Andrea Detti\textsuperscript{1} and Ludovico Funari\textsuperscript{1}

\textsuperscript{1}Affiliation not available

February 27, 2024
Cloud service meshes: analysis of the least outstanding request load balancing policy for large-scale microservice applications

Andrea Detti, Ludovico Funari

Abstract—Service meshes are emerging software frameworks to manage communications among microservices of distributed applications. With a service mesh, each microservice is flanked by an L7 sidecar proxy that intercepts any incoming and outgoing requests for better observability, traffic management, and security. The sidecar proxy uses an application-level load balancing policy to route outbound requests towards possible replicas of destination microservices. A widely used load balancing policy is the Least Outstanding Request (LOR), which routes requests to the microservice replica with the fewest outstanding requests. While the LOR policy significantly reduces request latency in scenarios with a single load balancer, our comprehensive investigation, spanning analytical, simulation, and experimental methodologies, reveals that its effectiveness decreases in environments with multiple load balancers, typical of service meshes serving applications with several microservice replicas. Specifically, the resulting request latency asymptotically tends to that provided by a random load balancing policy as the number of microservice replicas increases. To address this loss in efficacy, we propose a solution based on a new Kubernetes custom resource, named Proxy-Service, offering potential improvements in performance and scalability.

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

I. INTRODUCTION

Microservice architecture is a widely used approach to software development in which the server side of a Web application is broken down into small, independent, and loosely coupled services. Each service, known as a microservice, is responsible for a specific function and communicates with others through well-defined network APIs generally based on HTTP or gRPC [1], [2], [3]. A running copy of a microservice is called instance, and multiple instances of the same microservice can run simultaneously to handle different levels of workload and ensure fault tolerance.

A microservices application usually runs in a cluster of real or virtual machines, and related lifecycle management is handled by the Kubernetes orchestration platform [4]. Microservice instances run within Kubernetes computing units called Pods. A Pod is a collection of one or more Linux Containers that share the same network environment/namespace. The main Container of a Pod is the microservice instance, but other Containers can be included in the Pod, providing additional functionality, such as logging, monitoring or security, in a modular and isolated manner.

Kubernetes is undeniably an outstanding framework for reliably executing microservices applications. One of its key features is the ability to maintain the desired state of applications and automatically handle any deviations from that state by continuously monitoring and reconciling the actual state with the desired one. For example, if a Pod fails or a node becomes unhealthy, Kubernetes automatically reschedules or replaces the failed Pod with a new one to maintain the desired number of Pod replicas. If a Pod gets overloaded, Kubernetes provides autoscaling capabilities, allowing users to define policies that automatically adjust the number of Pod replicas based on resource utilization or custom metrics.

When there are multiple Pods/instances of the same microservice, the flow of requests for that microservice is automatically distributed by Kubernetes among the available replicas through a simple random load balancing policy.

Service Mesh: Increasingly, Kubernetes microservices applications are complemented by software frameworks called service meshes, such as Consul [5], Istio [6], Linkerd [7], etc. Service meshes aim to enhance the application with application-level observability, advanced service-to-service request routing, and load balancing [8], [9]. A major advantage of service meshes over other integrated solutions is that they provide such functionality without requiring any changes to the application code. A service mesh sits alongside the application rather than integrating within it.

Fig. 1 shows the data plane of a three-tier service mesh app-
Kubernetes control plane
incoming request
Service mesh control plane
outgoing requests
Kubernetes Pod
microservice instance S0
sidecar proxy
Service Endpoints
S1
S2
Sx
<instance 1>
<instance #w>
<instance ... policy
Load balancing policy
Kubernetes Pod

Fig. 2: A microservice Pod with service mesh functionality

application, specifically a microservice application with service mesh capabilities. In this setup, user requests traverse a chain of three microservices: S0 invokes S1, and subsequently, S1 interacts with S2. Each microservice has a different number of running instances. The links represent dependencies among instances, that is, a link exists if two instances can interact with each other to solve a user request.

