-12-01.

[22] 2023. From metrics to insight: Power your metrics and alerting with the leading open-source monitoring solution. https://prometheus.io/. Accessed: 2023-01-15.

[23] 2023. Kubernetes. https://kubernetes.io/. Accessed: 2023-5-24.

[24] 2023. Kuma - The universal Envoy service mesh for distributed service connectivity. https://kuma.io/. Accessed: 2023-11-30.

[25] 2023. Linkerd Architecture. https://linkerd.io/2.12/reference/architecture/. Accessed: 2023-01-15.

[26] 2023. Multi-cluster Traffic Management. https://istio.io/latest/docs/ops/configuration/traffic-management/multicluster/. Accessed: 2023-11-30.

[27] 2023. wrk2 - A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2. Accessed: 2023-11-30.

[28] 2024. AWS Global Accelerator features. https://aws.amazon.com/global-accelerator/features/. Accessed: 2024-03-09.

[29] 2024. Azure Traffic Manager - Performance traffic-routing method. https://learn.microsoft.com/en-us/azure/traffic-manager/traffic-manager-routing-methods#performance-traffic-routing-method. Accessed: 2024-03-09.

[30] 2024. Benchmarking Linkerd and Istio: 2021 Redux. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/. Accessed: 2024-03-16.

[31] 2024. Cloud Load Balancing. https://cloud.google.com/load-balancing. Accessed: 2024-03-09.

[32] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks* (Virtual Event, United Kingdom) *(HotNets '21)*. 229–236.

[33] Yong Cui, Ningwei Dai, Zeqi Lai, Minming Li, Zhenhua Li, Yuming Hu, Kui Ren, and Yuchi Chen. 2019. TailCutter: Wisely Cutting Tail Latency in Cloud CDNs Under Cost Constraints. *IEEE/ACM Transactions on Networking* 27, 4 (2019), 1612–1628.

[34] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[35] Yoav Einav. 2020. Amazon Found Every 100ms of Latency Cost them 1% in Sales. https://tinyurl.com/mr2hjw5x. 2020-01-20.

[36] Vajiheh Farhadi, Fidan Mehmeti, Ting He, Thomas F. La Porta, Hana Khamfroush, Shiqiang Wang, Kevin S. Chan, and Konstantinos Poularakis. 2021. Service Placement and Request Scheduling for Data-Intensive Applications in Edge Clouds. *IEEE/ACM Transactions on Networking* 29, 2 (2021), 779–792.

[37] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 3–18.

[38] Yuchen Jin, Sundararajan Renganathan, Ganesh Ananthanarayanan, Junchen Jiang, Venkata N. Padmanabhan, Manuel Schroder, Matt Calder, and Arvind Krishnamurthy. 2019. Zooming in on Wide-Area Latencies to a Global Cloud Provider. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. 104–116.

[39] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 122–1225.

[40] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers.. In *USENIX annual technical conference*. 363–378.

[41] Habib Mostafaei, Georgios Smaragdakis, Thomas Zinner, and Anja Feldmann. 2022. Delay-Resistant Geo-Distributed Analytics. *IEEE Transactions on Network and Service Management* 19, 4 (2022), 4734 – 4749.

[42] Yipei Niu, Fangming Liu, and Zongpeng Li. 2018. Load Balancing Across Microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 198–206.

[43] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-Distributed Data Analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. 421–434.

[44] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *Proceedings of The 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[45] Waleed Reda, Kirill Bogdanov, Alexandros Milolidakis, Hamid Ghasemirahni, Marco Chiesa, Gerald Q. Maguire, and Dejan Kostić. 2020. Path Persistence in the Cloud: A Study of the Effects of Inter-Region Traffic Engineering in a Large Cloud Provider's Network. *SIGCOMM Comput. Commun. Rev.* 50, 2 (may 2020), 11–23.

[46] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 969–985.

[47] Bhavana Vannarth Shobhana, Srinivas Narayana, and Badri Nath. 2022. Load Balancers Need In-Band Feedback Control. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (Austin, Texas) *(HotNets '22)*. 76–84.

[48] Akshitha Sriraman and Thomas F Wenisch. 2018. μtune: Auto-tuned threading for {OLDI} microservices. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 177–194.

[49] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 513–527.

[50] Lin Wang, Lei Jiao, Ting He, Jun Li, and Henri Bal. 2021. Service Placement for Collaborative Edge Applications. *IEEE/ACM Transactions on Networking* 29, 1 (2021), 34–47.

[51] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. 2021. NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. 1–9.

[52] C-Q Yang and Barton P Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*. IEEE Computer Society, 366–367.

[53] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2023. Dissecting Overheads of Service Mesh Sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*. 142–157.