Optimizing Service Selection and Load Balancing in Multi-Cluster Microservice Systems with MCOSS

Daniel Bachar
Dept. of Computer Science
Reichman University, Israel
bachar.daniel@post.runi.ac.il

Anat Bremler-Barr
Dept. of Computer Science
Tel-Aviv University, Israel
anatbr@tauex.tau.ac.il

David Hay
Dept. of Computer Science
The Hebrew University, Israel
dhay@cs.huji.ac.il

***Abstract*—With the advent of cloud and container technologies, enterprises develop applications using a microservices architecture, managed by orchestration systems (e.g. Kubernetes), that group the microservices into clusters. As the number of application setups across multiple clusters and different clouds is increasing, technologies that enable communication and service discovery between the clusters are emerging (mainly as part of the Cloud Native ecosystem). In such a multi-cluster setting, copies of the same microservice may be deployed in different geo-locations, each with different cost and latency penalties. Yet, current service selection and load balancing mechanisms do not take into account these locations and corresponding penalties. We present *MCOSS*, a novel solution for optimizing the service selection, given a certain microservice deployment among clouds and clusters in the system. Our solution is agnostic to the different multi-cluster networking layers, cloud vendors, and discovery mechanisms used by the operators. Our simulations show a reduction in outbound traffic cost by up to 72% and response time by up to 64%, compared to the currently-deployed service selection mechanisms.**

***Index Terms*—Optimization, Kubernetes, Cloud Computing, Multi-Cloud, Multi-Cluster, Microservices, Load Balancing**

I. Introduction

Microservice architecture, which has become prevalent nowadays, consists of splitting an *application* (e.g. from a monolithic architecture) into a set of smaller, interconnected, loosely-coupled logical units called *microservices* (or simply *services*). Services are typically deployed in the cloud as a group of containers (the Kubernetes jargon refers to it as pods [1]) within *clusters* in different geographic locations and clouds. Microservice architecture and specifically, cloud-native microservice architecture, provides numerous benefits over monolithic architecture, such as robustness [2], elasticity [3], security [4], redundancy, and scalability.

Nevertheless, this architecture poses several key challenges such as how to control and manage the services, how to route data between the services of the same application, and how to schedule jobs on services for competing applications. To address these challenges, *orchestration and management systems*, such as *Kubernetes* [5], have emerged. One of the critical challenges, which is the focus of this paper, is *how to manage communication traffic between services* which, in turn, includes service discovery and load balancing. This is done by *cloud native networks* (namely, overlay networks on top of existing networks, e.g. [6], [7]) and *service mesh* platforms (such as Istio [8] and Linkerd [9]), which along with Kubernetes, have become some of the most critical infrastructure components of the cloud-native stack [10].

Most cloud-native platforms have been designed to work under a *single* cluster. Yet, recently, organizations are starting to shift to *multi-cluster deployments*, either across geographical areas, across cloud providers, or hybrid on-premises/public cloud deployments. Such multi-cluster deployments hold great promise as they may offer higher availability, conformance to local privacy laws, better performance, and lower operational costs. Stoica and Shenker recently coined the term *sky computing* [11] to refer to deployments across multiple clouds and argued that, unlike the Internet and telephony, there is still a way to go before the cloud becomes a *commodity*. In this paper, we provide both architectural and algorithmic building blocks for one of the most important challenges of multi-cluster deployment: optimized service selection across multiple clusters.

Specifically, multi-cluster (and multi-cloud) deployments can be classified into two categories: *replica-based* and *service-based*. Replica-based multi-cluster deployments run complete copies of all services on different clusters, where the first requests (namely, by the users of the application) are directed to one of the clusters and may trigger a chain of requests *within this cluster*. On the other hand, service-based multi-cluster deployments are more flexible and run some services in some clusters, implying traffic/requests may be routed *across clusters*. While in most cases, two services that tend to communicate frequently will be located in the same cluster. In some cases, this does not hold either due to failures or placement considerations (e.g. compliance with privacy regulations such as GDPR and others). While in a single-cluster deployment, latencies and communication costs between services tend to be comparable (and small), sending a request from some service i to service j at one cluster or another, may result in completely different latency, and response time, and billing costs. This implies that multi-cluster service-based deployments pose a new *service selection*

Part of this work was done while A. Bremler-Barr was with Reichman University.

ISBN 978-3-903176-57-7

question and a new *load balancing* problem.

Specifically, all solutions to this problem should meet the following six requirements. *(i) Interoperability:* The solution should be implemented seamlessly into current microservice orchestration systems and cloud-native stacks and avoid changing the current APIs and implementation details. Furthermore, it should be agnostic to the cloud provider, implying it can be used to connect clouds (e.g., as in sky computing); *(ii) Flexibility:* Customers should have a sufficiently expressive interface to specify policies based on performance, load, capacity, traffic pricing, etc.; *(iii) Connection persistence:* Packets of the same connection should always be directed to the same service at the same cluster [12]; *(iv) Resilience:* The system should be resilient to outages both in the control and data planes and continue to work even with connectivity issues; *(v) Responsiveness:* The selection system should respond to changes such as service availability, load, and policy updates; and, *(vi) Scaling:* The system should support the scaling mechanism.

Currently, Kubernetes supplies an API for multi-cluster deployments and management but has no native support for service selection and multi-cluster load balancing. The support is added by solutions like *Submariner* [6], *Linkerd* [9], or *Istio* [8], but none of them have an optimized solution to a service selection (and load balancing) that works hierarchically, completely separating intra- and inter-cluster decisions (and therefore, can interoperate with any mechanism).

In this paper, we present *Multi-Cluster Optimized Service Selection (MCOSS)*, which meets all of the above requirements. First, MCOSS is focused on the common microservice architecture of Kubernetes, which is *hierarchical* (See Fig. 1): A proxy (e.g., `kube-proxy`) is responsible for load-balancing between the various pods (containers) within a single cluster, while an additional service selection process deals with load-balancing across clusters (the collection of all pods running the same service at the same cluster is called a *service instance*, thus the second service selection is done between service instances). In contrast, current service mesh architectures perform load-balancing using a Sidecar that is attached to each container and consider all Sidecars in a flat manner.