During the application deployment, the service mesh control plane automatically injects an L7 sidecar proxy within each Kubernetes Pod running microservice instances. In addition, a service mesh ingress proxy exposes the necessary HTTP/gRPC endpoints of internal microservices to users. The resulting data plane is a network of proxies responsible for monitoring and routing service requests exchanged among microservice instances. Fig. 2 shows the architecture of a Kubernetes Pod augmented with a sidecar proxy. The sidecar proxy transparently intercepts every service request received and sent by the microservice instance, and this interception allows the observation of application-level metrics, such as latency and rate of requests, and, in addition, a more configurable control of microservice-to-microservice request routing, bypassing the default Kubernetes random load balancing policy.

To route an outgoing request, the proxy looks up a service endpoints table that contains the IP addresses of the instances of the destination microservice and uses a specific load balancing policy to select one of them. The service mesh control plane continuously interacts with the Kubernetes one to keep the endpoint table synchronized with the state of the instances in the cluster.

In addition to traditional round-robin and random strategies, a simple and effective load balancing policy widely used for service mesh applications is the so-called Least Outstanding Requests, or shortly Least Requests. For each microservice instance, the sidecar proxy keeps track of the number of HTTP/gRPC requests for which it has not yet received a response. To route a request, the proxy randomly chooses d instances of the destination microservice and sends the request to the one with which it has the least number of outstanding requests, with the ties broken randomly. We use the acronym LOR(d) to refer to this policy.

Load balancing literature: The literature on load balancers considers systems made up of one or more load balancers that distribute requests to a pool of servers. These systems can model microservices-to-microservice interactions such as those between S0 and S1, or S1 and S2 in Fig. 1. The load balancers are the sidecar proxies of microservice instances (like S0) that send requests to the instances of a destination microservice (like S1), which represent the servers. Outstanding requests are equivalent to queued requests.

In this context, LOR(d) can be regarded as a policy that uses local knowledge of the state of server queues. A LOR(d) load balancer knows the number of its requests that are in a server queue, but it has no knowledge of the number of requests from other load balancers that are in the queue. This limited or local knowledge restricts the load balancer’s ability to make fully informed decisions about which server to route requests to.

A literature policy closely resembling LOR(d) is called join-the-shortest-queue d, abbreviated as JSQ(d) [10]. The difference is that JSQ(d) assumes that a load balancer has perfect knowledge of the server queue length. This means that the load balancer can accurately determine the length of each server’s queue, not just based on its own requests but also considering all requests in the system.

M. Mitzenmacher conducted a seminal analysis of the JSQ(d) policy [11]. He considered a system consisting of a single JSQ(d) load balancer that distributes a Poisson request stream to a pool of N homogeneous servers. The servers operate on the first-in, first-out (FIFO) principle with exponential service time. He called this system the supermarket model and proposed an asymptotic analytical model for large values of N that use mean field theory. This model showed a rather surprising result that is known in the literature as “the power of two random choices”, that is, with respect to a simple random load balancing, the JSQ(d) policy with just d = 2 choices leads to an exponential decrease of the average request latency, while having d = x choices is only a constant factor better than d = x − 1, for x > 2. Practically, the latency reduction is considerable for d = 2, but limited for more than two choices.

With a single load balancer, LOR(d) and JSQ(d) are functionally equivalent. However, for multiple load balancer scenarios, such as service meshes, LOR(d) is like JSQ(d) with imperfect/local knowledge of server queue lengths.

Motivations: We are interested in evaluating the effectiveness of LOR(d) policy for service mesh applications. The lack of suitable analytical models and the limited availability of experimental campaigns led us to carry out the study reported in this paper.

