This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3104662

Storage-saving scheduling policies for clusters running containers

Ludovico Funari, Luca Petrucci, Andrea Detti
Electronic Engineering Dept.
University of Rome Tor Vergata, Italy
Email: {ludovico.funari, luca.petrucci, andrea.detti}@uniroma2.it

Abstract—Container technology plays an important role in the virtualization landscape today. A container uses its own file system consisting of a stack of layers, which are stored on the execution server's disk. Containers running on the same server share the layers they have in common, and this sharing results in valuable savings in server storage space. Many containers can run on a single server, but when their resource demands grow enough, they are distributed across a cluster of nodes/servers by orchestration systems, such as Kubernetes. In this work, we found that for the same amount of containers to run, the storage required is higher for clusters consisting of a larger number of nodes. The severity of this storage overhead depends on the scheduling policy used to select the nodes that run the containers. By comparing different storage-saving scheduling policies that differ from each other in the depth of storage knowledge they leverage to make decisions, our analysis reveals that only deep, layer-level knowledge can effectively counter the growth in storage demand as the cluster size increases. Policies with coarser-grained knowledge achieve limited benefit because they achieve performance that is nearly equal to that of a random, zero-knowledge policy.

Index Terms—Containers, scheduling, storage

1 Introduction

Many modern applications are based on microservice architectures. Netflix, Amazon, Twitter, and other companies have indeed evolved in this direction. A microservice architecture can be defined as an architectural style that consists of breaking up an application into a set of small and lightweight services, each of them owning a focused and cohesive set of functionalities [1]. In particular, an application can be crumbled into tens or even hundreds of services that talk to each other by using HTTP REST, gRPC, or other kinds of network interfaces.

A widespread solution for packaging such services as self-contained units are Containers, isolated run-time environments whose file system encapsulates the software of the service and all its dependencies. Container technologies became, as a matter of fact, enablers of microservices architectures and, nowadays, one of the most common and easy-to-use engine for running, developing, and shipping containers is Docker [2]. However, there are other container engines available (e.g. Cri-O, Podman, ContainerD, etc.) that, along with Docker, follow the standard recommendations provided by the Container Initiative (OCI) [3] to promote interoperability.

A typical workflow followed by a developer to run a containerized service consists of three steps: assemble the base file system of the container, called *image* of the container; upload the image to a central repository, e.g. Docker Hub; run a new container based on such image on a selected server. During this last step, the server's container engine retrieves the image from the repository (if it is not available locally), creates an isolated run-time environment whose file system is that of the image, and, finally, within this environment, the engine runs the service packaged in the container.

The execution of containerized microservices applications is usually supported by a cluster of nodes, which are physical or virtual servers. When the application load grows, an advantage of microservice design is the ability to scale up the resources of only the impacted services and to perform this scaling horizontally, via replication. When a container running a service is overloaded, it is possible to run other replica containers in parallel and expose them all as a single entity by a load balancing function, which evenly distributes service requests to the containers in the *replica-set*. In this way, the cluster resources used by a service can be scaled horizontally by increasing or decreasing the number of containers running the service. For some applications, such as those related to Serverless Computing or Function as a Service (e.g., Azure Functions, AWS lambda, etc.), the number of containers running a service can drop to zero after quite long periods of no load, for instance, a few minutes; on the one hand this extreme scaling strategy is considered the most efficient way to adapt the use of resources to the actual demand, on the other hand the first request that arrives when no container is active suffers from a delay called "cold start" in which the container is started from scratch [4]. The cold start problem can also occur in the case of mobile edge computing applications when the user moves between edge data centers and containers of his services that require low latency are allocated on-demand on the cloud resources closest to him.

On which node to run a container is the result of a scheduling policy, whose first step is to filter out nodes that do not have enough resources or that do not meet user constraints such as *anti-affinity*, whereby containers of the same replica-set run on different nodes to increase service reliability. After the filtering step, the scheduling

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3104662

policy ranks the remaining nodes according to some scoring criteria that push the cluster towards the desired status; for instance, balanced resource consumption. Replication and scheduling are just two of the many orchestration tasks that are handled these days by indispensable high-level platforms like Kubernetes [5] or Docker Swarm [6].

Unlike a virtual machine whose operating system runs a lot of processes, a container runs only the packaged service. Therefore, there is no virtualization overhead in terms of memory and CPU consumption. A service running in a container has the same footprint as its uncontainerized version running directly in the node. As a result, many containers can run in a single node before exhausting its CPU and memory.

In contrast to their thrifty use of CPU and memory, the container technology is not very efficient in terms of storage consumption because each container carries with it an entire file system to be stored on the node's disk. To reduce the storage pressure (and more), containers use union mount file systems, such as overlayFS or AUFS, for which pieces of file systems, named *layers*, are combined and presented as a single file system to the applications running in the container. The layers are stored on the disk of the node, and running containers can *share* those layers that they have in common, thus saving disk space. In addition, union mount file systems also reduce the startup time of a new container because the container engine only needs to download the missing layers of its image from Docker Hub and decompress them, in fact each layer is stored and transmitted by Docker Hub as compressed archival files to save storage space and network bandwidth [7].