The data-plane of our architecture enhances the DNS process to perform an optimized service selection. As illustrated in Fig. 1, a pod in a service instance of Type 1 that requests a service instance of Type 2, sends a DNS request to a DNS resolver in the cluster, and receives the IP address of the *selected* service instance of type 2. By leveraging DNS, our architecture meets the requirements of *connection consistency* and *interoperability*. MCOSS's load balancing involves splitting outgoing requests (of the same type) from one cluster across several other clusters. This is done by implementing a weighted round robin in the DNS resolver, where each weight $w(c \to c', t)$ corresponds to the fraction of requests for service type t, originating from cluster c, that should be sent to cluster c' (out of all type t requests from cluster c). If cluster c' is chosen, then the DNS resolver in cluster c replies to the DNS request with the specific IP address of the specific service in cluster c'. The next DNS request for service type t may be resolved to another cluster, according to the weights.

The control plane of MCOSS is described in Fig. 2. We implemented MCOSS over *Submariner* [6], which provides L3 network connectivity between Kubernetes clusters. The key component is our optimizer that resides in the Submariner broker component [13]. The optimizer receives as input the predicted traffic demand (number of requests to the next service)[1] and the relevant objective metrics (cost, load, etc.) from all the clusters. Then, it returns the corresponding weights, to the DNS resolver in each cluster. The optimization is done one hop at a time, taking into account only the next service and not the full-service chain (due to several reasons, which will be explained later). We note that the solution is *scalable* since the optimization problem depends on the number of service instances and not the number of pods in the system (due to the fact that MCOSS has a hierarchical load-balancing architecture). To deal with the *flexibility* requirement, we define an abstract cost function that allows operators to define the metrics to optimize the service selection. In our paper, we show two cost functions; one that optimizes latency and cost (i.e, monetary prices) and thus requires linear optimization, and one that optimizes response time and cost. We show it requires solving a quadratic (yet convex) optimization problem. Our architecture is *resilient* since the data-plane is decentralized and the control-plane is centralized but enjoys the built-in redundancy of the Broker as the code is shared between the clusters (if one is failing we can just choose another). In Section II-D we explain how MCOSS architecture is *responsive* by showing how and when to recalculate the weights.

We note that our work focuses on optimizing the service selection and assumes the placement of services is given [14], [15]. Previous works on service selection assume that the full-service chain (in the level of possible service instances in different clusters) is given to the optimizer [16]–[19]. Yet, relying on the service chain hinders the solution's interoperability and increases its complexity. In addition, the full-service chain (i.e. the entire possible service instance chains) cannot be retrieved easily from the Kubernetes framework. Moreover, We argue that our solution achieves similar (near-optimal) results even without knowing the service chain, as multiple inter-cluster services *selections* within one service chain are very rare in practice [20].

To conclude, our primary contribution is twofold: first, to the best of our knowledge, we are the first to formulate and offer an applicable decentralized data-plane for the microservice load balancing problem in a multi-cluster, multi-cloud environment. Second, we have designed and implemented a system, which integrates seamlessly with Kubernetes, supports different multi-cluster connectivity, and discovery solutions using plugins. Importantly, our solution is modular in two

[1]We note that predicting traffic demands is orthogonal to the service selection problem and is outside the scope of this paper.

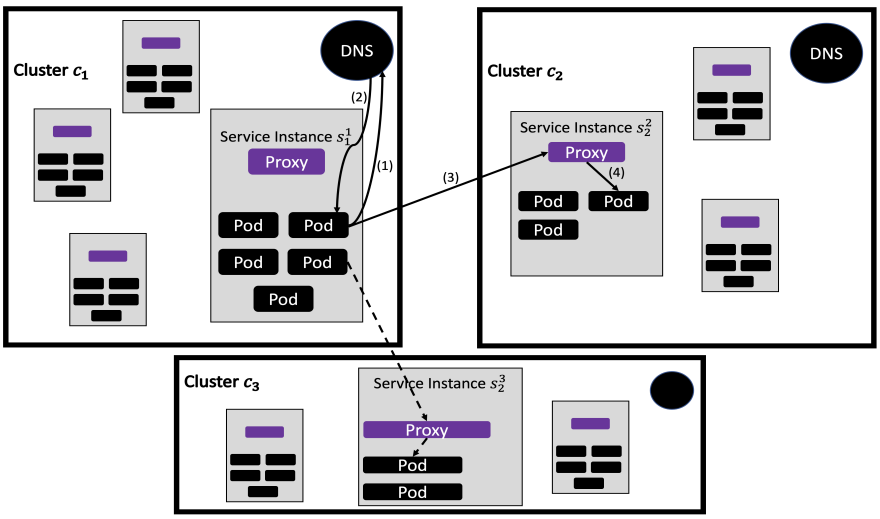


Figure 1: Data-plane design. For example, a service of Type 1 in Cluster c_1 sends a request to a service of Type 2, whose instances are deployed in Cluster c_2 and c_3. This is done by (1) sending a DNS request to a local DNS server in Cluster c_1; (2) receiving a DNS response with the IP address of a proxy (e.g., a kube-proxy) in one of the clusters; (3) sending a request to the proxy; (4) the proxy forwards the request to one of the pods running service Type 2. Inter-cluster load balancing is done by the DNS servers (namely, choosing different proxies for different DNS requests). Intra-cluster load balancing is done by the proxies.

