Critical Limitations of the Least Outstanding Request Load Balancing policy in Service Meshes for Large-Scale Microservice Applications

Andrea Detti, Ludovico Funari

***Abstract*—Service meshes are becoming pivotal software frameworks for managing communication among microservices in distributed applications. Each microservice in a service mesh is paired with an L7 sidecar proxy, which intercepts incoming and outgoing requests to provide enhanced observability, traffic management, and security. These sidecar proxies use application-level load balancing policies to route outgoing requests to available replicas of destination microservices. A widely adopted policy is the Least Outstanding Request (LOR), which directs requests to the replica with the fewest outstanding requests.**

While LOR effectively reduces latency in applications with a small number of replicas, our comprehensive investigation—combining analytical, simulation, and experimental methods—uncovers a novel and critical issue for large-scale microservice applications: the performance of LOR significantly degrades as the number of microservice replicas increases, eventually converging to the performance of a random load balancing policy.

To recover LOR performance at scale, we propose an open-source solution named *Proxy-Service*, tailored for microservice applications where load balancing incurs significantly lower resource demands than microservice execution. The core idea is to consolidate load balancing decisions per microservice into one or a few reverse proxies, transparently injected into the application.

***Index Terms*—Cloud Computing, Microservices Applications, Service Meshes, Load Balancing**

I. Introduction

Microservice architecture decomposes the server-side logic of a web application into independent services, each responsible for a specific function and interacting through standardized network APIs, typically HTTP or gRPC [1]–[3]. A running copy of a microservice is referred to as an *instance*; multiple instances can be deployed concurrently to handle varying workload demands and ensure fault tolerance through replication.

Microservice applications are typically deployed on Kubernetes clusters [4], where each microservice instance runs within a Pod, which is a group of one or more Linux containers that share the same network namespace. Each Pod main container runs the microservice logic, while optional *sidecar* containers provide logging, monitoring, security, etc.

Kubernetes orchestrates the lifecycle of these Pods by rescheduling failed ones and autoscaling instances based on system metrics. By default, it uses a Layer 4 (L4) random load balancing policy to distribute service requests across available microservice instances.

Authors are with CNIT and the Department of Electronic Engineering, University of Rome "Tor Vergata", Italy, e-mail: {name.surname@uniroma2.it}

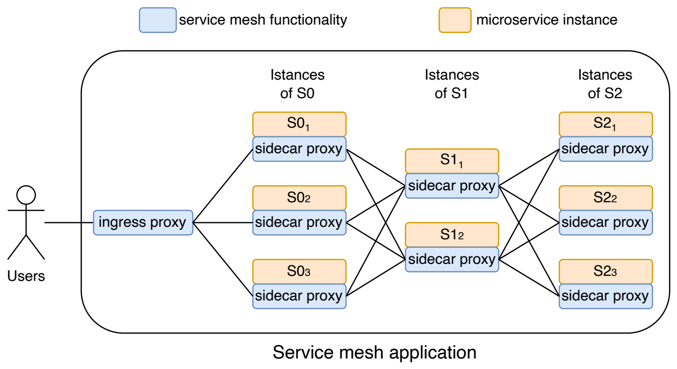


Fig. 1: A three-tier microservice application enhanced with service mesh functionality. The request path is S0 $\rightarrow$ S1 $\rightarrow$ S2. Each service has a varying number of instances.

Service Mesh: Service meshes, such as Consul [5], Istio [6], and Linkerd [7], extend Kubernetes by offering observability, secure communication, and advanced traffic control without altering application code [8], [9]. A Layer 7 (L7) proxy is automatically injected as a sidecar into each Pod, intercepting all ingress and egress gRPC/HTTP requests to enforce traffic policies and perform application-layer load balancing. This overrides the default Kubernetes L4 random policy.

Fig. 1 illustrates a three-tier microservice application enhanced with service mesh functionality. Requests enter the system through an ingress proxy and traverse a chain of microservices (S0 $\rightarrow$ S1 $\rightarrow$ S2). Each microservice is replicated; for example, S0 and S2 have three replicas, while S1 has two. The sidecar proxies associated with S0 instances perform load balancing for requests directed to S1, and those of S1 handle load balancing for requests forwarded to S2.

Sidecar proxies apply advanced Layer 7 load balancing policies, such as *Least Outstanding Requests (d)*, denoted as LOR(d). In this policy, each proxy monitors the number of its own outstanding HTTP/gRPC requests to each server, where each server corresponds to a downstream microservice instance, and forwards new requests to the least loaded one among d randomly selected candidates.[1]

[1]We define *downstream* as the direction in which a request flows and *upstream* as the reverse direction. If microservice S0 sends a request to S1, then S1 is a downstream microservice of S0, and conversely, S0 is an upstream microservice of S1.

The policy has a simple implementation and offers significant latency improvements over random or round-robin policies [10]. For this reason, it is widely adopted as the default in service meshes[2].

Load Balancer Literature: A related and well-studied policy is the Join-the-Shortest-Queue, abbreviated as JSQ(d) [11], where the load balancing function selects the server with the smallest queue among d randomly chosen candidates. Mitzenmacher's seminal analysis [10] modeled JSQ(d) in a system with a single load balancer distributing a Poisson stream of jobs to N homogeneous servers, each with FIFO queues and exponential service times. Known as the *supermarket model*, this system revealed the "power of two choices": the majority of latency gains with respect to random policy occur already at $d = 2$, with diminishing returns beyond.

In systems with a single load balancer, LOR(d) and JSQ(d) are functionally equivalent, since the number of outstanding requests effectively reflects the server queue length. However, service meshes load balancing operation are made by multiple and independent load balancers, i.e., the sidecar proxies of the microservice instances, each making routing decisions with only local information. This results in what we term a *non-collaborative distributed supermarket model*, where each load balancer sees only its own outstanding/queued requests to a server, not the total queue length. Consequently, LOR(d) in such contexts becomes a JSQ(d) with imperfect/local knowledge of the server queue.

Motivations: Our experiments on real microservice applications [12] revealed that the limited visibility of queue states in LOR(d) leads to a significant degradation in performance as the level of microservice replication increases. This observation motivated the development of a continuous-time analytical model that captures the key characteristics of service mesh environments with multiple independent load balancers. Our analysis shows that, as the number of load balancers grows, the performance of LOR(d) asymptotically converges to that of a random policy, thereby exposing a scalability limitation in its application within service meshes.

Addressing this limitation requires rethinking the design of load balancing mechanisms. While prior research has proposed collaborative, feedback-based strategies for time-slotted systems [25], [26], these methods inevitably introduce synchronization complexity and operational overhead. In contrast, we propose a practical and non-collaborative solution specifically tailored for service mesh applications, where load balancing is not the performance bottleneck.

Our approach, called *Proxy-Service*, augments the microservice application with one or a few reverse proxies per microservice. These proxies intercept all incoming requests for the instances of a microservice and perform the LOR(d) selection centrally, thereby consolidating the load balancing decision into a single or few control points. This architectural change restores the effectiveness of LOR(d) by mitigating the performance degradation caused by distributed, non-collaborative load balancing operations made by sidecar proxies.

We implemented Proxy-Service as an open-source Kubernetes custom resource [19]. Experimental evaluations confirm that it significantly improves LOR(d) performance at scale, with negligible CPU and memory overhead, and without requiring any modifications to application code or sidecar proxy implementations.

Contributions: This paper makes the following contributions:

- **New Analytical Model Reflecting Service Mesh Realities**. We introduce a novel analytical model—the *non-collaborative distributed supermarket model*—that accurately captures the behavior and limitations of LOR(d) in service mesh environments with distributed load balancers.
- **Discovery of Asymptotic Latency Degradation in LOR(d)**. We theoretically and empirically demonstrate that, as the number of microservice replicas increases, LOR(d) loses its latency advantage and asymptotically approaches the performance of random policy.
- **Proxy-Service Mitigation Strategy for Microservice Applications**. We propose a novel architectural solution that mitigates this degradation by consolidating load balancing decisions per microservice through the introduction of reverse proxies, thereby restoring the effectiveness of LOR(d) in service mesh environments.
- **Kubernetes/Istio Implementation**. We implement this strategy as a Kubernetes-native custom resource integrated with Istio, demonstrating the feasibility of Proxy-Service in real-world deployments and confirming its performance benefits through experimental evaluation.