Although sharing layers brings important savings when containers are run on a single node, we argue that distributing these containers across multiple nodes risks hampering this benefit, thereby leading the entire cluster to consume more storage space. Specifically, for a given number of running containers, the overall storage space used by them increases as the number of nodes in the cluster scales out. Fig. 1 shows a hands-on proof of this observation. We deployed 50 containers on a Kubernetes (k8s) cluster consisting of a variable number of nodes. For each container, its image has been randomly selected from a pool of 10807 Docker Hub images. For the selection, we used an empirical popularity distribution based on Docker Hub statistics. In the figure, the first bar in each group is the total amount of disk space used by the containers in the cluster. The other bars are measures of the disk space used in each individual node. We note that as we increase the number of nodes in the cluster, the storage space used at each node decreases, but less than linearly, therefore the total disk space usage increases.

This storage inefficiency at cluster scaling out occurs because containers that have common layers may be distributed across different nodes in the cluster, thus preventing the ability to share these layers. For example, imagine we have two containers $C1$ and $C2$. The file system of $C1$ consists of an ordered set of two stacked layers, called $L_{u(1)}$ (upper) and L_l (lower). The file system of $C2$ consists of the layers $L_{u(2)}$ and L_l. We note that the lower layers L_l are equal. In a cluster consisting of one node, their joint usage of the node's disk consists of three layers: $L_{u(1)}$, $L_{u(2)}$ and L_l,

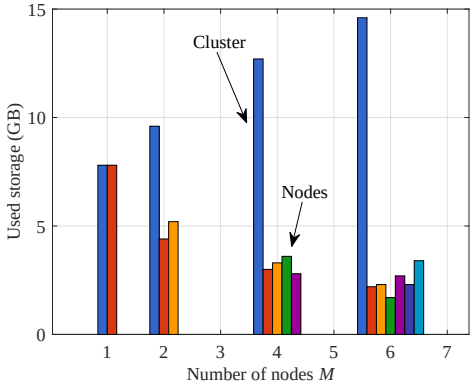

Fig. 1: Used storage space of a Kubernetes cluster in case of $N = 50$ containers. First bar or each group is the total storage space used in the cluster, other bars are the per-node usage

where the latter is shared. In a cluster with two nodes, the scheduling policy might decide to run the containers $C1$ and $C2$ on two different nodes, e.g., $N1$ and $N2$ respectively. In this case, their joint usage of the storage space is larger because composed of four layers rather than three. Node $N1$ stores on its disk the layers needed by $C1$, namely $L_{u(1)}$ and L_l. Node $N2$ stores the layers needed by $C2$, i.e. $L_{u(2)}$ and L_l. The layer L_l is no more shared.

While less efficient from a storage perspective, having a cluster with many small nodes rather than few giant ones is undeniably useful in other respects, such as fault tolerance, and is sometimes even necessary due to the unavailability or excessive cost of very powerful servers. This has motivated our interest in analyzing solutions that reduce the inefficiency of storage in large clusters. With this regard, we notice that the previous example highlighted two aspects of the scaling issue. First, increasing the number of nodes introduces the risk of using more storage space. Second, this risk actually occurs depending on the decisions made by the scheduling policy. Accordingly, in this paper, we propose and compare storage-saving scheduling policies, reducing disk occupancy at the cluster level.

The considered policies differ in the degree of knowledge they have about the storage status of the cluster. Our exploration ranges from a zero-knowledge policy for which a new container is deployed on a node chosen at random, to a policy with a deep knowledge, at the layer-level, whereby a new container is run on a storage-optimal node. The analysis also evaluates the impact of replication and anti-affinity constraints, which are features widely used for the deployment of microservice applications.

All in all, the contribution of this paper is threefold: i) we devised an analytical model for evaluating cluster-level storage occupancy for containerized applications; ii) the model revealed a storage problem not addressed so far, i.e., storage utilization grows as cluster size increases; iii) once we understood the nature of the problem, we analyzed and proposed several storage-saving scheduling policies, and one of them, called Layer Locality, turned out to be very promising.

Regarding the impact of our research, we note that, undeniably, storage is one of the cheapest resources in a server today compared to CPU and memory, so the use

of storage-saving scheduling policies may raise practical concerns. However, for large real/virtual infrastructure with hundreds of servers, reducing the size of the batch of disks to be installed can result in valuable capital expenditure savings. Moreover, the resulting benefits of a storage-saving policy are also about a reduction in the time needed to start a container, which has an impact on the flexibility of a system to effectively adapt to dynamic loads with rapid scale-ups, while also avoiding long cold start times [8]. In fact, if a scheduling policy selects a node where a container already finds all or part of its layers, the container engine does not have to either download the missing layers or decompress them, which reduces the container startup time.

The paper is organized as follows. In Sec. 2, we present an analytical proof that storage inefficiency occurs when the scheduling policy doesn't use any storage-related information and chooses nodes at random. In Sec. 3, we describe the considered scheduling policies whose performance is evaluated and discussed through simulations and real measurements in Sec. 4. Finally, we revised related works in Sec. 5 and draw conclusions in Sec. 6.