Rather than the supermarket model, the interactions between microservices-to-microservices are better represented by a model we called non-collaborative distributed supermarket model. The term “distributed” denotes the presence of multiple load balancers, while the “non-collaborative” nature means that load balancers and servers operate without collaborating to attain knowledge of the total number of queued requests across servers. Instead, load balancers only have visibility on the status of their own requests. To the best of our knowledge, this specific model has not yet been theoretically addressed in the existing literature.

We note that a seemingly similar model has recently been studied by relevant papers [12] [13] [14] that deal with time-slotted multi-dispatcher (i.e., multi load balancer) systems. For
these systems, (i) load balancers make routing decisions at the same time, and (ii) operate on a “fire-and-forget” basis, lacking knowledge about the completion of the request by the server. These two unique features raise challenging problems discussed in Sec. VI that have led researchers to model and solve them with specific load balancing policies and low overhead server feedback schemes. However, these problems do not arise in service meshes, as these are asynchronous systems, in which routing decisions are made by load balancers at the arrival of the request; furthermore, the L7 proxies of service meshes operate on a request-response basis, providing implicit feedback at the completion of the request. Thus, the use of results presented in [12] [13] [14] to model microservice-to-microservice interactions is at risk of showing nonexistent problems, thus compromising the accuracy of the analysis and the effectiveness of related solutions.

**Contributions:** Using a new asymptotic analytical model and experimental evaluation, this paper shows that as the level of microservice replication increases, the average latency of user requests provided by LOR(d) policy tends to approach that achieved by a random policy.

Practically, the effectiveness of a service mesh with LOR(d) policy in reducing request latency decreases during the horizontal scaling of the application. This asymptotic loss of performance occurs because the latency of a user request is the result of a chain of microservice-to-microservice interactions. Each of these interactions can be modeled as a non-collaborative distributed supermarket model. For this model, our theoretical analysis shows that the latency converges to that of a random policy as the number of load balancers increases.

In addition to identifying and modeling the problem, we also propose a first practical solution to restore the effectiveness of LOR(d) for service mesh applications. Our solution is based on a new Kubernetes custom resource called Proxy-Service, whose benefits and limitations are evaluated by using benchmark microservice applications.

**Research methodology:** In Sec. II we provide an up-front experimental demonstration of the loss of LOR(d) effectiveness as a microservice application scales horizontally.

Once the existence of the problem has been confirmed in a real-world context, we move on to motivate it by analytical means, provide extensive experimental support, and, finally, devise an initial practical solution named Proxy-Service.

Specifically, in Sec. III we present a theoretical analysis of the non-collaborative distributed supermarket model and show that the average latency tends asymptotically to that of a system using a random policy. The theoretical analysis focuses on modeling the latency of microservice-to-microservice interactions. However, it does not capture the latency associated with sequences of microservice-to-microservice interactions, as those taking place in the processing of user requests within multi-tier microservice applications. In this regard, in Sec. VI we present an experimental evaluation of the LOR(d) policy for several benchmark microservice applications with a twofold objective: first, to demonstrate that the conclusions drawn from the theoretical analysis are still valid for multi-tier applications, and second, to evaluate the benefits and limitations of the Proxy-Service in restoring the latency advantage of LOR(d).

Finally, in Sec. VI we comment on the differences between our work and that of the literature, and in Sec. VII we draw conclusions.

**II. EXPERIMENTAL EVIDENCE OF THE PROBLEM**

We provide evidence of reduced effectiveness of the LOR(d) policy at the increase of the number of replicas using a two-tier microservice application made of a microservice $S_0$ and a microservice $S_1$, such as that in Fig. 2 but without the third tier made of microservice $S_2$. The application has been built using the µBench software [15] and runs in a Kubernetes cluster using the Istio service mesh. We considered a variable number $M$ of replicas of the microservice $S_0$ and $N = 20$ replicas of the microservice $S_1$.