Paper Outline: Sec. II presents the analytical formulation of the non-collaborative distributed supermarket model and validates its predictions through simulation and real-world experiments. Sec. III details the Proxy-Service concept and Kubernetes implementation. Sec. IV evaluates the performance of Proxy-Service under realistic workloads. Sec. V compares Proxy-Service method with collaborative approaches. Sec. VI surveys related literature, and Sec. VII concludes with key insights and directions for future research.

II. Analysis of the Non-Collaborative Distributed Supermarket Model

A. Definition

Consider a system in which external requests are fairly handled by M load balancers that route traffic to N servers. Servers use a first-in, first-out (FIFO) queue strategy and the service time for a request is exponentially distributed with mean $T_s = 1$. Each load balancer receives a Poisson stream of requests with a rate of $N\lambda$ req/s and uses the LOR(d) policy to distribute requests to servers. When a request is received, the load balancer chooses d servers at random with replacement and, among them, routes the request to the server that currently holds the minimum number of outstanding requests originating from the load balancer, that is, the load balancer can only

[2]In multi-cluster service-mesh environments, LOR(d) is typically applied within each individual cluster. Cluster-level routing is handled separately, often using locality-aware policies that prefer the local cluster if the target microservice is available; otherwise, remote clusters are selected based on configurable inter-cluster traffic criteria.

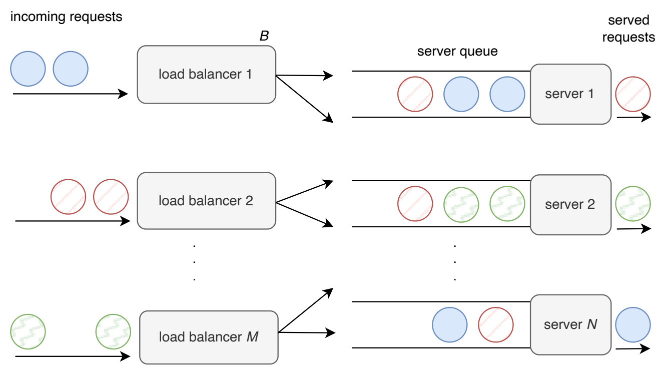


Fig. 2: Non-Collaborative Distributed Supermarket Model

observe its outstanding requests. In case of a tie among servers, the choice among them is random. We named this system "non-collaborative distributed supermarket model".

Fig. 2 illustrates the non-collaborative distributed supermarket model, emphasizing the imperfect knowledge that each load balancer has about the actual queue occupancy of the servers. In the figure, requests handled by different load balancers are represented with different colors.

From the perspective of load balancer 1, server 1 has 2 outstanding requests, server 2 has zero, and server N has 1. These outstanding request counts are used by the load balancer to make dispatching decisions. However, as previously noted, these values differ from the actual number of requests in the server queues because each load balancer only tracks the requests it has issued. As a result, load balancing decisions are made based on partial and local information, which may lead to suboptimal request distribution and performance degradation at scale.

B. Analytical model

In this subsection, we present an asymptotic analysis of the non-collaborative distributed supermarket model whose results get closer and closer to reality as the number of servers N and load balancers M increases. The analysis extends that in [10].

Consider a generic load balancer B at time t, e.g., the first load balancer in Fig. 2. We define m_i, as the number of servers with at least i queued requests coming from B. We define $s_i = m_i/N$ as the fraction of servers with at least i queued requests of B.

For large values of N, s_i can be regarded as the probability that the number of requests of the load balancer B contained in a server queue is greater than or equal to i.

In a time interval dt, the probability that a request arrives at the load balancer B is equal to $N\lambda dt$. The probability that the request is routed to a queue that contains $i-1$ requests from B is equal to $s_{i-1}^d - s_i^d$, i.e., the probability that the d servers chosen at random all have at least $i-1$ requests of B but not all have more than i requests of B. Consequently, the increase in m_i due to the arrival of requests in dt seconds is equal to $N\lambda(s_{i-1}^d - s_i^d)dt$.

In a time interval dt, the probability that a request of the load balancer B leaves a queue that contains i requests of B is equal to $N\mu_i(s_i - s_{i+1})dt$, where $s_i - s_{i+1}$ is the probability that a queue contains i requests from B and, under this condition, μ_i is the average departure rate of requests of the load balancer B. Note that this rate only takes into account the requests of B that leave the queue, rather than any request, so it may be less than the inverse of the average service time, i.e., $\mu_i \leq 1$.

Combining the increase in m_i due to arrivals and the decrease due to departures, we can write the derivative of m_i as follows.

$$\frac{dm_i}{dt} = N\lambda(s_{i-1}^d - s_i^d) - N\mu_i(s_i - s_{i+1}) \tag{1}$$

Dividing by N, we obtain the following set of differential equations.

$$\frac{ds_i}{dt} = \lambda(s_{i-1}^d - s_i^d) - \mu_i(s_i - s_{i+1}) \tag{2}$$

Each server receives an overall rate of requests equal to $M\lambda$ and the whole system is stable if $M\lambda < 1$ [3]. In this case, the derivative ds_i/dt in Eq. 2 must be zero in a steady state. Consequently, we can compute the probabilities s_i, for $i > 1$, solving the following set of recursive equations [4]:

$$s_{i+1} = s_i - \frac{\lambda}{\mu_i}(s_{i-1}^d - s_i^d)\ , \text{ for } i > 1 \tag{3}$$

To start the recursion, we need s_0 and s_1. Obviously,

$$s_0 = 1 \tag{4}$$

Regarding s_1, we note that since the system is stable, the rate of requests from the load balancer B entering and leaving a server queue must be equal, in formulas,

$$\lambda = \sum_{i=1}^{\infty}(s_i - s_{i+1})\mu_i \tag{5}$$

where λ is the input rate of request from a load balancer B, $s_i - s_{i+1}$ is the probability that the server queue contains i requests of B and μ_i is the departure rate of B requests in this condition.

To compute μ_i, we use a *mean-field* approximation for which the effect of other load balancers on any given server queue is approximated by a single averaged effect [10] [13]. Consequently, we assume that when a request of B enters a queue, it finds $(M-1)Eq$ requests from other load balancers, where Eq is the average number of requests of a load balancer in a queue. Therefore, since we consider an average service time $T_s = 1$, μ_i can be approximated as follows.

$$\mu_i \approx \frac{i}{i + (M-1)\,Eq} \tag{6}$$

[3]Our model shows that the performance of the non-collaborative distributed supermarket model is always better than that of a system in which the load balancers use a random policy. Such random systems are stable when $M\lambda < 1$, so we conjecture that this condition also holds in our model. A similar argument was used in [10] to justify the same stability condition for single load balancer JSQ(d) systems.

[4]In [10], the Author solved this recursion in closed form for $\mu_i = 1$. However, in our non-collaborative distributed supermarket model μ_i is not equal to one and varies with i. Therefore, his elegant solution is unfortunately not applicable.

Regarding Eq, it can be readily written as [10],

$$Eq = \sum_{i=1}^{\infty} s_i \tag{7}$$

To compute the expected time T a request spends in the system, that is, the average request latency, we can reason as follows. A request from the load balancer B enters a queue with $i-1$ queued requests of B with a probability equal to $s_{i-1}^d - s_i^d$. In this scenario, the request has ahead of it $i-1$ requests of B and $(M-1)Eq$ requests of the other load balancers, with unit service time. Subsequently, the average request latency can be written as:

$$\begin{aligned} T &= \sum_{i=1}^{\infty} \left(i + (M-1)\, Eq\right)\left(s_{i-1}^d - s_i^d\right) \\ &= (M-1)\, Eq + \sum_{i=0}^{\infty} s_i^d \end{aligned} \tag{8}$$

The Eq. 3, Eq. 4, and Eq. 5, combined with Eq. 6 and Eq. 7 provide all the relations needed to compute s_i and in turn request latency T. We were unable to solve them in closed form, however numerical methods can be used [5].