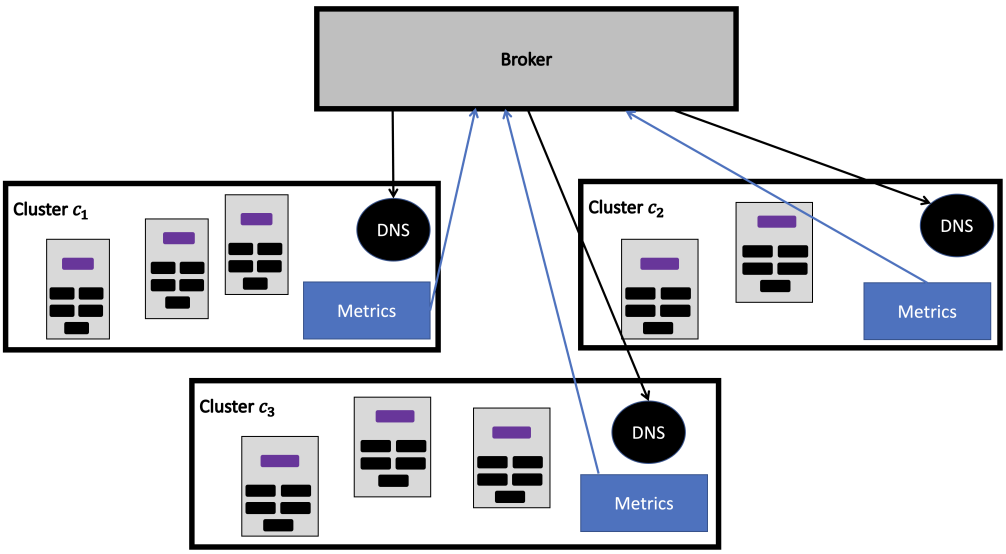


Figure 2: Control plane design. A logical-centralized *broker* collects metrics from the different clusters, solves the corresponding optimization problem, and populates the weights used by the DNS servers. The DNS server at cluster c_i, receives weights $w(c_i \rightarrow c, t)$ for each destination cluster c and requests for service type t, indicating the portion of the requests of type t that should be sent to cluster c.

senses. The optimization problem we solve does not depend on the data-plane architecture (namely, one can replace our DNS-based solution with a different one). In addition, the cost function we used is pluggable and can be changed to reflect the needs of specific customers. Moreover, it is dynamic and can be changed on-the-fly, for example, upon a change in cost model or customer needs.

II. The Multi-Cluster Service Selection Optimization Problem

The multi-cluster service selection problem lies at the heart of our system. Based on the inputs collected from the different clusters (namely, *the demand*), it strives to optimize the weights sent to the *load balancer* module, according to which, the load balancing between clusters and services is performed.

Importantly, our system and the corresponding optimization problem are working in a *hop-by-hop* manner, where only the next-hop destination of each request is considered for service selection. Moreover, as we look at the entire system together, we handle the interdependence of services over every single hop and accommodate competition between the different (requesting) services. We will use a simple toy example to demonstrate our model and its notations. In our example, the application is built with two services, where the first service

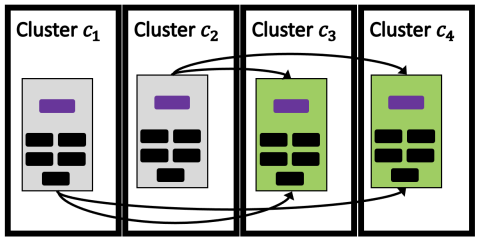


Figure 3: Illustration of our toy example. The service instances of the first (second) service type are in gray (green). Arrows represent situations where the traffic between service instances may be strictly greater than zero.

sends requests to the second one. The services are deployed in four different clusters as described in Fig. 3.

Let C be our *cluster* set, such that $c \in C$ is a single cluster representing some location in our mesh (i.e. geo-location, cloud provider, etc.). Let T be the set of all service *types* in the system, and let $S_t = \{s_t^c | c \in C, t \in T\}$ be the set of *service instances* of type t, where a service instance $s_t^c \in S_t$ is a logical abstraction of all pods in cluster c performing the function of type t. Finally, let $S = \bigcup_{t \in T} S_t$ be the set of all service instances. In our toy example, $C = \{c_1, c_2, c_3, c_4\}$, $T = \{t_1, t_2\}$, $S_{t_1} = \{s_{t_1}^{c_1}, s_{t_1}^{c_2}\}$, and $S_{t_2} = \{s_{t_2}^{c_3}, s_{t_2}^{c_4}\}$.

Our framework works in *epochs* of some predefined period of time. $cap(s_t^c)$ denotes the available capacity of a service instance s_t^c; namely, the number of requests that service instance s_t^c can handle over an epoch. Note that the service capacity might change over epochs, due to long-lived requests (that span across epochs or scaling mechanisms in the cloud). In some cases, a service instance also has a *minimal* traffic constraint (e.g., to make sure that a service instance is still viable). To capture this, we denote by $\alpha(s_t^c)$ the minimum number of requests that should be sent to service instance s_t^c over an epoch. The demand is captured by $N_{c,t}$, which is the total number of requests for service type t originating from cluster c (i.e. the total traffic to be distributed between the Service Instances of type t) over the epoch.

We strive to optimize some *abstract cost function*, denoted by $cost(c', s_t^c)$, which indicates the performance penalty between *any* service instance in cluster $c' \in C$ to some target service instance $s_t^c \in S$. Cost functions can incorporate the latency between clusters, the response time, the monetary price that should be paid between clouds, a combination of several metrics, or specific metrics that are important to a specific deployment. If cluster c' is not connected to cluster c or traffic cannot be sent between the two, then $cost(c', s_t^c)$ is set to infinity (see Sections II-A and II-B for more details). Thus, the generic optimization problem is formulated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{s \in S} \sum_{c \in C} n_{c,s} \cdot cost(c, s) \\ \text{s.t.} \quad & \sum_{s \in S_t} n_{c,s} = N_{c,t} && \text{for all } c \in C, t \in T \\ & \sum_{c \in C} n_{c,s} \leq cap(s) && \text{for all } s \in S \\ & \sum_{c \in C} n_{c,s} \geq \alpha(s) && \text{for all } s \in S \\ & n_{c,s} \geq 0 && \text{for all } c \in C, s \in S \end{aligned} \tag{1}$$

where $n_{c,s}$ is the number of requests sent from cluster c to service instance s (in another cluster).

Once the optimization problem is solved, the weight for each specific cluster c' to a service instance s_t^c is calculated as follows:

$$w(c' \to c, t) = \frac{n_{c', s_t^c}}{N_{c',t}}, \tag{2}$$

where $w(c' \to c, t) = 0$ if $N_{c',t} = 0$.

Furthermore, as we use *weighted round robin*, we allow the relaxation of the constraints and treat the variable $n_{c,s}$ as a real number. Note that if $w(c \to c, t)$ is the portion of traffic sent within the cluster; one can strictly prioritize local traffic by setting $cost(c, s_t^c) = 0$.