Each user request is an HTTP GET forwarded by the ingress gateway to an instance $S_{0i}$ of the microservice $S_0$. When the request arrives at $S_{0i}$, the instance does not perform any complex internal processing and simply sends another HTTP request to an instance $S_{1j}$ of $S_1$. When $S_{1j}$, receives the request, it performs a CPU-intensive internal processing, at the end of which it sends a response to $S_{0j}$, which in turn responds to the ingress gateway that transfers the response to the user. The application bottleneck, that is, what introduces the most of the request latency, is the microservice $S_1$, even in the case of a single replica of $S_0$.

Since $S_0$ is unloaded, the increase in the number of $S_0$ replicas does not provide any CPU related latency reduction, but only increases the number of parallel load balancers that distribute the request stream to the instances of $S_1$. This allows isolating the impact of the load balancing policy on latency as the number of replicas of $S_0$ varies.

We loaded the application with a request flow of 100 req/s generated by the Jmeter software [16] and considered two load balancing policies offered by Istio: random and LOR(2). The results in Fig. 3 show that with an increase in the number of $S_0$ replicas increases, the Least Outstanding Requests policy loses effectiveness as the request latency asymptotically approaches that resulting from the random policy. In the next section, we provide analytical support for this finding.
III. 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 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. 4 depicts an example of this model, in which requests handled by different load balancers have different colors. From the point of view of the load balancer 1, server 1 has 2 outstanding requests, server 2 has zero outstanding requests, and server \( N \) has 1 outstanding requests. These values of outstanding (queued) requests are used to make dispatching decisions and, as previously noted, they are different from the exact amount of queued requests on the servers.

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 [11].

Consider a generic load balancer \( B \) at time \( t \), e.g., the first load balancer in Fig. 4. 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 \( i \) requests from \( B \) is equal to \( s_i^d \), i.e., the probability that the \( d \) servers chosen at random all have at least \( i \) 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 \( N\lambda(s_i^d - s_i^d)dt \).

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, \( \mu_i \) 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.

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

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

\[
\frac{ds_i}{dt} = \lambda(s_i^d - s_i^d) - \mu_i(s_i - s_{i+1})
\]

Each server receives an overall rate of requests equal to \( M\lambda \) and the whole system is stable if \( M\lambda < 1 \). 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.

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

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

\[
s_0 = 1
\]

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,

\[
\lambda = \sum_{i=1}^{\infty} (s_i - s_{i+1})\mu_i
\]

where \( \lambda \) 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 \( \mu_i \) is the departure rate of \( B \) requests in this condition.

To compute \( \mu_i \), 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 [11] [17]. Consequently, we assume that when a request of \( B \) enters a

\[1\]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 [17] to justify the same stability condition for single load balancer JSQ(\(d\)) systems.

\[2\]In [11], the Author solved this recursion in closed form for \( \mu_i = 1 \). However, in our non-collaborative distributed supermarket model \( \mu_i \) is not equal to one and varies with \( i \). Therefore, his elegant solution is unfortunately not applicable.
queue, it finds \((M - 1) E_q\) requests from other load balancers, where \(E_q\) is the average number of requests of a load balancer in a queue. Therefore, since we consider an average service time \(T_s = 1\), \(\mu_i\) can be approximated as follows.

\[
\mu_i \approx \frac{i}{i + (M - 1) E_q}
\]  

(6)

Regarding \(E_q\), it can be readily written as \([11]\).

\[
E_q = \sum_{i=1}^{\infty} s_i
\]  

(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 \((M - 1) E_q\) requests of the other load balancers, with unit service time. Subsequently, the average request latency can be written as:

\[
T = \sum_{i=1}^{\infty} (i - 1) (E_q) (s_{i-1}^{d} - s_{i}^{d})
\]  

(8)

\[
= (M - 1) E_q + \sum_{i=0}^{\infty} s_i
\]  

(9)

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 \([3]\).