We note that for an average service time T_s different from 1, the latency of the request in Eq. 8 must be simply multiplied by T_s. The motivation is that we can carry out the analysis on a different time scale equal to $1/T_s$. Consequently, the average service time turns out to be equal to 1, therefore, the proposed formulas are valid. Finally, it is necessary to rescale the time backward by multiplying the resulting latency by T_s.

C. Asymptotic behavior

Theorem 1. *As the number of load balancers increases, the average request latency T asymptotically tends to that of a system where load balancers choose servers at random, i.e.,*

$$T \to \frac{1}{1-\rho} \quad \text{as} \quad M \to \infty \tag{9}$$

where $\rho = M\lambda$ is the utilization factor of a server.

Proof. Since the stream of requests generated by users is distributed evenly across load balancers, as the number M of load balancers increases, each load balancer will handle a smaller and smaller portion of the request stream. Consequently, the probability of finding more than one request from a specific load balancer in a server queue tends to zero, i.e.,

$$s_i \to 0 \quad \text{as} \quad M \to \infty, \quad \text{for } i > 1 \tag{10}$$

Using Eq. 10 in Eq. 5, Eq. 6 and Eq. 7, and considering that $M-1$ tends to M as M increases, we can compute the asymptotic value of s_1 as follows.

$$s_1 \to \frac{\lambda}{1-\lambda M} \quad \text{as} \quad M \to \infty \tag{11}$$

[5]For instance, we used Matlab numerical methods while assuming that for large values of i (e.g., $i = 10000$) $s_i \approx 0$.

Using Eq. 11 in Eq. 8 and considering that, for $d > 1$, s_1^d tends to zero much faster than s_1 as M increases, we have

$$T \to 1 + Ms_1 \quad \text{as} \quad M \to \infty \tag{12}$$

Using Eq. 11 in Eq. 12, we get Eq. 9 that is the well-known latency of a system with unit service time where load balancers choose servers at random [14]. □

The asymptotic tendency of a system using the LOR(d) policy to behave like a system that uses the random policy results from the following observation. As the number M of parallel load balancers increases, the probability s_1 of finding at least one request from a load balancer B in a queue becomes smaller and smaller. Consequently, it is increasingly likely that the d servers chosen at random by the LOR(d) algorithm do not contain any outstanding requests from B. Consequently, there is a tie and B chooses a server at random.

D. Validation of the Model Through Simulations

To validate the theoretical model, we developed a simulator of the non-collaborative distributed supermarket system. The setup uses a Poisson request stream with average service time T_s = 20ms and target utilization $\rho = 0.75$, such that $\lambda = \rho/(MT_s)$. The number of servers is fixed at $N = 40$.

Fig. 3a presents the latency results for the LOR(2) policy as the number of load balancers varies, with 95% confidence intervals computed from ten simulation runs. Model and simulation results align closely and converge as M increases, consistent with mean-field approximations[6]. The latency of LOR(2) approaches that of the random policy, $T_s/(1-\rho)$ = 80ms, confirming the model's predictive accuracy and Theorem 1.

Fig. 3b explores the impact of increasing utilization ρ for fixed values of $M = 1$ and $M = 20$. In line with prior findings on JSQ(d) [15], latency reduction with LOR(d) is most pronounced under high load. Note the considerably greater effectiveness of LOR systems with a single load balancer ($M = 1$) compared to the cases of multiple load balancers ($M = 20$).

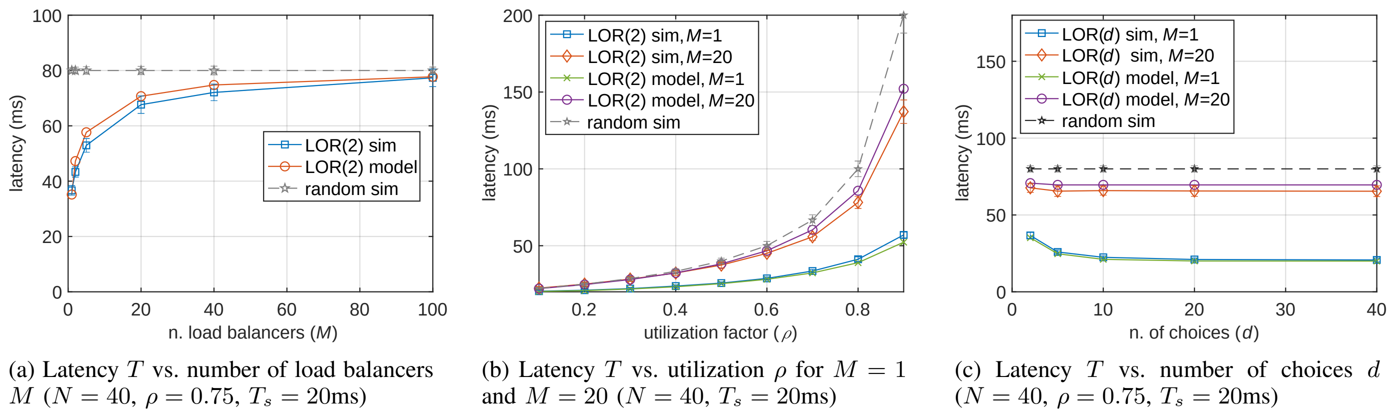
Fig. 3c examines the effect of varying d on LOR(d). Results confirm the "power of two choices": most of the benefit is gained at $d = 2$, with diminishing returns beyond. As M increases, latency flattens and approaches that of random routing, indicating reduced sensitivity to d.

We also tested under bursty traffic using Lognormal inter-arrival times with varying coefficient of variation (CV). As shown in Fig. 4, latency increases with burstiness, but the key trend holds: LOR(2) performance deteriorates with more load balancers, converging to the behavior of the random policy.

E. Validation of the Model Findings in Real-World Contexts

To assess the practical relevance of our theoretical insights, we evaluated the model under real-world conditions, where assumptions such as Poisson arrivals and idealized service times may not strictly hold. While exact numerical agreement

[6]In all simulations, the mean-field approximation in Eq. 6 slightly overestimates Eq and latency, but this overestimation diminishes as M increases.



(a) Latency T vs. number of load balancers M ($N = 40$, $\rho = 0.75$, $T_s = 20$ms)

(b) Latency T vs. utilization ρ for $M = 1$ and $M = 20$ ($N = 40$, $T_s = 20$ms)

(c) Latency T vs. number of choices d ($N = 40$, $\rho = 0.75$, $T_s = 20$ms)

Fig. 3: Simulation results validating the model and illustrating: (a) LOR(2) degradation with more load balancers, (b) stronger degradation under high traffic, and (c) the effectiveness of two random choices.

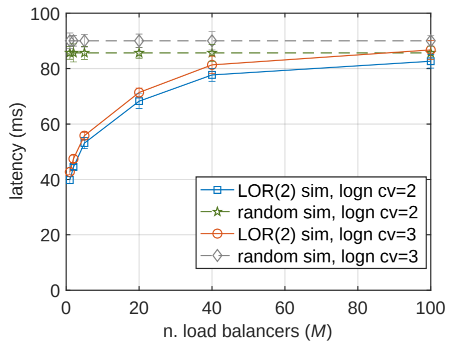


Fig. 4: Latency T vs. number of load balancers M under Lognormal inter-arrival times (CV=2 and CV=3). LOR(2) degradation persists under non-Poisson traffic.