2 Analytical understanding

In this section, we present an analytical model that helps to understand how increasing the number of cluster nodes impacts storage space utilization. We consider the simplest case of a so-called Random scheduling policy that chooses the node where to deploy a new container at random, using a uniform distribution. To help the comprehension of the model, we first describe how a container engine builds the file system of a container. Then we present the analytical model and related considerations, which are based on a performance evaluation that uses a set of images gathered from Docker Hub (Tab. 1).

This set of images contains 10807 of the most pulled images, as of April 2020. Overall, the number of layers that compose the images is 77249 layers, a layer on average is contained in 1.446 images (layer reuse factor). For each image, we saved: the image ID, the pull count [1], the sizes and the identifiers (SHA256) of the relative layers it is composed [2] . From this information, we computed all required model inputs.

Total number of images	10807
Total number of layers	77248
Sum of image sizes	2778 GB
Average number of layers per image	10.31
Average image size $\overline{S_{img}}$	257 MB
Average layer reuse	1.446

Table 1: Docker Hub image set

1. The pull count value is the number of times an image has been pulled and, after normalization, it is the popularity of the image $P_{img}(k)$.

2. We considered the *compressed* size of the layers, which is the only information available from Docker Hub. Therefore, the actual storage occupation may be higher than what we calculated in our analytical analysis and simulations. However, we are interested in understanding phenomena and comparing scheduling policies, and for these purposes the use of compressed sizes is not relevant

Background

As shown in Fig. 2, a container image is built up from a stack of layers. The figure shows two images, I_1 and I_2, each made of three layers which are identified by a digest that is calculated by applying the SHA256 algorithm to a layer's content. The two bottom layers (L_1, L_2) of the images are the same, whereas the next layers (L_3, L_4) are different. The figure also shows how these layers are stored and combined to run 4 containers on a host. Containers 1 and 2 are based on the image I_1; containers 3 and 4 on the image I_2. The files of layers L_1 and L_2 are shared among the four containers. The files of the layers L_3 and L_4 are shared only among containers based on the same image. Moreover, for each container, a different (thin) upper layer (L_u) is added. Every write operation made by applications running in the container will be saved in the upper layer, the other lower layers are read-only.

N	Number of containers running in the cluster
M	Number of nodes of the cluster
K	Number of images of the image pool
H	Number of layers that form the images of the image pool
S_c	Average storage utilization of the cluster
S_n	Average storage utilization per-node
$S_{nc}(n)$	Average storage utilization per-node when it executes n containers
I_k	Image kth
$\overline{S_{img}}$	Average image size
$P_{img}(k)$	Popularity of the image I_k
L_h	Layer hth
$S_{lyr}(h)$	Size of the layer L_h
$P_{lyr}(h)$	Probability that the layer L_h is included in an image chosen using the image popularity distribution P_{img}
$P_{lxn}(h,n)$	Probability that the layer L_h is stored on a node when it runs n containers
$P_{lxn}(h)$	Probability that the layer L_h is stored on a node

Table 2: Summary of Notation

Analytical model

We assume that N containers are randomly distributed in a cluster of M nodes (Random scheduling policy). The containers use an image pool that includes K images. These images are formed by a unique set of layers consisting of H layers. A container uses the image I_k ($1 \leq k \leq K$) with a probability $P_{img}(k)$, which is the *popularity* of the image. Fig. 3 shows the P_{img} rank-plot of the Docker Hub images [3].

Each node of the cluster runs a subset of the N containers, which consumes S_n bytes of the node storage space. The average value S_c of the storage space used by the N containers overall the cluster can be written as M times S_n. Indeed, containers are randomly distributed on nodes, and, therefore, nodes are statistically equivalent.

$$S_c = M\, S_n \tag{1}$$

The value S_n is equal to the following weighted average, where the parameter $S_{nc}(n)$ is the average amount of bytes used on a node when it executes n containers,

3. We see that the image popularity distribution P_{img} resembles a Zipf, unless in the first part of the ranking where it is flatter.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3104662

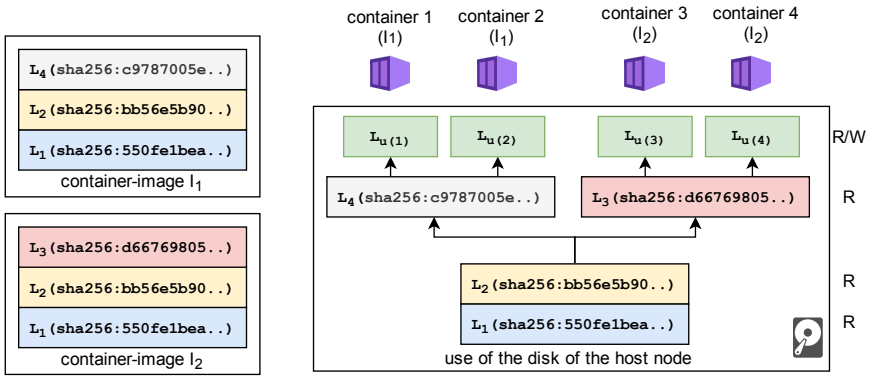

Fig. 2: Disk footprint of docker containers

and $b(n, N, 1/M)$ is the binomial probability that the node executes n containers out of the N running in the whole cluster.