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.,

\[
T \rightarrow \frac{1}{1 - \rho} \quad \text{as} \quad M \rightarrow \infty
\]  

(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.,

\[
s_i \rightarrow 0 \quad \text{as} \quad M \rightarrow \infty, \quad \text{for} \quad i > 1
\]  

(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.

\[
s_1 \rightarrow \frac{\lambda}{1 - \lambda M} \quad \text{as} \quad M \rightarrow \infty
\]  

(11)

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

\[
T \rightarrow 1 + M s_1 \quad \text{as} \quad M \rightarrow \infty
\]  

(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 \([19]\).

D. Simulation Results

We developed a simulator of the non-collaborative supermarket model to evaluate the validity of the theoretical analysis. We loaded the system with a Poisson request stream with an average service time \(T_s = 20\) ms and request frequency \(\lambda\) so that the utilization factor \(\rho\) is 75%, i.e., \(\lambda = \rho/(MT_s)\). The number of servers is kept constant at \(N = 40\). Fig. 5a shows the results for the LOR(2) policy.

Simulation and model results are quite similar and get closer as the number of load balancers increases. The latency of the LOR(2) policy tends to that of a random policy as \(M\) increases, that is, \(T_s/(1 - \rho) = 80\) ms. This demonstrates the validity of the analytical model and the asymptotic conclusions of Theorem 1.

In all simulations, the mean-field approximation in Eq. 6 leads to a slight overestimation of \(E_q\) and consequently of the latency. This overestimation decreases as \(M\) increases. This observation is supported by the inspection of simulation results, which we have not reported for the sake of brevity. The asymptotic accuracy of the model arises from the fact that the exact calculation of \(\mu_i\) in Eq. 6 would require the use of the expected value of the number of requests from other load balancers conditioned on the fact that the number of requests from the load balancer \(B\) is equal to \(i\). This dependence on \(i\) fades away as \(M\) increases, and thus the conditioned expected value converges to the unconditioned expected value \((M - 1) E_q\) used in Eq. 6.

Fig. 5b shows the result of a simulation in which the number of load balancers is kept constant at \(M = 1\) and \(M = 20\) while the server utilization factor \(\rho = \lambda M\) increases. As already noted in the literature for JSQ\((d)\) \([19]\), also for LOR\((d)\) the latency reduction with respect to a random policy is more relevant for heavy-traffic regimes, that is, at the increase of the load \(\rho\). 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. 5c shows the result of a simulation in which the number of choices $d$ varies. The figure allows us to derive that the LOR($d$) policy is also characterized by the “power of two random choices”. The main reduction in latency occurs for $d = 2$, with smaller improvements increasing $d$. In addition, the ineffectiveness of increasing $d$ beyond 2 is greater as $M$ increases. Indeed, the request latency flattens asymptotically toward that of a random load balancer, thus progressively losing the dependence on the parameter $d$.

IV. PROXY-SERVICE

Proxy-Service (PS) is a novel Kubernetes custom resource aimed to recover the effectiveness of LOR($d$) policy for microservice applications where replication is massively used.

The analysis in the previous section shows that the smaller the number of load balancers the more effective the LOR($d$) policy is. Based on this observation, the intuition behind the concept of Proxy-Service is shown in Fig. 6.

For each microservice $S$, there exist $M_{PS}$ instances of a Proxy-Service that intercept all incoming requests of $S$ and distribute them to the instances of $S$ using a LOR($d$) policy.

Let us assume that $M$ microservice instances generate the request stream received by $S$. With a Proxy-Service in the middle, such microservice-to-microservice interactions resam-ple to a non-collaborative distributed supermarket model with the number of load balancers changing from $M$ to $M_{PS}$. The value of $M_{PS}$ is much lower than $M$ because the Proxy-Service performs only load balancing operations with very low complexity. Therefore, $M_{PS} << M$ and LOR($d$) reduces latency better. It is likely that for many applications, a single Proxy-Service per microservice is sufficient, i.e., $M_{PS} = 1$, thus fully recovering the effectiveness that a LOR($d$) loses on large-scale applications.