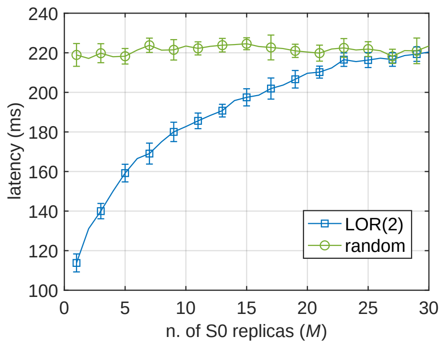


Fig. 5: Average request latency of the application in Fig. 1 excluding `S2` vs. `S0` replicas. LOR(2) performance degrades with replication, consistent with theoretical predictions.

is not expected, the goal is to confirm whether the degradation trend of LOR(d) persists in practice.

We deployed a two-tier microservice application using the µBench tool [17] on a Kubernetes cluster with Istio. The application comprises microservices `S0` and `S1` (as in Fig. 1, omitting `S2`). We varied the number of `S0` replicas (M) and fixed the number of `S1` replicas at $N = 20$.

User requests (HTTP GET) were routed via an Istio ingress gateway to randomly selected `S0` instances. Each `S0` instance immediately forwarded the request to an `S1` replica via its sidecar proxy by using LOR(2) policy. The `S1` service performed CPU-bound processing and returned a response, which was relayed back to the user.

This setup replicates the non-collaborative distributed supermarket model: `S0` proxies act as independent load balancers, and `S1` replicas as servers.

Using JMeter [18], we generated a constant load of 100 req/s and compared LOR(2) with a random load balancing policy. Fig. 5 reports average latency and 95% confidence intervals. Results confirm that LOR(2) effectiveness declines as M increases, with latency approaching that of random policy.

III. Proxy-Service: Consolidating Load Balancing Decisions to Restore LOR(d) at Scale

The theoretical analysis in the previous section showed that the performance of LOR(d) degrades due to insufficient visibility into queue states, resulting from the concurrent presence of two key issues: (1) load balancing decisions are fragmented across many parallel load balancers, and (2) these load balancers operate independently, without feedback from servers regarding their queue states.

Several literature approaches address the second issue by introducing collaborative mechanisms that balance synchronization overhead with queue visibility [25]–[27].

In contrast, this paper proposes a novel approach specifically tailored to microservice applications, under the key assumption that load balancing operations are substantially less resource-intensive than microservice execution. The effectiveness of the proposed solution relies on this assumption, which enables the consolidation of load balancing decisions into a small number of control points, called *Proxy-Services*, that are transparently integrated into the application.

As shown in Fig. 6, each Proxy-Service functions as a reverse proxy, forwarding requests to microservice instances using the LOR(2) load balancing policy. Given that load balancing is less resource-intensive than microservice execution, a single Proxy-Service instance (e.g., `PS-S1`) can efficiently

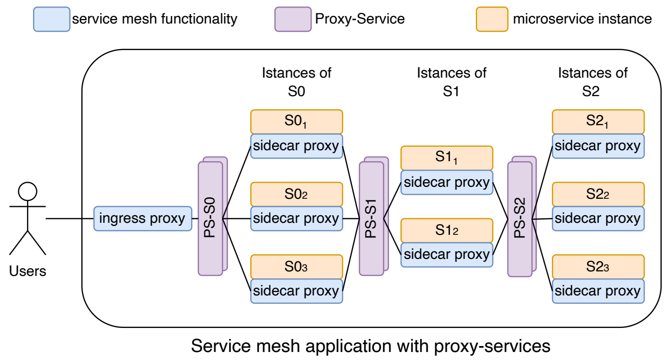


Fig. 6: A three-tier microservice application with service mesh functionality and Proxy-Services.

serve traffic from many upstream microservice instances (e.g., `S0`), thereby restoring the performance of LOR(2) by eliminating fragmentation in the load balancing process.[7]

If the incoming request volume exceeds the capacity of a single Proxy-Service, horizontal scaling can be applied. While this increases the number of load balancers, the number of Proxy-Service instances M_{PS} remains significantly smaller than the number of upstream microservice replicas, substantially reducing fragmentation and maintaining effective load balancing.

A key innovation of Proxy-Service is that it maintains the simplicity of non-collaborative LOR(d). It neither alters the load balancing algorithm nor introduces state synchronization among proxies. Instead, it transparently modifies the microservice dependency graph by inserting a small number of centralized decision points. This enables effective aggregation of load balancing without requiring changes to application code, sidecar behavior, or the service mesh control plane.

Proxy-Service introduces two potential sources of additional latency: (i) a slight increase in end-to-end round-trip time (RTT) due to the extra network hop, and (ii) a processing delay associated with the load balancing operation. The RTT increase is negligible in modern data center networks, and the processing delay is effectively mitigated through autoscaling of the Proxy-Service as needed. As reported in Sec. IV, both overheads are minimal compared to the latency reduction achieved through load balancing consolidation in case of moderate or heavy traffic conditions.

Under low-traffic conditions, these small delays may slightly increase overall latency without being offset by load balancing gains. However, since baseline latencies are typically well below service-level objectives in such cases, the trade-off remains acceptable. Moreover, Proxy-Service can be selectively enabled on a per-microservice basis, allowing fine-grained control of latency-performance trade-offs.

[7]Performance measurements on our Kubernetes testbed show that a single Proxy-Service instance can sustain approximately 14,000 requests per second while consuming only 0.5 CPU cores, confirming that Proxy-Service introduces minimal processing overhead and can support high-throughput workloads.

A. *Proxy-Service Implementation*

We implemented Proxy-Service for Kubernetes clusters with Istio service mesh support [4], [6]. The implementation defines a custom Kubernetes resource named *Proxy-Service* (PS) [19], composed of a Kubernetes Deployment, Service, Horizontal Pod Autoscaler (HPA), and Istio Gateway, Virtual Service, and Destination Rule. These are managed by a dedicated Kubernetes Proxy-Service Operator.

Consider a microservice `S1` with multiple replicas managed by a Kubernetes Service named `SN1`. When Proxy-Service is enabled for `S1`, the Operator performs the following:

- Deploys a new `PS-S1` Deployment with a Pod running an Istio Gateway proxy performing as reverse proxy for `S1` instances.
- Creates an HPA for `PS-S1` to autoscale based on CPU usage (e.g., targeting 70%).
- Renames the original service `SN1` to `SN1-target`, pointing directly to `S1` instances.
- Defines a new `SN1` service pointing to the `PS-S1` Pods, so that requests to `S1` are transparently routed through the Proxy-Service.
- Installs an Istio Virtual Service and Destination Rule ensuring that `PS-S1` forwards requests to `SN1-target` using the LOR(2) policy (configured as `LEAST_REQUEST` in Istio).

These resources are fully integrated with the Istio and Kubernetes control planes, allowing Proxy-Service to seamlessly adapt to microservice dynamics.

IV. Experimental Results

This section presents the results of an experimental study conducted on a Kubernetes cluster with 6 worker nodes, each equipped with 8 CPUs. The cluster implements the Istio service mesh, where load balancing operations are performed by the sidecar proxies injected into the Pods. The Proxy-Service has been implemented as an Istio-Ingress, i.e., a Pod that contain only the sidecar (Envoy) proxy connected with the Istio control plane.

The objectives of the campaign are twofold:

- Assess experimentally whether the latency performance degradation of the LOR(d) policy, suggested by the analytical model for a single microservice-to-microservice interaction, is also valid for the overall latency of a complete microservice application as replication increases.
- Evaluate the performance of Proxy-Service while also considering other state-of-the-art load balancing strategies.

The reported plots present average values along with their corresponding 95% confidence intervals.

Benchmark Applications: The first experimental campaign uses benchmark applications generated by µBench [17]. We consider two example applications. The first, shown in Fig. 7a represents an application with a “hub-and-spoke” dependency graph. Users send requests to S0 and the sequence of HTTP interactions to serve a user request is as follows: `S0→S1, S0→S2, S0→S3`. The second application, shown in Fig. 7b, represents an application with a “chain“ dependency graph.

TABLE I: Throughput (req/s) used for the tests of μBench applications versus number of replicas (R)

R	1	2	5	10	15	20
hub-and-spoke	10	18.5	41	75	100	120
chain	9.3	17.7	38	72.5	99	122

The sequence of HTTP interactions to serve a user request is as follows: `S0→S1, S1→S2, S2→S3`.