$$\begin{aligned} S_n &= \sum_{n=1}^{N} b(n, N, 1/M)\ S_{nc}(n) \\ b(n, N, 1/M) &= \binom{N}{n} \left(\frac{1}{M}\right)^n \left(1 - \frac{1}{M}\right)^{N-n} \end{aligned} \tag{2}$$

A layer L_h is stored on a node if used by at least one of the containers run by the node. Consequently, the average amount of bytes stored on a node $S_{nc}(n)$ when it executes n containers can be written as the sum of the sizes $S_{lyr}(h)$ $(1 \leq h \leq H)$ of the layers, weighted by the probabilities $P_{lxn}(h, n)$ that layers are used in at least one container out of n [4]; i.e:

$$\begin{aligned} S_{nc}(n) &= \sum_{h=1}^{H} P_{lxn}(h, n)\ S_{lyr}(h) \\ P_{lxn}(h, n) &= (1 - (1 - P_{lyr}(h))^n) \end{aligned} \tag{3}$$

$P_{lyr}(h)$ is the popularity of the layer L_h, that is the probability that the layer is included in a container (and hence in an image), whose image is chosen using the image popularity distribution P_{img}. It can be written as:

$$P_{lyr}(h) = \sum_{k=1}^{K} P_{img}(k)\ \mathbb{1}_{(L_h \in I_k)} \tag{4}$$

where $\mathbb{1}_{(L_h \in I_k)}$ is the indicator function equal to 1 if the layer L_h belongs to those in the image I_k. Fig. 4 shows the P_{lyr} rank plot of Docker Hub images [5].

Using Eq. 2, Eq. 3 and Eq. 4, the average value of the storage space used in a node S_n can be written and simplified as follows, where $q(h) = 1 - P_{lyr}(h)$:

$$\begin{aligned} S_n &= \sum_{n=1}^{N} b(n, N, 1/M) \sum_{h=1}^{H} (1 - q(h)^n)\ S_{lyr}(h) \\ &= \sum_{h=1}^{H} S_{lyr}(h) \left(\sum_{n=1}^{N} b(n, N, 1/M)(1 - q(h)^n) \right) \\ &\text{change of inner sum initial value from } n = 1 \text{ to } n = 0 \\ &\text{since } 1 - q(h)^n = 0 \text{ for } n = 0 \\ &= \sum_{h=1}^{H} S_{lyr}(h) \left(\sum_{n=0}^{N} b(n, N, 1/M)(1 - q(h)^n) \right) \\ &= \sum_{h=1}^{H} S_{lyr}(h) \left(1 - \sum_{n=0}^{N} \binom{N}{n} \left(\frac{q(h)}{M}\right)^n \left(1 - \frac{1}{M}\right)^{N-n} \right) \\ &\text{reverse binomial expansion} \\ &= \sum_{h=1}^{H} S_{lyr}(h) \left(1 - \left(\frac{q(h)}{M} + (1 - \frac{1}{M}) \right)^N \right) \\ &= \sum_{h=1}^{H} S_{lyr}(h) \left(1 - \left(1 - \frac{P_{lyr}(h)}{M} \right)^N \right) \end{aligned} \tag{5}$$

Finally, we have:

$$S_n = \sum_{h=1}^{H} S_{lyr}(h)\ P_{lxn}(h) \tag{6}$$

$$P_{lxn}(h) = 1 - \left(1 - \frac{P_{lyr}(h)}{M} \right)^N \tag{7}$$

The Eq. 6 can be better understood observing that $P_{lxn}(h)$ is the probability that the layer L_h is stored on a node, i.e., the probability that at least one container deployed on the node uses that layer [6]. Therefore, the sum

4. We are not considering the thin upper layers of containers because they are expected to be much smaller than images for most services (excluding large databases, etc.). Since these upper layers are not shared, their impact on the overall storage has no relation to the number of nodes in the cluster and, if necessary, can be taken into account by adding an application-dependent constant per container.

5. We notice that the sum of $P_{lyr}(h)$ values is greater than one and is equal to the average number of layers per image.

6. In fact, the probability that a container run by the user has the layer L_h and it is deployed by the scheduler on a considered node is equal to: $P_{lyr}(h)\ 1/M$. Consequently, the layer L_h is not stored on the node if no container out of the N having the layer L_h is deployed on the node, i.e., $(1 - P_{lyr}(h)/M)^N$. It follows that the probability $P_{lxn}(h)$ that at least one container deployed on the node uses the layer L_h is the complementary probability of $(1 - P_{lyr}(h)/M)^N$ in Eq. 7.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3104662