In our toy example, assume the costs are $cost(c_1, s_{t_2}^{c_3})$=1, $cost(c_1, s_{t_2}^{c_4})$=100, $cost(c_2, s_{t_2}^{c_3})$=10, $cost(c_2, s_{t_2}^{c_4})$=20, the capacities of all service instances are 100, and their minimal traffic constraints are 0. As only service instances of type t_1 send requests to service-instance of type t_2, the only non-zero demand values are N_{c_1,t_2} and N_{c_2,t_2}, which are set to 90 and 80, respectively. Solving the optimization problem yields[2]; $w(c_1 \to c_3, t_2) = \frac{n_{c_1, s_{t_2}^{c_3}}}{N_{c_1,t_2}} == 1$, $w(c_1 \to c_4, t_2) = \frac{n_{c_1, s_{t_2}^{c_4}}}{N_{c_1,t_2}} = 0$, $w(c_2 \to c_3, t_2) = \frac{n_{c_2, s_{t_2}^{c_3}}}{N_{c_2,t_2}} = 1/8$, $w(c_2 \to c_4, t_2) = \frac{n_{c_2, s_{t_2}^{c_4}}}{N_{c_2,t_2}} = 7/8$

It is important to note that our optimization problem can be split into $|T|$ different problems, one for each service type T, as the capacity constraints are on the target service instances (and not, for example, the communication links between clusters).

What customers consider "optimal" can differ: one may try to balance the load across all available services, while others may seek to minimize the latency or the billing costs. Some might want to combine several requirements, such as improving latency and throughput while keeping the cost as low as possible. The abstract cost function supports maximum flexibility and expressiveness for our customers, allowing them to define their own optimization. Some examples, and their consequences, are described as follows:

A. Mixing latency and billing costs

Let p_{c_1,c_2} be the monetary price (e.g., in USD) that needs to be paid for sending 1 GB of data from cluster c_1 to c_2, and let l_{c_1,c_2} be the latency between the clusters. To consider both metrics simultaneously (after normalization), we define a linear combination using the following cost function:

$$cost(c', s_t^c) = \frac{p_{c',c}}{P} \cdot \beta + \frac{l_{c',c}}{L} \cdot (1 - \beta), \tag{3}$$

where $\beta \in [0, 1]$ captures the relative weight between price and latency (larger β values imply price is more important), and $P = \max_{c1,c_2 \in C} p(c_1, c_2)$ and $L = \max_{c1,c_2 \in C} l(c_1, c_2)$ are the maximum price and latency and are used as normalization factors. In our toy example, assume the latencies

[2]all other weights are zero, by definition.

Table I: The effect of the value of β on the resulting weights in our toy example.

β	0	0.25	0.5	0.75	1
$w(c_1 \to c_3, t_2)$	1	1	0.778	0	0
$w(c_1 \to c_4, t_2)$	0	0	0.222	1	1
$w(c_2 \to c_3, t_2)$	0.125	0.125	0	0.875	0.875
$w(c_2 \to c_4, t_2)$	0.875	0.875	1	0.125	0.125

are l_{c_1,c_3}=1, l_{c_1,c_4}=100, l_{c_2,c_3}=10, l_{c_2,c_4}=20, and monetary prices are $p(c_1, s_{t_2}^{c_3})$=100, $p(c_1, s_{t_2}^{c_4})$=1, $p(c_2, s_{t_2}^{c_3})$=20, $p(c_2, s_{t_2}^{c_4})$=10. Table I shows how weights are changed with different values of β.

Note that in this synthetic toy example, latencies are inversely proportional to monetary prices. However, in practice, latencies are often proportional to monetary prices, reducing the effect of parameter β. See Section IV for more details. Moreover, this cost function results in a simple linear program, that can be solved efficiently. Yet, it assumes a constant latency between clusters that do not depend on the load on the service instance itself.

B. Considering response times

The response time of a request is the total time the request is in the system (i.e., from the time the originating service instance sends the request until it is fully processed). For a single-hop application, this includes the latency between the services, queuing when applicable, the processing time in the target service instance, and the latency required to send feedback to the originating service. The response times of multi-hop applications may become more complex to model, as calls to some services in the service chain may be done in parallel and some must be executed sequentially. These scenarios are out of the scope of this paper, as in practice service chains spanning more than two clusters, with multiple choices, are rare.

While some may model response time as a queuing system [21], we have taken a different approach and observed that, up to a certain threshold, there is a linear correlation between the response time and the load (namely, the number of requests) [20]. Beyond this threshold, most services will drop the request (e.g., by replying with an error message). As our load balancer strives to avoid such drops, we treat these thresholds as service instance capacities, and therefore we have the following cost function to capture the response time:

$$cost(c', s_t^c) = l(c', c) + prc(s_t^c) \cdot \sum_{c' \in C} n(c', s_t^c), \tag{4}$$

where $l(c', c)$ is the latency between the clusters, and $prc(s_t^c)$ is the slope of the linear relation between the processing time and load on the service instance, and $\sum_{c' \in C} n(c', s_t^c)$ is the load assigned to the service instance by our optimization problem. Note that as $l(c', c)$ is the y-intercept of the linear function, it can be easily extended to other metrics, such as monetary cost or a mix between latency and monetary cost, as was done in (3): $\frac{p_{c',c}}{P} \cdot \beta + cost(c', s_t^c) \cdot \frac{(1-\beta)}{L}$, where $cost(c', s_t^c)$ is the cost function in (4), and $L = \max_{c' \in C, s_t^c \in S}(l(c', c) + prc(s_t^c) cap(s_t^c))$.