Microservices run a CPU-intensive task whose complexity has been configured so that their CPU utilization is almost equal. Microservices have the same number of instances/replicas equal to R. The CPU allocation for each microservice is the same and equal to 300 milliseconds per second, as know as "millis". Specifically, 300 millis is the value of Kubernetes CPU `Request` and `Limit` of each microservice Pod.

User requests for S0 are generated by JMeter [18]. In each test, we vary R and use a request throughput so that the request latency is about 240ms when the service mesh uses the random load balancing policy. The resulting throughput is reported in Tab. I and increases with R, since the application has more resources and therefore can handle more requests with the same latency.

The same values of throughput are used to measure the latency in the case of LOR(2) (Istio default) and LOR(2) with Proxy-Service. For all tests, we use a single instance of each Proxy-Service because the resulting CPU utilization is very low and replication is not necessary.

Fig. 7c and Fig. 7d show the latency for the hub-and-spoke and chain μBench applications, respectively. At the increases in the number of replicas, the LOR(2) policy results in a latency that tends to the random one. Therefore, the conclusion we draw for the non-collaborative distributed supermarket model still holds for a whole microservice application [8].

The introduction of Proxy-Service remarkably reduces the latency and makes it independent of the application scale. This independence from R is motivated as follows. With Proxy-Services, each microservice-to-microservice interaction turns out to be mediated by a single load balancer system ($M_{PS} = 1$). Consequently, each iteration can be modeled with the supermarket system analyzed in [10] whose performance depends only on the load ρ, in the case of Poissonian interarrivals. In the different tests, we noticed that R replicas/servers have approximately the same CPU load (ρ). Therefore, as R increases, the performance remains approximately constant, as predicted by the supermarket model, despite the fact that in our case we do not have Poissonian interarrivals.

Sock-Shop: A final performance assessment was conducted using the Sock-Shop e-commerce application [12]. The configuration required to reproduce the test is available in the GitHub repository [19]. This application comprises a heterogeneous set of microservices implemented in different programming languages and utilizing various databases. All microservices communicate via REST over HTTP.

[8]The comparison with the simpler two-service chain application in Fig. 5 shows that as the number of microservices traversed per request increases, the performance gap between LOR(2) and random load balancing slightly widens. This is because, in more complex service architectures, the extended request path amplifies the impact of uninformed load-balancing decisions.

The dependency graph is shown in Fig. 8a, which also indicates the number of replicas per microservice. For example, the front-end microservice has 30 replicas, while other microservices have a single instance unless otherwise specified. For the tests with Proxy-Services, we deployed a single Proxy-Service Pod per microservice, each requesting 100 milliCPU.

The benchmark load was generated using JMeter, simulating 200 concurrent users accessing the application. During each interaction, a user generates a structured sequence of HTTP requests, as outlined in Tab. II. The request rate is controlled to ensure a specific throughput in requests per second (req/s).

Step	**HTTP Req.**	**Step**	**HTTP Req.**	**Step**	**HTTP Req.**
1	Home	8	Catalogue	15	Orders
2	Login	9	Detail	16	Cart-del
3	Category	10	Basket	17	Cart-post
4	Catalogue	11	Cart-post	18	Cart-del
5	Detail	12	Orders	19	Orders
6	Catalogue	13	Cart-del	20	Cart-del
7	Detail	14	Cart-post	21	Cart-post
				22	Orders

TABLE II: HTTP request sequence during user interaction with the Sock-Shop microservice application.

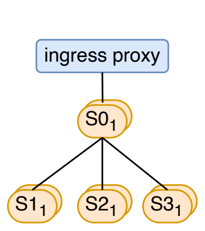
Fig. 8b presents the request latency as a function of throughput, varied between 100 and 600 req/s in increments of 50 req/s. The evaluation considers different Istio load balancing policies, namely random, LOR(2), LOR(2) with Proxy-Service (PS), round-robin, and ring-hash [9].

As expected, latency increases with traffic load across all balancing strategies. However, the experimental results reveal significant performance differences under heavy traffic. The application became unstable beyond 400 req/s when using the ring-hash policy, as it failed to sustain higher throughput with the available processing resources. Both random, round-robin and LOR(2) supported up to 450 req/s, with LOR(2) achieving lower latency due to its ability to make a more informed load balancing decisions. LOR(2) with Proxy-Service provided the lowest latency and the highest supported throughput of 600 req/s, demonstrating superior efficiency in utilizing available processing resources [10].

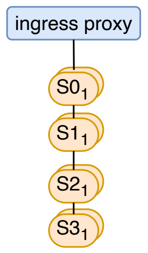
To explain the superior performance of Proxy-Service, Fig. 8c and Fig. 8d show the CPU utilization ratio of three selected Pods from the `orders` microservice under LOR(2) without and with Proxy-Service, respectively. The CPU utilization ratio is defined as the fraction of CPU usage relative

[9]The ring-hash policy implements a consistent hashing mechanism, where microservice Pods are mapped onto a virtual ring of 1024 points based on their IP addresses. Requests are assigned by hashing the source IP and selecting the nearest clockwise Pod.

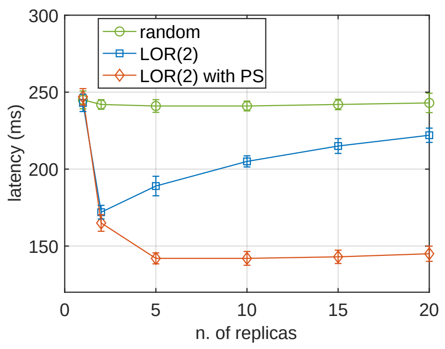
[10]We also repeated the test with two Proxy-Service instances per microservice ($M_{PS} = 2$). Compared to the case with a single Proxy-Service, the additional instance introduced a small delay increase, whose maximum value is 8 ms at 550 req/s, due to the loss of perfect knowledge of microservice occupancy.



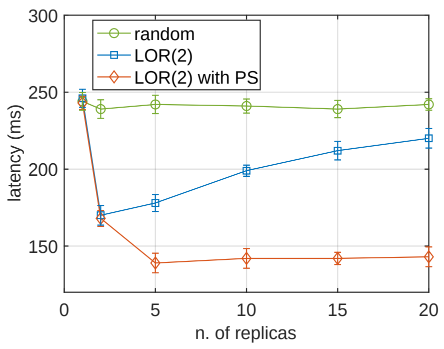
(a) hub-and-spoke



(b) chain



(c) Average request latency of μBench hub-and-spoke application vs n. of replicas



(d) Average request latency of μBench chain application vs n. of replicas

Fig. 7: Dependency graphs and experimental results of μBench applications, showing Proxy-Service maintains latency performance with increasing replicas per microservice.

to the CPU requested by the Pod[11].

Without Proxy-Service, LOR(2) relies solely on local information, leading to *load imbalances* that increase latency. For instance, at 500 req/s, one of the `orders` Pods (Pod #1) becomes over-utilized, resulting in application instability. This issue arises because upstream microservices lack an accurate view of `orders` Pod occupancy and inadvertently overload a Pod #1. Similar behavior is observed with random and round-robin policies at 500 req/s, and with ring-hash at 450 req/s.

In contrast, Proxy-Service improves load balancing by ensuring *full awareness of Pod occupancy*. The results in Fig. 8d demonstrate that CPU usage is more evenly distributed across `orders` Pods when Proxy-Service is enabled. By preventing load imbalances, Proxy-Service reduces latency and enhances the system's ability to handle higher traffic loads efficiently. This balanced distribution mitigates the risk of overloading individual Pods, thereby increasing system stability even under high request rates.