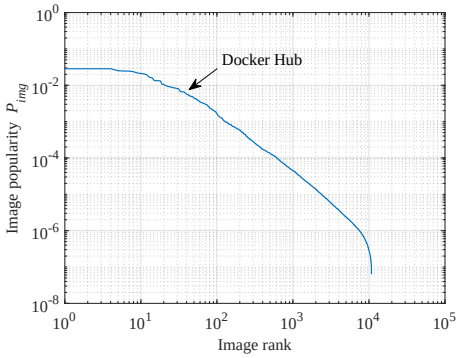

Fig. 3: Image popularity $P_{img}(k)$ vs image rank

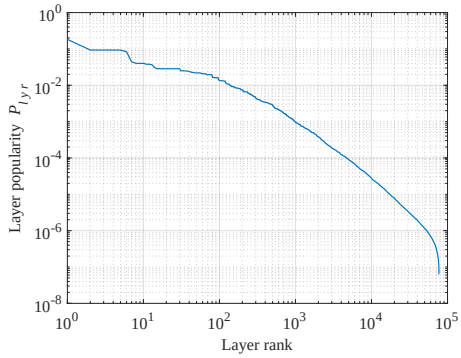

Fig. 4: Layer popularity $P_{lyr}(h)$ vs layer rank

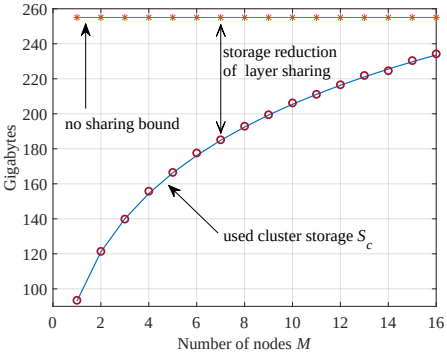

Fig. 5: Used cluster storage space by $N = 3000$ containers randomly distributed over the nodes of the cluster vs cluster size M

of $P_{lxn}(h)$ for $1 \leq h \leq H$ weighted by the layers' size $S_{lyr}(h)$ (Eq. 6) is the average amount of bytes S_n stored on the node. This value multiplied by the number M of nodes of the cluster is equal to S_c.

$$S_c = M \sum_{h=1}^{H} S_{lyr}(h)\, P_{lxn}(h) \tag{8}$$

Fig. 5 shows the storage space utilization of the cluster as the number of nodes varies, in the case of 3000 containers. The plot includes analytical results from Eq. 8 (line without markers) and results obtained from a simulator we developed (circle markers), which confirm model validity. The storage utilization tends to increase as the cluster size increases (Q.E.D.). This means that if we need to deploy a new cluster to run a given amount of containers, it is better to install a few large nodes than many small ones, if our goal is limited to saving storage space[7].

Let us now discuss some asymptotic limits. When the cluster size tends to infinity, Eq. 8 reaches the following horizontal asymptote:

$$\lim_{M\to\infty} S_c = N \sum_{h=1}^{H} S_{lyr}(h) P_{lyr}(h) = N\overline{S_{img}} \tag{9}$$

To understand this result conceptually, we observe that, in the absence of layer sharing, N containers use on average an amount of disk space equal to N times the mean image size $\overline{S_{img}}$. This value is exactly the result of the limit that, therefore, can be considered as a *no-sharing bound* towards which the cluster tends as it grows, while the number of containers is kept constant. In fact, as the number of nodes increases, containers tend to run alone within their nodes thus preventing any layer sharing.

Fig. 5 includes the no-sharing bound, and this allows us to make two considerations. First, the approach of the curve to the bound means that increasing the number of nodes tends asymptotically to completely cancel the benefit of layer sharing, bringing system performance toward the performance without sharing. Second, the difference between the no-sharing bound and the value of S_c can be seen as a measure of the effectiveness of the choice made by container technology developers of using union mount file systems and layer sharing to save storage resources.

So far we have explored the pros and cons of varying the cluster size when we need to run an expected number of containers. Let us now consider a more dynamic situation for which the number of containers does not remain constant, but increases over the lifetime of the cluster, and, concurrently, the cluster infrastructure is scaled proportionally to support the increasing load. For example, suppose that the infrastructure rule we use is to install one node per η (e.g. 16 [9]) containers. For this evolutionary context, an interesting asymptotic behavior concerns the cluster storage utilization as the number of containers and nodes increase, and their ratio $\eta = N/M$ is kept constant. As reported in Eqs.10 and 11, this performance can be evaluated using Eqs. 6, 7, and 8, first considering the asymptotic value of the storage utilization per node (S_n) and then using this value to compute the cluster-level storage utilization (S_c).

$$\lim_{\substack{N\to\infty \\ M=N/\eta}} S_n = \sum_{h=1}^{H} S_{lyr}(h) \left(1 - e^{\eta\, P_{lyr(h)}}\right) \tag{10}$$

$$S_c \approx M \sum_{h=1}^{H} S_{lyr}(h) \left(1 - e^{\eta\, P_{lyr(h)}}\right) \quad \text{for } N >> 1 \text{ and } M = N/\eta \tag{11}$$

Eq. 11 shows that if we increase the number of containers and the cluster size together, the storage utilization linearly grows because the sum is independent of M.

We close this section noting that the conclusions drawn so far refer to a Random scheduling policy, which does not care about storage. Policies that reward storage-saving solutions, like those we present in next Sec. 3, can greatly reduce the scale-out storage penalty, thus allowing us to take full advantage of all the other benefits (resiliency, flexibility, etc.) of clusters with many nodes.