In Section IV we discuss how $prc(s_t^c)$ and $cap(s_t^c)$ are determined. As the cost function in (4) depends on the actual load on service instances, the optimization problem, which we call RESPONSE TIME OPTIMIZATION, is a quadratic program (QP). The problem can be solved efficiently using a variety of convex optimization methods. The proof of the convexity of the problem is omitted for brevity. Recall that our generalized optimization can be split into T smaller, independent programs (one for each service type). This provides another boost in performance for our RESPONSE TIME OPTIMIZATION problem and makes it practical to solve in real-life applications. In our toy example, assume now that the latencies are l_{c_1,c_3}=0ms, l_{c_1,c_4}=1000ms, l_{c_2,c_3}=1000ms, l_{c_2,c_4}=0ms, capacities are $2000rps$ and demands are $N(c_1, t_2)$=$1000rps$ and $N(c_2, t_2)$=$100rps$. Assuming $pcr(s_{t_2}^{c_3})$=$pcr(s_{t_2}^{c_4})$=1, the corresponding weights are $w(c_1 \to c_3, t_2) = 0.8$, $w(c_1 \to c_4, t_2) = 0.2$, $w(c_2 \to c_3, t_2) = 0$, and $w(c_2 \to c_4, t_2) = 1$. This yields an average response time of $845.45ms$. For comparison, ignoring processing time, and solving the optimization program in (3) yields an average response time of $918ms$, while distributing the requests in a round-robin manner yields an average response time of $1050ms$.

C. Allowing service instances to scale out

One of the benefits of deploying services in the cloud is the ability of service instances to scale out to accommodate more requests. However, such scaling comes with additional costs, that should be taken into account in the optimization problem. One approach to tackle scaling out is to add an (integer) variable for each service instance, to determine how much its capacity constraint should be relaxed and how it affects the overall cost function. In this paper, we take a more restrictive approach and assume that at any epoch, for each service type, only a *single* service instance s can scale out by a predefined capacity $scap(s)$ at a predefined cost $scost(s)$. We use the fact that we can solve our generalized optimization problem (1) for each service type separately. Given a type $t \in T$ we solve the following $S_t + 1$ optimization problems (namely, the program in (1) with slight modifications), for each $s' \in S_t \cup \{\perp\}$, where, with a slight abuse of notations we define $cap(\perp) = scap(\perp) = scost(\perp) = 0$:

$$\begin{aligned}
\text{minimize} \quad & scost(s') + \sum_{s \in S_t} \sum_{c \in C} n_{c,s} \cdot cost(s, c) \\
\text{s.t.} \quad & \sum_{s \in S_t} n_{c,s} = N_{c,t} && \text{for all } c \in C \\
& \sum_{c \in C} n_{c,s} \le cap(s) && \text{for all } s \in S_t \setminus \{s'\} \\
& \sum_{c \in C} n_{c,s'} \le cap(s') + scap(s') \\
& \sum_{c \in C} n_{c,s} \ge \alpha(s) && \text{for all } s \in S_t \\
& n_{c,s} \ge 0 && \text{for all } c \in C, s \in S_t
\end{aligned} \tag{5}$$

The program for $s' = \perp$ represents the solution in which we do not scale any service instance of type t. After solving these $|S_t| + 1$ programs we can choose the one that minimizes the total cost, implying we know which service instance to scale (if any) and what are the corresponding weights. We repeat this

for any service type, resulting in a total of $\sum_{t \in T} \cdot (|S_t| + 1)$ programs, and the ability to scale out up to T service instances. In our toy example, with the parameters of Section II-A and given $scap$ is 5 for all service instances, the optimal solution will be to scale out $s_{t_2}^{c_3}$ if $scost(s_{t_2}^{c_3}) < 0.5$; otherwise, no service instance will scale out no matter what its $scost$ value is. In fact, this scale-out process will continue, step by step, at every invocation of the program until $cap(s_{t_2}^{c_3})$ reaches $170rps$, thus accommodating all requests for service type t_2. The example of Section II-B will not scale out, as there are no capacity bottlenecks.

D. When to recalculate the weights?

There are two approaches to determining when weights should be re-calculated and disseminated throughout the system.

The *periodic* approach recalculates the weights for every predefined amount of time τ, based on measurements done before that time interval. In general, as τ is smaller, the weights are closer to the optimal solution. However, a smaller τ value implies additional overhead on the system, as well as more frequent fluctuations in traffic. Additionally, traffic does not change significantly over time, as such these calculations may be redundant.

On the other hand, the *reactive* approach recalculates the weights, only if the *objective function* deviates significantly from the forecast (after scaling). Specifically, assume the weights were calculated at some time τ_0. Let $N = \sum_{c \in C, t \in T} N_{c,t}$ according to which the weights were calculated, and $O = \sum_{s \in S} \sum_{c \in C} n_{c,s} \cdot cost(c, s)$ is the resulting value of the objective function. At some time $\tau_1 > \tau_0$, we calculate $N(\tau_0, \tau_1) = \sum_{c \in C, t \in T} N_{c,t}(\tau_0, \tau_1)$, where $N_{c,t}(\tau_0, \tau_1)$ is the *actual* number of requests of type t from cluster c between time τ_0 and τ_1. Furthermore, let O' be the *actual* cost (defined by the $cost$ function) of delivering the traffic over that time interval. Note that calculating O' does not involve resolving the optimization problem, as we use existing weights; yet it requires collecting the metrics used for the specific cost function (recall Fig. 2). As these metrics are collected continuously, one can calculate O' very frequently. We set a threshold $\gamma > 1$, and trigger weight recalculation when

$$\frac{O'}{N(\tau_0, \tau_1)} > \gamma \frac{O}{N}.$$

This implies that weight calculations are done only when needed, and may be triggered upon a major event (e.g., failures or unpredictable peaks in traffic). Notice that the larger γ is, the less sensitive the recalculation process becomes. In addition, one can set a lower threshold $\gamma' < 1$ to indicate that the original solution was too pessimistic (namely, when $\frac{O'}{N(\tau_0, \tau_1)} < \gamma' \frac{O}{N}$). In many cases, this indicates significant changes in traffic patterns since τ_0, implying the recalculation might yield an even better solution.

Recall our toy example with the parameter and cost function detailed in Section II-B. Suppose now that at some time, demand is starting to fluctuate: in each 20 minutes interval, the first 10 minutes period has a fixed demand of $N(c_1, t_2)$=1000 and $N(c_2, t_2)$=100, while the second 10 minute period has a fixed demand of $N(c_1, t_2)$=100 and $N(c_2, t_2)$=1000. If one does not react to these changes in demand, the average response time will also fluctuate, as shown in Fig. 4. However, weights recalculation as a reaction to these changes (e.g., with $\gamma = 1.1$ or, in this case, every 10 minutes as demand changes periodically), will keep the average response time at its optimal value of $845.45ms$. We note that when there are capacity bottlenecks, failing to promptly adapt to the demand may also result in request drops.