Fig. 8e reports the CPU consumption per request by Proxy-Service across different microservices. The observed computational overhead is minimal, averaging only 0.033 CPU milliseconds per request. This indicates that a single Proxy-Service instance (an Envoy Proxy), configured to reserve with 100 CPU millis, can efficiently handle thousands of requests per second with negligible resource consumption. Additional measurement confirmed that Proxy-Service introduces a minimal overhead to the application resource consumption, accounting for less than 5% of total CPU and memory consumption in any considered load conditions[12].

The introduction of Proxy-Service improves load balancing efficiency but also introduces an additional proxy in the request path. To quantify the associated latency penalty, a controlled experiment was conducted. The number of `catalogue` microservice replicas was reduced from five to one, and a workload was generated in which users exclusively accessed the catalogue page. This setup ensured that each request traversed a single Proxy-Service instance without benefiting from load balancing, allowing the direct measurement of additional traversal delay. The results, shown in Fig. 8f, compare the latency of LOR(2) with and without Proxy-Service under varying request rates. The analysis reveals that Proxy-Service introduces an average latency penalty of only 1.08 ms.

This minor additional delay is negligible in low-load conditions and is entirely offset under high traffic scenarios, where Proxy-Service significantly enhances load balancing efficiency. As horizontal scaling increases the number of replicas, the advantages of Proxy-Service outweigh the small traversal delay, further confirming its effectiveness in large-scale deployments.

V. Comparison with Collaborative Solutions

Proxy-Service represents a "first attempt" to mitigate the performance degradation of the LOR(2) load balancing strategy in microservice applications by consolidating load balancing operations, while maintaining the non-collaborative design between servers and load balancers.

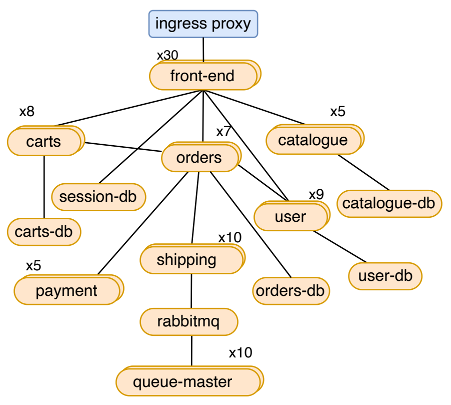
Several prior works, though not specifically targeting service mesh environments, propose collaborative load balancing solutions in which servers and load balancers share precise or approximate information about queue occupancy [25]–[27]. Adapting these theoretical approaches to real-world service mesh environments would require sidecar proxies to implement collaborative strategies in order to preserve the application transparency property of service mesh frameworks.

Inspired by the Local Shortest Queue (LSQ) family of collaborative load balancing strategies proposed for *time-slotted* systems [25], [26], we implemented an *Asynchronous LSQ(2)* strategy, referred to as A-LSQ(2), in our simulator and adapted it to the continuous-time dynamics of service mesh environments. The policy operates as follows:

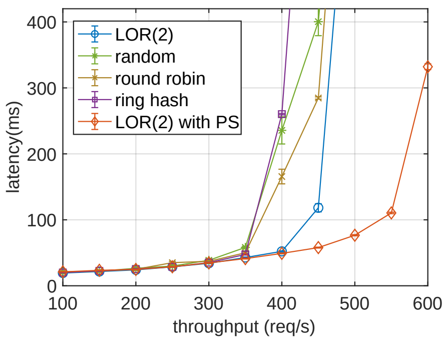
- When a request arrives, a load balancer applies the LOR(2) policy using its local queue state.
- When a server finishes processing a request, for each $1 \leq b \leq M$, it sends its updated queue status to load balancer b with probability $p = \min(\omega/M, 1)$. The parameter ω

[11]CPU Request is the amount of CPU reserved by Kubernetes for a Pod. The CPU Limit defines the maximum CPU a Pod can use. For Sock-Shop, Requests and Limits are identical. The exact configuration is available in the example folder of the GitHub repository [19].

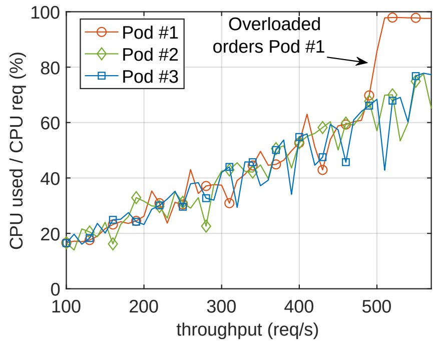
[12]We observe that to further reduce the Proxy-Service footprint, a single instance can be shared among a group of microservices with low request rates.



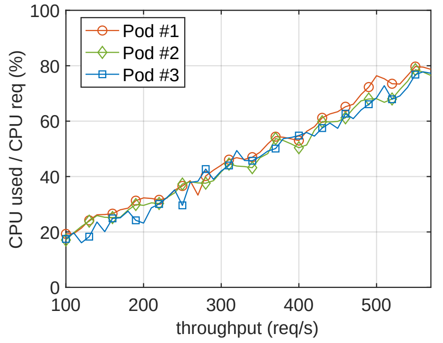
(a) Sock-Shop microservices application



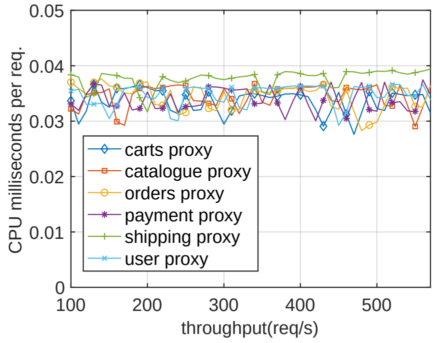
(b) Average request latency of Sockshop microservices application vs request throughput



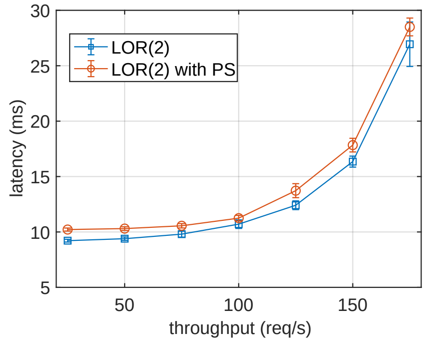
(c) Ratio of Requested and used CPU for three Pods of Orders microservice vs request throughput for LOR(2)



(d) Ratio of Requested and used CPU for three Pods of Orders microservice vs request throughput for LOR(2) with Proxy-Service



(e) CPU usage per request of Proxy-Services vs request throughput



(f) Latency of Catalogue requests vs request throughput for a single replica of the Catalogue microservice

Fig. 8: Dependency graph and experimental results for the Sock-Shop application, showing: (b) Proxy-Service achieves higher throughput than other load-balancing strategies; (c,d) it balances CPU usage across microservice replicas; (e) it incurs minimal resource consumption per request; and (f) it introduces limited latency overhead when used without replication.

denotes the maximum update overhead, expressed as the average number of update messages per service request.

- Each load balancer updates its local queue state as follows: (1) it decrements the queue size by one when a response is received, mimicking the implicit update behavior of LOR at the reception of a gRPC/HTTP response; (2) it applies the new value provided by an update message if one is received.

Fig. 9 shows the average request latency versus the number of load balancers for different strategies, including LOR(2), LOR(2) with Proxy-Service, and A-LSQ(2) with varying update overheads ω. For Proxy-Service, we model the performance as that of LOR(2) with a single load balancer, adding a constant latency penalty of 2 ms, based on our worst-case experimental observations from Fig. 8f. We neglect additional processing overhead for A-LSQ(2) updates, due to the lack of concrete experimental data.