3 Scheduling Policies

A scheduling policy is an algorithm whose execution is triggered by a user request to deploy a single container or a group of R replica containers, called replica-set. The case of a single container is equivalent to the case of a replica-set made of a single item; therefore, we consider the replica-set as the general input request of a scheduling policy.

7. We note that, as mentioned in the introduction, the utilization of the storage space of the single node decreases at the increase of cluster size, however, the decrease rate is slower than $1/M$ and consequently the whole storage utilization of the cluster increases.

The output of the policy is a result-set D composed of R nodes, whose rth node is the one chosen to run the rth replica container. The user may request an anti-affinity constraint that, whether possible, forces the scheduler to select different nodes for the different replica containers. Without the anti-affinity constraint, the scheduler decides independently for each container in the replica-set and, thus, replica containers can also be deployed on the same node.

Algorithm 1: Generic scheduling algorithm with anti-affinity

```
R = size of the replica-set
C = {c_1...c_K} # set of constraint functions
S = {s_1...s_W} # set of scoring functions
P = {1...M} # initial pool of schedulable nodes

# filtering phase
foreach m ∈ P do
    foreach c_j ∈ C do
        if c_j(m) == false then
            # node m doen't respect constraint c_j
            remove m from P
            break
        end
    end
end

# scoring phase
foreach m ∈ P do
    foreach s_j ∈ S do
        # increase the score of node m by using s_j
        m.score+ = s_j(m)
    end
end

# selection phase
Ps = pool P sorted on node score. First node has the highest
  score
D = {} # scheduled nodes
for r = 0 to R − 1 do
    k = mod(r, |Ps|) + 1 # node in Ps to be used for replica r
    add kth node of Ps to D
end
return D
```

The algorithm 1 is a generic logic that can be used to implement different scheduling policies with anti-affinity constraint. Different policies use different constraint C and scoring S functions. The scheduling decision is made in three phases, called *filtering*, *scoring*, and *selection*. In the filtering phase, nodes that do not meet the constraints are eliminated from the pool P of schedulable nodes. In the scoring phase, each node in the pool receives a score from each scoring function. In the selection phase, nodes are sorted according to their score and this sorted list forms the set P_s. Next, the first R nodes of P_s are inserted into the result-set D. If the number of schedulable nodes $|P_s|$ is greater than R, the result-set consists of different nodes and the anti-affinity constraint is satisfied. Conversely, if $|P_s|$ is less than R, the anti-affinity constraint cannot be satisfied. Nevertheless, we consider it more important to deploy all replicas, therefore, we tolerate nodes being reused in these cases. This is the effect of the modulo operator.

This algorithm can also be used in the absence of the anti-affinity constraint by independently executing it for each replica, and passing as input to each run a replica-set consisting of only one container.

In the following, we present five scheduling policies aimed at reducing the storage consumption of the cluster. These policies leverage different storage status knowledge, ranging from zero knowledge to layer-level knowledge, for which the scheduler has a view of what layers are stored on the nodes. Comparing these policies allows us to understand how useful it is to have a deep knowledge of the storage status to gain a real advantage in disk savings. The related algorithms can be implemented on the base of algorithm 1 by defining specific constraint (C) and scoring (S) functions.

We note that the scheduling policy of a real system focuses not only on storage optimization but also on other aspects of the system such as balanced CPU utilization, fair distribution of containers on nodes, etc., which can be considered in the algorithm 1 through appropriate constraints and scoring functions. However, from this paper, we want to derive guidelines for the design of storage-targeted constraints and scoring functions and, therefore, we do not consider other aspects in depth.

3.1 Random

This policy has zero knowledge of the storage status of the cluster. When a request of deploying a new replica-set arrives, the scheduler chooses nodes at random, without repetition. It can be implemented by algorithm 1 by considering a void set of constraints C and the single scoring function

$$s_1(m) = \texttt{rand} \tag{12}$$

that returns a uniform random value in the interval $(0, 1)$.

3.2 Least Used Disk

This algorithm leverages a disk-level knowledge of the storage status of the cluster whereby favors nodes that have the least disk space utilization. It can be implemented by algorithm 1 by considering a void set of constraints C and a single scoring function

$$s_1(m) = -S_n(m) \tag{13}$$

where $S_n(m)$ is the amount of bytes of the disk used by containers running on node m.

3.3 Image Locality

This algorithm exploits an image-level knowledge of the storage status through which it tries to deploy new containers on nodes that already have containers based on the same image [8]. The related algorithm has no constraints C and has a single scoring function

$$s_1(m) = \mathbb{1}_{(I_k \in \text{ node } m)} - \epsilon\, \rho(m) \tag{14}$$

$$\rho(m) = \frac{S_n(m)}{\sum_{j \in P} S_n(j)} \tag{15}$$

where I_k is the image of the container to deploy; $\mathbb{1}$ function returns 1 if the image is already available on node m, zero otherwise; $\rho(m)$ is the disk space utilization of the

8. A similar policy is included in Kubernetes, the other three ones are instead introduced by us.

node m normalized to the overall storage occupation of the schedulable nodes (P) and ϵ is a number much smaller than one (e.g. 0.001). The value $-\epsilon\rho(m)$ is used to break the tie, by penalizing nodes with higher storage occupancy.