A Proxy-Service unfortunately introduces two additional delays in microservice-to-microservice interactions. First, a transmission delay due to the stretch of the network path between the interacting instances. Second, a processing delay due to the execution of the load balancing operation.

When the capacity of the data center network is very high compared to the data rate generated by a microservice, the additional transmission delay is limited. Also, the additional processing delay is greatly reduced by using autoscaling mechanisms that replicate the Proxy-Service when the CPU is overloaded.

Although very small, these two additional delays could still worsen the latency of user requests in low-traffic regimes. In these cases, the benefit of reducing the load balancers from $M$ to $M_{PS}$ does not provide such a large latency reduction to compensate for the small additional delays. However, for a low-traffic regime, the latency of the user request is expected to be far below the service level objective, so an increase of a few ms should not be a problem.

Finally, it is important to highlight that if the advantages in load balancing brought about by a Proxy-Service for a microservice fail to offset the associated additional delays, it may be prudent to deactivate the Proxy-Service specifically for that microservice. This implies the possibility of configuring service mesh applications wherein only a selected subset of microservices is facilitated by a Proxy-Service.
A. Implementation Detail

We conclude the section by providing some details about Proxy-Service implementation [20], which assumes technical knowledge of Kubernetes and Istio [4] [6]. The Proxy-Service is a Kubernetes Custom Resource made of a combination of a Kubernetes Deployment, Service and Horizontal Pod Autoscaler (HPA), and an Istio Gateway, Virtual Service and Destination Rule. The life cycle of these resources is managed by a Proxy-Service Operator.

Consider a microservice $S_1$ that has several instances/Pods grouped by a Kubernetes Service whose name is $SN1$. When a Proxy Service for $S_1$ is activated, the Operator performs the following operations:

- creates a new Proxy-Service Kubernetes Deployment, named $PS-S1$, consisting of a Pod running only an Istio sidecar proxy whose service endpoints are managed by the Istio control plane [Fig. 2]. This Pod is actually an Istio Ingress Gateway;
- creates a new HPA associated with $PS-S1$ so that the Pods CPU usage is limited to a specific value, e.g. 70%;
- rename the Kubernetes Service that groups $S1$ Pods from $SN1$ to $SN1$-ep;
- creates a new Kubernetes Service named $SN1$ that groups the Pods of $PS-S1$. In this way, the $PS-S1$ Pods transparently intercept all calls to the microservice $S1$;
- create a new Istio Gateway, Virtual Service and Destination Rule so that requests for $SN1$ received by the $PS-S1$ Pods are forwarded to the Pods grouped by $SN1$-ep using a LOR(2) policy (alias, LEAST_REQUEST for Istio).

Proxy-Service resources are automatically managed by the Istio and Kubernetes control planes, thus reliably following the dynamics of the microservices in the cluster.

V. EXPERIMENTAL RESULTS

This section reports the results of an experimental campaign using a Kubernetes cluster consisting of 6 worker nodes with 8 CPUs each. The purpose is two-fold. First, to provide evidence that the modeling conclusions are valid not only for the interaction between two microservices, but also for chains of interactions that contribute to the final latency of a user request. Second, evaluate the effectiveness of the Proxy-Service.

Benchmark Applications: The first experimental campaign uses benchmark applications generated by µBench [15]. We consider two example applications. The first, shown in [Fig. 7a], represents an application with a "hub-and-spoke" dependency graph. The sequence of HTTP interactions to serve a user request is as follows: $S0\rightarrow S1$, $S0\rightarrow S2$, $S0\rightarrow S3$. The second application, shown in [Fig. 7b], represents an application with a "chain" dependency graph. The sequence of HTTP interactions to serve a user request is as follows: $S0\rightarrow S1$, $S1\rightarrow S2$, $S2\rightarrow S3$.

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 millis. Specifically, 300 millis is the value of Kubernetes CPU Request and Limit of each microservice Pod.

User requests are generated by JMeter [16]. 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 throughputs 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.