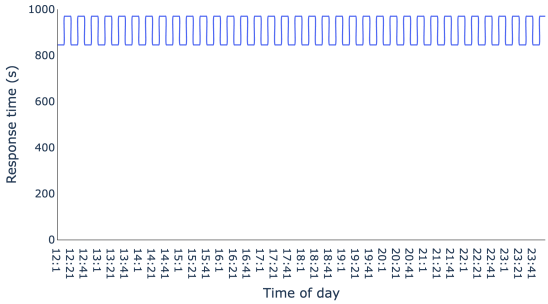


Figure 4: Toy example performance under fluctuating demand, without weight updates.

III. Implementation Details

The system is implemented as a management layer above Submariner [3]. Submariner provides a solution for network connectivity and service discovery (namely, as in our data and control planes) but lacks an efficient service selection mechanism, which MCOSS provides.

Our management layer uses native Kubernetes APIs to perform several tasks.

First, MCOSS supports *weighted round robin* at the DNS level. This is done through Submariner's CoreDNS plugin, named Lighthouse. Lighthouse customizes CoreDNS and facilitates DNS service discovery in multi-cluster connected environments. We have added weighted round-robin support to Lighthouse and turned off the CoreDNS caching layer so that all queries arrive at the plugin. We run 10,000 queries to the CoreDNS servers with and without the caching layer. Our testing shows that DNS response time has increased by $4ms$ on average, which is negligible compared to the response time improvement we gained by MCOSS. Moreover, the Submariner system supports an opt-in mechanism where you define a specific FQDN that only if used, the request will be routed to the DNS plugin.

Second, MCOSS collects metrics (such as demands and latencies) from each cluster. This is done by using Prometheus , which is an open-source monitoring solution, which queries Submariner to extract the inter-cluster latency and CoreDNS to extract the demands.

Third, and at the heart of our solution, there is the optimization module which resides within the logically-centralized broker (recall Fig. 2). MCOSS polls the different Prometheus services, as well as cloud providers' APIs, to receive the information needed for solving its optimization problem. After the problem is solved, the optimization module disseminates

[3]See our implementation code at [22].

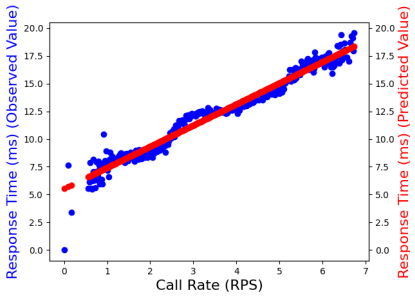


Figure 5: Example of a single service (Service $0023\ldots3ec6$ from [25]), whose response times (in blue) correspond to our MCOSS-QP approximation (in red).

the weights by creating an event to which all clusters listen. We note that Lighthouse plugins listen to events from the broker anyway (e.g., to get information about newly-exported services), and MCOSS just adds additional metadata to these events. We used Grobi [23] for the QP-Optimization, and PuLP [24] for the LP-Optimization.

IV. Evaluation

In this section, we first investigate, under different settings, the influence of load on the service instance's average response time. These results show that a simple offline experiment can provide good estimates for the values of $prc(s_t^c)$ and $cap(s_t^c)$, which later can be plugged into our optimization program. Then, we show how beneficial MCOSS is for reducing the total $cost$, compared with round-robin selection, greedy selection, and other techniques. We also show how MCOSS affects the response time when response times are not considered directly in the objective function (i.e., when we use the cost function of (3), which results in a linear program, instead of the one of (4), which results in a quadratic program). Finally, we discuss the convergence time of our optimization problems and show that it is negligible, even with large deployments that are unlikely to materialize in the foreseeable future.

A. Response time and load

Alibaba Group has recently analyzed traces of more than ten billion call traces among nearly $20,000$ microservices in a 7-day time frame [20]. One of their conclusions is that the response times of most microservices are stable even when the call rate (load) varies. This is due to most calls in Alibaba clusters can be processed immediately without any queuing delay (The CPU utilization is less than 10%, even for large loads). This implies that our *MCOSS-LP* model is the right one to choose (rather than, the more computing-intensive, but more accurate, *MCOSS-QP* model). However, a closer look into the traces shows that the response time of some services depends linearly on their load (namely, a linear regression results an R-value greater than 0.7 and a regression line whose slope is greater than 0). For these services (see an example taken from Alibaba's dataset [25] in Fig. 5), our *MCOSS-QP* model is a better approximation.

Table II: Service Instance locations and capacity in RPS

Service / Cluster	gcp-1	gcp-2	aws-1	aws-2	aws-3
Rating	70,000	70,000	70,000		
Review	70,000	70,000	70,000		
Details	70,000	70,000			70,000
Product Page	30,000	30,000	30,000	30,000	30,000

To conclude, both MCOSS-QP and MCOSS-LP are supported by the industry's most important criteria, literature, real trace analysis, and experimentation.

B. MCOSS reduces response time and prices

We have conducted numerous simulations and experiments, using different applications and different placements of service instances.[4]. First, we focus on one specific application, Bookinfo [8], which is a common benchmark in literature for microservice architecture [8], [19], [27]. BookInfo consists of 4 service types, named *product-page*, *review*, *details*, and *rating*. The *product-page* instances, send requests to the instances of *review* and *details*; The *review* send requests to the *rating* instances; while the *details* instances do not propagate any further requests.