3.4 Layer Locality

Of all the algorithms, the Layer Locality one requires the most in-depth knowledge of the status of the storage. The scheduler needs to know which layers are present on the nodes and which layers form the I_k image of the container to be deployed. Combining this information, the scheduler computes the increase $\delta(I_k, m)$ in storage occupancy that would occur if the new container were deployed on node m. For each layer of I_k, if the layer is not present on node m, the value $\delta(I_k, m)$ is increased by the layer size; otherwise, if the layer is present, the value $\delta(I_k, m)$ is not increased. After this computation, the scheduler favors the nodes with the lowest storage increase. Likely, the chosen nodes are those running containers that have a high number of common layers with the new container; indeed, common layers do not increase storage utilization, and this leads to lower values of $\delta(I_k, m)$.

In the long term, such priority management leads to a kind of "preferential attachment" process for which the more containers a node has, the more likely it is chosen by the scheduler. In fact, a node with many containers is more likely to have layers in common with the new container and thus to have a lower value of $\delta(I_k, m)$. As the number of containers grows, the preferential attachment process leads to an increasingly unbalanced use of cluster resources. To re-establish a proper balance, we introduced in the algorithm the following fairness constraint: a node m can be chosen only when the normalized disk usage $\rho(m)$ is less than the perfect fair share value $1/M$, multiplied by a tolerance factor $\gamma > 1$. The greater γ, the greater the possible storage unfairness.

Overall, the Layer Locality policy can be implemented by using the following constraint and scoring functions,

$$c_1(m) : \rho(m) \leq \frac{\gamma}{M} \tag{16}$$

$$\begin{aligned} s_1(m) &= -\delta(I_k, m) - \epsilon\rho(m) \\ \delta(I_k, m) &= \sum_{h|L_h \in I_k} S_{lyr}(h)\mathbb{1}_{(L_h \notin \text{ node } m)} \end{aligned} \tag{17}$$

Also in this case the value $-\epsilon\rho(m)$ is used to break tie conditions by penalizing nodes with higher storage occupancy.

3.5 Multi-Constraint Layer Locality

The Layer Locality policy considers a single constraint (c_1) that aims at a fair use of the nodes' storage space. To analyze possible performance losses in multi-constraint environments, we considered another policy called Multi-Constraint Layer Locality, which takes into account another resource constraint, in addition to the storage fairness one. We assumed that a container with base image I_k, running on node m, impacts its resources both in terms of storage by an amount equal to $\delta(I_k, m)$, and in terms of another *generic* resource (e.g., CPU, memory, etc.) by an amount equal to $Gr_{img}(I_k)$. The amount $Gr_n(m)$ of the generic resource consumed on the mth node is equal to the sum of the $Gr_{img}(k)$ values of the containers run by the node.

Eq. 18 shows the two constraints c_1 and c_2 of the Multi-Constraint Layer Locality policy. The first constraint c_1 enforces a fair consumption of the generic resource among nodes. The set of nodes that are not filtered by c_1 are subsequently subject to the second constraint c_2, which aims at storage fairness, as in the case of the plain Layer Locality policy.

$$\begin{aligned} c_1(m) &: \sigma(m) \leq \frac{\gamma}{M} \\ c_2(m) &: \rho(m) \leq \frac{\gamma}{M} \\ \sigma(m) &= \frac{Gr_n(m)}{\sum_{j \in P} Gr_n(j)} \end{aligned} \tag{18}$$

The value of $Gr_{img}(k)$ obviously depends on what the resource we are considering actually is and what is the service run by the image I_k. In the next performance evaluation section, we considered this value as a random number, between 1 and 500, for the only purpose of evaluating a possible performance degradation of the Layer Locality policy in the presence of an additional, generic, resource constraint. The specific range is not as relevant because the constraint c_1 aims at fairness.

4 Performance Evaluation

We compared the five policies using a simulator to understand the extent to which a system can benefit from increasing knowledge of its storage situation. In fact, Random, Least Used Disk, Image Locality, and Layer Locality policies have an increasing fine-grained knowledge of storage: zero, disk-level, image-level, and layer-level knowledge, respectively.

The simulation workload provides the sequential deployment of replica-sets, the image of each of them is chosen following two possible strategies, called *vanilla* and *hybrid*:

- vanilla: for this strategy, the image of a replica-set is one from the Docker Hub image set (Tab. 1), chosen at random using the image popularity in Fig. 3 or a synthetic Zipf distribution. Therefore, the cluster exclusively executes containers based on unchanged Docker images.
- hybrid: this strategy aims to simulate configurations in which the cluster runs containers that use both unchanged Docker Hub images and custom images based on them. We believe that this configuration better captures real-world situations for which the cluster is used to run microservices applications. In fact, in a microservices application, only a few containers use unchanged Docker Hub images, e.g. to run generic services such as databases or HTTP proxies. On the other hand, for the many remaining containers, the developer uses Docker Hub images (e.g. those including programming runtimes as Python or Node.js) to create new custom images containing the code of specific services to be executed. The hybrid strategy simulates these configurations as follows: the base image used by a replica-set is initially extracted from the Docker Hub image