The results demonstrate that collaborative policies such as A-LSQ(2) can reduce request latency compared to non-collaborative LOR(2). The improvement increases with greater update overhead ω. However, we observed that the update effort required to achieve meaningful improvements is not neg-

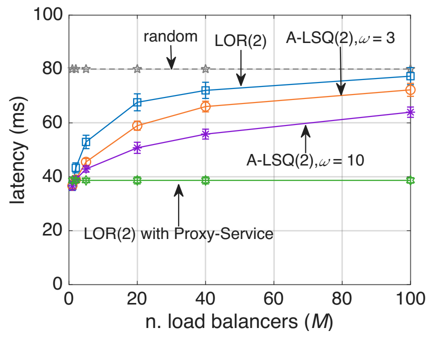


Fig. 9: Average request latency versus the number of load balancers M, for $N = 40$, $\rho = 0.75$, and $T_s = 20$ms, showing latency reduction from both non-collaborative Proxy-Service and collaborative A-LSQ(2) policy.

ligible. For example, with $M = 100$, an overhead of $\omega = 10$ update messages per served request yields a latency reduction of approximately 15 ms, compared to the 40 ms improvement achieved in case of a single Proxy-Service instance and no-collaboration.

It is also important to note that A-LSQ(2), like LOR(2), tends toward random performance at large scale. This occurs because, as M increases, the probability p of a load balancer receiving recent updates diminishes, causing the system to asymptotically approach LOR(2) behavior, and eventually, random load balancing. We further observed (though not reported here) that A-LSQ(2) can achieve a lower asymptotic latency than random if the per-load balancer update probability p is kept constant (i.e., $\omega = M \cdot p$). However, this approach would result in a very poorly scalable system, as the required update overhead would grow with the system size.

These observations suggest that while collaborative strategies such as A-LSQ(2) can improve performance, they may introduce substantial complexity and operational overhead. In contrast, Proxy-Service offers a lightweight and practical alternative that achieves effective load balancing at scale without requiring architectural changes. However, in systems where load balancing consumes significant resources, Proxy-Service may become a performance bottleneck and collaborative approaches are recommended to reduce LOR(2) latency.

VI. Related Works

Load balancing: Load balancing plays a crucial role in optimizing the performance and resource utilization of distributed systems. A seminal work that has significantly influenced the field is by M. Mitzenmacher [10] for which each request is routed by a single load balancer whose strategy is as follows: the balancer randomly selects d servers and chooses among them the one that has the least number of queued requests. When the number of choices d is equal to the number of servers, this strategy is also known as join-the-shortest-queue (JSQ) [11], while for values of d less than the number of servers, it is known as JSQ(d). As the number of servers increases, performance modeling becomes quickly intractable, which motivates the use of an asymptotic mean-field approximation [10].

Compared to random request routing, JSQ provides valuable delay reduction and, surprisingly, asymptotic results show that, for a large number of servers, using only two random choices (d=2) instead of considering the entire set of servers produces exponential delay improvement, while increasing d yields only marginal improvements. This concept is usually referred to as *the power of two random choices*.

Several subsequent studies have built upon the foundational concepts presented in the aforementioned paper. Some papers explore variations, such as the case of heterogeneous servers [13] [20], or different scheduling, such as processor sharing [21]. Other works analyzed different properties of the JSQ. Other papers focus on a deeper performance evaluation. For example, in [22] [23], it is shown that JSQ minimizes the total time needed to finish processing all jobs that arrive by a fixed period of time and that JSQ minimizes the delay in the heavy-traffic regime, that is, when the arrival rate approaches the maximum capacity of the system. Notable follow-up works have delved into refining the theoretical underpinnings [24].

Recently, the literature has also considered use cases of JSQ policy with multiple independent load balancers, also called dispatchers. Such scenarios are of particular interest for high-load cloud applications for which a single load balancer can become a performance bottleneck. In [25], the authors introduced a time-slotted multi-dispatcher system model in which a number of jobs arrive from external clients to each dispatcher at the beginning of a time slot and the dispatcher immediately routes them all to a back-end server that is chosen according to a load balancing policy. The load balancing decisions of the different dispatchers are then made at the same time, that is, at the beginning of a time slot. At the end of a time slot, each server has drained a certain number of jobs from its queue, and for each job served, the corresponding client is notified, but not the dispatcher, which therefore is unaware of the job completion, i.e., it operates in a "fire-and-forget" basis. The number of arriving jobs to different dispatchers per slot and the number of departing jobs per slot from different servers are both i.i.d. integer random variables.

This time-slotted multi-dispatcher model has had several follow-ups in the literature as it poses two challenging problems:

- (problem 1) When dispatchers have accurate information about the status of server queues, the performance worsens because all dispatchers turn to have the same view of the system and therefore synchronously make the same routing decision, overloading the same server. This problem has driven research toward stochastic load balancing strategies that avoid it.
- (problem 2) The fire-and-forget approach leads load balancers to know how much they send to servers, but do not know how much goes out from servers. This has led research to introduce low-overhead feedback systems from servers to load balancers to update the load balancers' view of server queue length.

In the same paper, the Authors proposed a policy called Local Shortest Queue (LSQ), in which each dispatcher has a local estimate of the queue length of the servers and routes the bulks of arrived jobs to the server with the shortest estimated queue length. The estimate is updated immediately after the routing decision as follows: i) the queue of the selected server is increased by the number of routed jobs; a stochastic set of dispatcher-server pairs is chosen. For each pair, the server informs the dispatcher of its actual queue length, and the dispatcher sets the queue length estimate to this value. These infrequent control communications allow the queue estimate to not drift too far from reality. The Authors modeled this system and demonstrated its stability and the effectiveness of their LSQ policy through simulations.

In [26], the Authors extended the theoretical analysis to a wider family of *tilted* load balancing policies and also discussed how to design optimal load balancing schemes using delayed information. Moreover, it has been observed that for these systems, inaccurate information, i.e., queue length estimates, can lead to better performance simply because they are synchronous, so having complete real-time knowledge of the queue state leads all dispatchers to make the same routing decision at the beginning of each slot, thus overloading the same selected server [25] [26]. To solve this problem, in [27], the Authors propose a stochastic load balancing algorithm based on the intuition that the goal of load balancing resembles that of a water-filling algorithm, but with discrete distributions considering that the jobs are not divisible. Simulations show that their algorithm outperforms those previously proposed.

Unlike these time-slotted systems, our work focuses on service meshes that are continuous-time systems in which load balancing decisions are made asynchronously, that is, as requests arrive. This native asynchronicity avoids problem 1 and makes time-slotted models unsuitable, as non-existent problems would show up. In part, our system does not even present problem 2, because the dispatchers are L7 proxies and at least know that their jobs have been completed by receiving responses from the servers. Furthermore, our model, as much as current service mesh software, does not consider the presence of queue status feedback from servers to load balancers to reduce complexity, considering the high dynamicity of microservice applications in Kubernetes clusters.

In general, with respect to the time-slotted model considered in the literature so far, the non-collaborative distributed supermarket model is more closely aligned with the use cases of microservice applications using service meshes with LOR(d). In addition, to the best of our knowledge, our work is the first showing that the latency performance of a continuous-time multi load balancer system twith LOR(d) policy approaches asymptotically that of a system using a random policy.

Microservice optimization: In recent years, the application of load balancers has extended beyond traditional access to back-end replica servers to modern service mesh frameworks supporting microservice applications, although it has received only limited attention in the literature so far.

In [28], the Authors observe that microservice applications serve user requests by involving chains (sequences) of microservices. Many chains may exist for different types of requests, and chains may share microservices. As a result, they propose a chain-oriented load balancing algorithm (COLBA) that takes into account the possible chains in the application and aims to reduce response latency.

In [29], the Authors studied basic load balancing and QoS-aware among interdependent IoT microservices. Similarly to [28], the paper combines load balancing policy with application logic, which in this case is represented by the dependency graph of microservices. From the graph of dependencies among microservices, it is possible to create a graph of dependencies among instances, called infrastructure graph, in which nodes are instances of microservices, and a link between instances exists if the related microservices have a dependency. Each node has a capacity, that is, a maximum rate of requests it can handle, and introduces a constant processing delay. The load balancing problem is to identify the request rate to be allocated on each link of the infrastructure graph to not overload nodes (basic policy) or to guarantee a specific delay (QoS-aware policy).