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

<table>
<thead>
<tr>
<th>$R$</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>hub-and-spoke chain</td>
<td>9.3</td>
<td>17.7</td>
<td>38</td>
<td>72.5</td>
<td>99</td>
<td>122</td>
</tr>
</tbody>
</table>

Fig. 7c and Fig. 7d show the latency of user request for the hub-and-spoke and chain µBench applications, respectively. We note that 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 supermarket model still holds for a whole microservice application.

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 [11] whose performance depends only on the load $\rho$, in the case of Poissonian interarrivals. In the different tests, we noticed that $R$ instances/servers have approximately the same CPU load ($\rho$). Therefore, as $R$ increases, the performance remains approximately constant, as predicted by the model, despite the fact that in our case we do not have Poissonian interarrivals.

Sockshop: We made a final performance assessment using the demo Sockshop e-Commerce application [21]. It is a rather heterogeneous application whose microservices use different programming languages and databases. All microservices communicate using REST over HTTP. The dependency graph is shown in [Fig. 8a]. The figure also includes the number of replicas we used per microservice, e.g., the front-end microservice has 30 replicas. Where not indicated, there is a single instance.

Fig. 8b and Fig. 8c show the average request latency and the 95th percentile (P95) varying throughput of requests. Requests perform different tasks, such as getting the socks catalog, adding a sock to the cart, ordering, logging in, logging out, etc. Consistent with the result of Tab. II, even for a whole microservice application, we note that the effects of the LOR policy with respect to a random one occur under heavy-load traffic and in these conditions, the introduction of Proxy-
Services significantly reduces the average and P95 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 [11] 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) [10], 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 [11].

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 [17] [22], or different scheduling, such as processor sharing [23]. Other works analyzed different properties of the JSQ. Other papers focus on a deeper performance evaluation. For example, in [24] [25], 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 [26].

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 [12], 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 [12], 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 [13], 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 [12] [13]. To solve this problem, in [14], 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 management 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, we believe that our non-collaborative distributed supermarket model is a bit more closely aligned with the use cases of microservice applications using service meshes with LOR(d) policy. 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 that uses 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 [27], 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 [28], the Authors studied basic load balancing and QoS-aware among interdependent IoT microservices. Similarly to [27], 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 dependendencies 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 [29], 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

Least Outstanding Request is undeniably an effective load balancing policy that combines simplicity and effectiveness in limiting request latency. As a result, popular service mesh software implements and uses it by default. However, our study unveils that in scenarios where multiple non-collaborative load balancers employ this policy, such as microservice applications with numerous replicas that use a service mesh, the efficacy of latency reduction diminishes. The reason lies in the fact that each load balancer knows only the number of its own pending requests that a server is handling, rather than the total number of requests in the server queue. This knowledge error grows as the number of load balancers increases, leading the policy to make progressively less informed decisions, with performance tending asymptotically to that achieved by a random load balancing policy.

Proxy-Service is an initial solution to restore the effectiveness of the Least Outstanding Request policy for service mesh applications. It provides that requests addressed to a microservice are routed to one or more load balancers, called Proxy-Services, which then distribute them to replicas of the microservice using the Least Outstanding Request policy. This centralization reduces or even eliminates, in the case of a single Proxy-Service, the knowledge error on the server queue length, thus restoring the optimal performance of the load balancing strategy. However, centralization comes with obvious risks, such as the lengthening of microservice-to-microservice network paths and the possible CPU overhead of the Proxy-Service. If these risks do not arise, the Proxy-Service proves to be very effective in the field.

This study has revealed a new problem of an important emerging technology, namely service meshes, and future works could focus on solutions directly embedded in sidecar proxies, thus avoiding any centralized mediation. We believe that future solutions should also place great emphasis on implementation feasibility, as well as easy use by developers of microservice applications, features for which, for example, the Least OutStanding Request policy is widely used, while other more complex policies are not.

REFERENCES