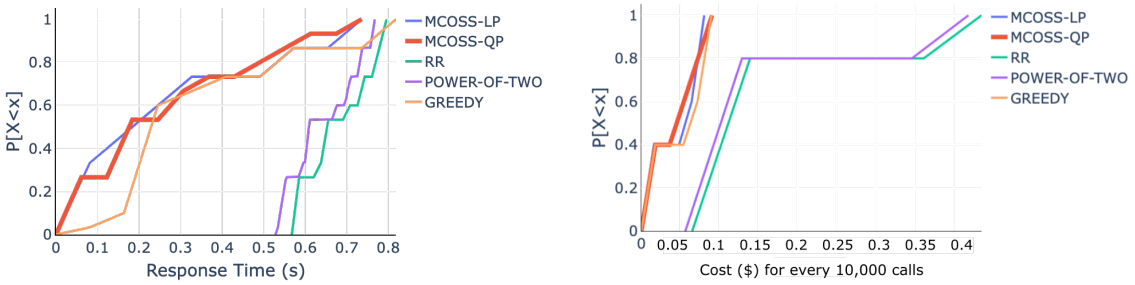
We have deployed BookInfo in 5 clusters across North and South America, two in Google Cloud and three in Amazon Web Services (mentioned in Table II). Pricing and latencies between these clusters are publicly-available [28]–[31]. We assume that the latency between clusters (region/zones) is relatively steady on average (besides edge cases) [14]. Nevertheless, we introduce noise (uniformly around the average latency) per request for more realistic behavior [32]. The different services are deployed only in a subset of the clusters, and their capacities are shown in Table II. We assume traffic is admissible with 28,000–35,000 RPS [33], [34] for each instance of the front-end *product-page* service.

Fig. 6 describes the performance of BookInfo in terms of response time and cost. For $\beta = 0$, our experiments show up to 64% improvement over other load balancing techniques. While for $\beta = 1$, our experiments show a 72.82% improvement, reducing the mean cost from 141.50 USD per 1 million requests to 38.46 USD per 1 million requests (assuming the average request size is 50 KB). Importantly, the difference between the two objective functions of (3) and (4) is negligible, implying that it may be useful to use the faster and less computationally-intensive linear program, with almost no performance penalty.

We have also investigated the sensitivity of our solution to the value of β. Fig. 7 describes the 95^{th} percentile cost using the different load balance techniques. Similar results regarding response time are omitted for brevity. Evidently, β values have minimal effect on performance, as, in real-life, latencies, response times, and monetary costs are highly correlated.

We repeat these experiments with two additional applications, deployed alongside BookInfo and share services with it (namely, their workload competes with BookInfo on these shared services). This corresponds to the latest reports that, in

[4]See our simulation code at [26].



(a) $\beta = 0$, optimizing response time

(b) $\beta = 1$, optimizing cost.

Figure 6: A CDF presenting the response time in ms (in Fig. 6(a)) and cost in USD (in Fig. 6(b)) of a full path request over all the clusters in the system, from the user perspective for specific β value. MCOSS-QP is defined in (4), MCOSS-LP is defined in (3), RR denotes the round-robin load balancer, POWER-OF-TWO is defined in [35], and GREEDY stands for local optimization of MCOSS-LP.

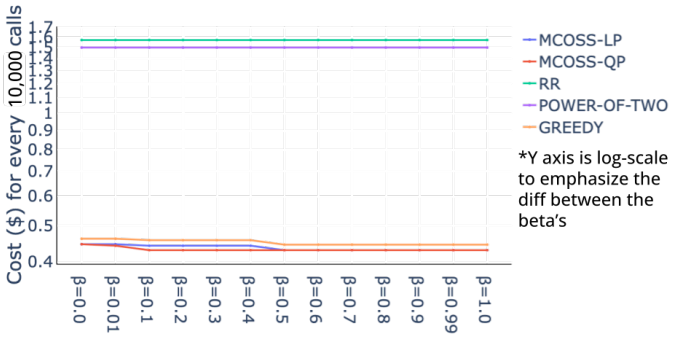


Figure 7: The 95th percentile of cost (in USD), as a function of β (log scale).

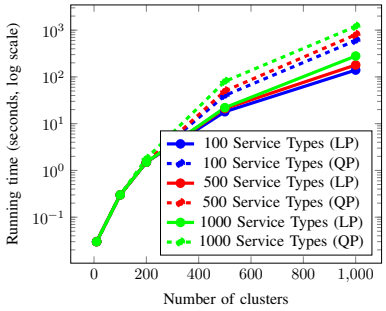


Figure 8: Solver runtime as a function of the clusters and service types (log scale).

the Alibaba clusters, about 5% of the services are used by 90% of the applications [20]. We have deployed the three applications in five different global clusters (North America, South America, Europe, and East Asia). Under this deployment, the mean monetary cost (with $\beta = 1$) has been improved by 72.08% and the response time has been improved by 61.53%, compared to a round-robin load balancer. In addition, both the linear and quadratic programs yield almost the same results and the value of β has minimal effects on performance.

C. *MCOSS is performant and adaptive*

We have tested our optimization problem solver under an increasing number of clusters and an increasing number of service types. Our results show that the number of service types almost does not affect the convergence time. On the other hand, when the number of clusters increases, the convergence time also increases. Our results in Fig. 8 show that up to 200 clusters, which cover all setups we have encountered so far, the converge time difference of (3) and of (4) is negligible.Note that MCOSS uses weighted round robin which is highly efficient, resulting in high routing performance with no further latency compared to the previous round robin mechanism [36]. Moreover, the data collection, sharing, and weight distribution are negligible in the size and amount of requests compared to the average data size and the number of user requests that are running within the system. See Fig. 9. Finally, recall that solving and constructing the minimization problem is done within the control-plane and does not interfere with the computing power needed to operate the data plane.

Finally, an interesting experiment can show how the greedy approach is prone to herd behavior and attacks on the system. We use the same Bookinfo application as in the previous experiment, but with a different layout. The two clusters in the US have a small capacity, and the one in Tokyo has a high capacity for different services. This caused the greedy and round-robin approaches to suffer from herd behavior and server starvation, and send more (less) traffic that can be handled by the services, while MCOSS, with its global view on the loads, distributes the load according to the capacity and avoids herd behavior and under-utilization.

V. Conclusion

In this work, we have investigated the service selection problem in a multi-cluster deployment of micro-service-based applications. Our solution consists of a data and control planes. In the data plane, we select services hierarchically: first, the destination cluster is selected and then the pod within the cluster is selected (by a component within the destination cluster). At the heart of the control plane solution is an optimization problem that strives to minimize a generic cost function. Solutions of this optimization problem are distributed

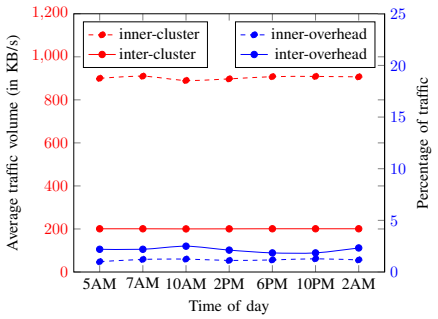


Figure 9: Data transfer in the system throughout a day per second

back to the data plane as *weights*, so that load balancing can be done using a *weighted round robin* (in our case, using DNS).