In this paper, we focus on a more "traditional" scenario, for which the service mesh software and related load balancers are application-agnostic. While coupling load balancing policies with the application logic, i.e., knowledge of the possible chains in this case, can help optimization, it can also reduce the DevOps capability of the microservices architecture, since, for example, any upgrade or extension of the application not only impacts the microservices involved, but also requires a revision of the load balancing strategy of the entire application.

In [30], the Authors consider a different application scenario from ours, in which they assume that any microservice instance can perform any task. The user requires the execution of a sequence of tasks and the load balancing problem consists of choosing one instance for each task. It resembles a task scheduling problem where tasks must be executed by computing workers. The proposed algorithm aims to minimize a cost function composed of two weighted factors: the first is a measure of the imbalance between the average CPU utilization of different instances and the second factor is network traffic.

VII. Conclusions

This study identifies a fundamental limitation of the Least Outstanding Request (LOR) load balancing policy in service mesh environments: as the degree of microservice replication increases, LOR's effectiveness degrades and eventually converges to that of random policy. We supported this finding through a novel theoretical model and validated it with simulations and experiments on real-world microservice applications.

The root cause of this degradation lies in LOR's reliance on partial and local information: each sidecar proxy performs load balancing by observing only the number of outstanding requests it has issued, rather than the total number of requests queued at each destination microservice instance. As the number of load balancers increases — due to microservice replication — this knowledge gap widens, leading to increasingly suboptimal decisions that eventually approximate random selection.

To address this issue, we proposed and implemented *Proxy-Service*, a practical mechanism tailored for microservice applications where load balancing imposes significantly lower resource demands than microservice execution. Proxy-Service consolidates load balancing decisions into one or a few reverse proxies per microservice, which transparently forward requests to backend instances using the LOR(2) policy. This consolidation restores global server queue visibility by eliminating the fragmentation inherent in the standard distributed load balancing process of service mesh frameworks, without requiring changes to application code or introducing collaboration mechanisms. Importantly, service mesh control and data planes can remain unmodified, preserving compatibility with existing deployment architectures.

The centralization approach adopted by Proxy-Service may naturally prompt initial skepticism regarding its scalability. However, the broader industry trend confirms the viability of this direction: successful service mesh frameworks, such as Istio's ambient mode, are already adopting more centralized L7 proxy architectures [31]. Moreover, recent advancements like eBPF-based acceleration offer promising opportunities to further optimize Proxy-Service by reducing its processing footprint and improving data path efficiency [32].

However, Proxy-Service is not a one-size-fits-all solution. Its effectiveness relies on the explicit assumption that the computational overhead of load balancing is negligible compared to that of microservice execution; a condition that holds in many, but not all, microservice-based systems. In scenarios where load balancing operations consume a non-negligible portion of resources with respect to microservices, collaborative approaches that enhance queue visibility through feedback (e.g., A-LSQ) remain a valid and promising alternative. Our preliminary results indicate that such collaboration can partially recover LOR's lost performance. However, achieving meaningful improvements at scale typically requires frequent state updates, which may introduce significant synchronization overhead.

Acknowledgment

This work was partially supported by the EU under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership "Telecommunications of the Future" (PE00000001 - program "RESTART"). The authors would also like to thank the anonymous reviewers for their valuable feedback, which significantly motivated us to improve the quality and clarity of the manuscript.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[2] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[4] Kubernetes (k8s): Production-grade container orchestration. (Accessed 2025-03-17). [Online]. Available: https://kubernetes.io/

[5] Consul service mesh. (Accessed 2025-03-17). [Online]. Available: https://www.consul.io/

[6] Istio service mesh. (Accessed 2025-03-17). [Online]. Available: https://istio.io/

[7] Linkerd service mesh. (Accessed 2025-03-17). [Online]. Available: https://linkerd.io/

[8] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225.

[9] H. Saokar, S. Demetriou, N. Magerko, M. Kontorovich, J. Kirstein, M. Leibold, D. Skarlatos, H. Khandelwal, and C. Tang, "{ServiceRouter}: Hyperscale and minimal cost service mesh at meta," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 969–985.

[10] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[11] V. Gupta, M. H. Balter, K. Sigman, and W. Whitt, "Analysis of join-the-shortest-queue routing for web server farms," *Performance Evaluation*, vol. 64, no. 9-12, pp. 1062–1081, 2007.

[12] Sock shop : A microservice demo application. (Accessed 2025-03-17). [Online]. Available: https://github.com/microservices-demo/microservices-demo

[13] S. Allmeier and N. Gast, "Mean field and refined mean field approximations for heterogeneous systems: It works!" *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–43, 2022.

[14] L. Kleinrock, "Queuing systems, volume i: Theory," 1975.

[15] X. Zhou, F. Wu, J. Tan, Y. Sun, and N. Shroff, "Designing low-complexity heavy-traffic delay-optimal load balancing schemes: Theory to algorithms," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 2, pp. 1–30, 2017.

[16] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1998, pp. 151–160.

[17] A. Detti, L. Funari, and L. Petrucci, "µBench: An Open-Source Factory of Benchmark Microservice Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 968–980, 2023.

[18] Apache jmeter. (Accessed 2025-03-17). [Online]. Available: https://jmeter.apache.org/

[19] Proxy-service repository. (Accessed 2025-03-17). [Online]. Available: https://github.com/mSvcBench/Proxy-Service.git

[20] D. Hurtado-Lange and S. T. Maguluri, "Throughput and delay optimality of power-of-d choices in inhomogeneous load balancing systems," *Operations Research Letters*, vol. 49, no. 4, pp. 616–622, 2021.

[21] A. Mukhopadhyay and R. R. Mazumdar, "Analysis of randomized join-the-shortest-queue (jsq) schemes in large heterogeneous processor-sharing systems," *IEEE Transactions on Control of Network Systems*, vol. 3, no. 2, pp. 116–126, 2015.

[22] A. Ephremides, P. Varaiya, and J. Walrand, "A simple dynamic routing problem," *IEEE transactions on Automatic Control*, vol. 25, no. 4, pp. 690–693, 1980.

[23] G. Foschini and J. Salz, "A basic dynamic routing problem and diffusion," *IEEE Transactions on Communications*, vol. 26, no. 3, pp. 320–327, 1978.

[24] J. Abdul Jaleel, S. Doroudi, K. Gardner, and A. Wickeham, "A general "power-of-d" dispatching framework for heterogeneous systems," *Queueing Systems*, vol. 102, no. 3-4, pp. 431–480, 2022.

[25] S. Vargaftik, I. Keslassy, and A. Orda, "Lsq: Load balancing in large-scale heterogeneous systems with multiple dispatchers," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1186–1198, 2020.

[26] X. Zhou, N. Shroff, and A. Wierman, "Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 48, no. 3, pp. 57–58, 2021.

[27] G. Goren, S. Vargaftik, and Y. Moses, "Distributed dispatching in the parallel server model," *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1521–1534, 2023.

[28] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 198–206.

[29] R. Yu, V. T. Kilari, G. Xue, and D. Yang, "Load balancing for interdependent iot microservices," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 298–306.

[30] Y. Liang and Y. Lan, "Tclbm: A task chain-based load balancing algorithm for microservices," *Tsinghua Science and Technology*, vol. 26, no. 3, pp. 251–258, 2021.

[31] Istio ambient mode. (Accessed 2025-03-17). [Online]. Available: https://istio.io/latest/docs/ambient

[32] Merbridge - accelerate your mesh with ebpf. (Accessed 2025-03-17). [Online]. Available: https://istio.io/latest/blog/2022/merbridge/

Andrea Detti is a professor of Wireless Networks and Cloud Computing at the University of Rome "Tor Vergata". His current research activity spans on different topics in the area of computer networks and cloud computing. He is co-author of more than 80 papers on journals and conference proceedings, and participated to several EU funded projects with coordination and research roles.

Ludovico Funari is a researcher at the University of Rome Tor Vergata. His research activity includes IoT, Cloud and Edge computing. He has worked in European projects and RESTART National Program as a CNIT (Italian National Inter-University Consortium for Telecommunications) researcher at the University of Rome Tor Vergata.