This work can be extended in many ways. First, we assume that the placement of services to a cluster is given. However, similarly to our dealing with scaling-out scenarios, one may decide to deploy services in additional clusters to reduce cost. It will be interesting to explore whether solving the placement and load balancing problems together may improve the performance. Second, in this work, we assume that the demands are known (or can be accurately estimated) to the optimizer. A promising future research direction is to explore how sensitive the optimizer is to errors in these estimations and/or unpredictable fluctuations in both demands and service instance capacities (e.g., as a result of failures). In addition, we plan to extend our model to deal, on one hand, with auto-scaling (implying, in a sense, that capacity constraints can be relaxed with a certain penalty in the cost function), and on the other hand, with situations in which requests are dropped.

One last suggestion for improvement can tackle the scale issue. As deployments are moving towards thousands of clusters on edge environments, we would like to enable linear degradation in the performance of the algorithm, as shown in Fig. 8 the degradation in performance is not linear and when we pass 1000 clusters it could take minutes for the algorithm to converge. Lastly, one can better utilize the autoscale mechanism, where the price delta between different metrics (including scale-related metrics such as CPU, memory, etc.) allows better control over the scale and optimizes its related cost.

Acknowledgements: The work was partly supported by Red Hat. We would like to thank Ilya Kolchinsky, Mike Kolesnik, Idan Levi, Louisa Nachshon, Tom Pantelis, Livnat Peer, Vishal Thapar, and Nir Yechiel of Red Hat for the useful discussions on MCOSS in general, and the interplay between MCOSS and Submariner in particular.

REFERENCES

[1] "What is a kubernetes service." [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/
[2] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city internet of things platform with microservice architecture," in *IEEE FiCloud'15*, 2015, pp. 25–30.
[3] M. Villari *et al.*, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, 2016.
[4] D. Lu *et al.*, "A secure microservice framework for IoT," in *IEEE SOSE'17*, 2017, pp. 9–18.
[5] "The kubernetes authors. 2017. kubernetes — production-grade container orchestration." [Online]. Available: https://kubernetes.io/
[6] "Submariner project official websitey." [Online]. Available: https://submariner.io/
[7] "Cilium load balancing." [Online]. Available: https://github.com/cilium/cilium/blob/master/Documentation/cmdref/cilium-agent.md
[8] "What is istio." [Online]. Available: https://istio.io/docs/concepts/what-is-istio/
[9] "Linkerd overview." [Online]. Available: https://linkerd.io/2/overview/
[10] "Cloud native computing foundation landscape." [Online]. Available: https://landscape.cncf.io/
[11] I. Stoica and S. Shenker, "From cloud computing to sky computing," in *HotOS '21*, 2021, p. 26–32.
[12] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *USENIX NSDI'16*, 2016, pp. 523–535.
[13] "Submariner broker." [Online]. Available: https://submariner.io/getting-started/architecture/broker
[14] D. Bhamare *et al.*, "Multi-objective scheduling of micro-services for optimal service function chains," in *IEEE ICC'17*, 2017.
[15] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 2956–2983, 2018.
[16] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *IEEE INFOCOM'18*, 2018, pp. 198–206.
[17] F. Wan, X. Wu, and Q. Zhang, "Chain-oriented load balancing in microservice system," in *IEEE WCCCT'20*, 2020, pp. 10–14.
[18] R. Yu *et al.*, "Load balancing for interdependent iot microservices," in *IEEE INFOCOM'19*, 2019, pp. 298–306.
[19] Z. Sun, "Latency-aware optimization of the existing service mesh in edge computing environment," 2019.
[20] S. Luo *et al.*, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *ACM SoCC'21*, 2021, pp. 412–426.
[21] C. Ayimba, P. Casari, and V. Mancuso, "Sqlr: Short-term memory q-learning for elastic provisioning," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 2, pp. 1850–1869, 2021.
[22] D. Bachar, A. Bremler-Barr, and D. Hay, "The MCOSS POC code." [Online]. Available: https://github.com/danibachar/kube-multi-cluster-managment
[23] "Grobi optimization (python framework)." [Online]. Available: https://www.gurobi.com/
[24] "Pulp optimization (python framework)." [Online]. Available: https://coin-or.github.io/pulp/
[25] "Alibaba cloud treace repository." [Online]. Available: https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021
[26] D. Bachar, A. Bremler-Barr, and D. Hay, "MCOSS simulator and graph generator." [Online]. Available: https://github.com/danibachar/Kube-Load-Balancing
[27] S. Ashok, P. B. Godfrey, and R. Mittal, "Leveraging service meshes as a new network layer," in *ACM HotNets '21*, 2021, p. 229–236.
[28] "Google cloud platform VPC pricing estimator." [Online]. Available: https://cloud.google.com/vpc/network-pricing
[29] "Cloud platform network tier dependant pricing estimator." [Online]. Available: https://cloud.google.com/network-tiers/pricing
[30] "AWS VPC pricing estimator." [Online]. Available: https://aws.amazon.com/vpc/pricing/
[31] "AWS API gateway pricing estimator." [Online]. Available: https://aws.amazon.com/api-gateway/pricing/
[32] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *ACM SIGCOMM'15*, 2015, pp. 139–152.
[33] K. Takahashi *et al.*, "A portable load balancer for kubernetes cluster," in *HPC Asia'18*, 2018, p. 222–231.
[34] A. C. Beltrão, B. B. N. de França, and G. H. Travassos, "Performance evaluation of kubernetes as deployment platform for iot devices," in *Ibero-American Conference on Software Engineering*, 2020.
[35] "Beyond round robin: Load balancing for latency." [Online]. Available: https://linkerd.io/2016/03/16/beyond-round-robin-load-balancing-for-latency/
[36] W. Wang and G. Casale, "Evaluating weighted round robin load balancing for cloud web services," in *IEEE SYNASC'14*, 2014, pp. 393–400.