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3104662

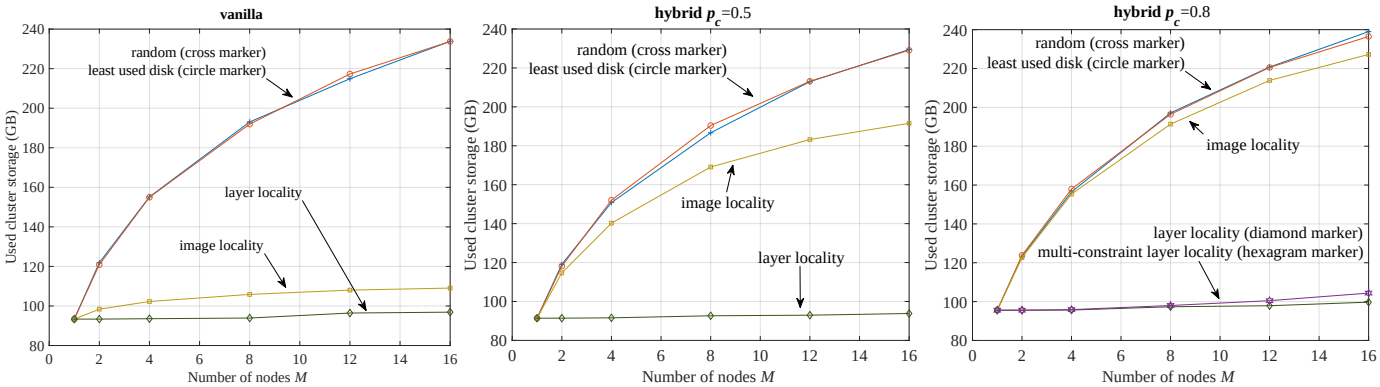

Fig. 6: Used cluster storage space for 3000 containers versus cluster size, vanilla and hybrid image selection

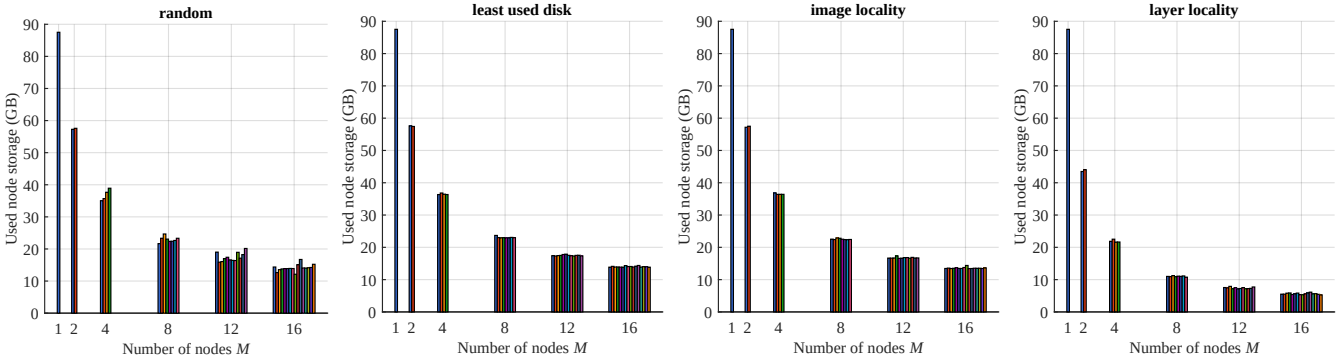

Fig. 7: Used node storage space for 3000 containers versus cluster size, hybrid image selection with 80% of custom containers

set following the image popularity in Fig. 3 or a synthetic Zipf distribution. Then, with probability p_c, this base image is used to create a custom image, which has a new unique identifier; otherwise, with probability $1-p_c$, the base image is used directly. For simplicity, we do not consider the additional storage space resulting from the new layers added to a base image to create a custom image. Therefore, the size of a custom image is equal to the size of its base image. The two images differ only in their identifiers [9]. We note that if $p_c = 0$, vanilla and hybrid strategies are equal.

In the following, we call replica-set consisting of a single container ($R = 1$) simply "container", and use the term replica-set only when it contains more than one replica container ($R > 1$). We also use Docker Hub related image popularity, unless otherwise noted.

Fig. 6 shows the cluster storage space used by 3000 containers. The left plot refers to a vanilla configuration, the other two plots refer to a hybrid configuration with an increasing number of custom containers, from 50% (center) to 80% (right) of the whole amount of containers. Random and Least Used Disk policies practically have the same performance. Therefore, the knowledge of disk usage doesn't help in limiting storage inefficiency at the increase of the number of nodes. Indeed, the Least Used Disk policy tends to equalize the use of storage space but this goal, in the long term, is reached also by the zero-knowledge Random policy.

The performance of the Image Locality policy is strongly dependent on the amount of custom containers in the cluster. For no custom containers (vanilla case), the policy is quite effective in limiting the storage footprint when scaling out the cluster. This happens because the popularity of images is rather skew (Fig. 3) and, as a result, many containers use the same image. The policy tends to cluster these homogeneous containers on the same nodes, thus promoting layer reuse and saving storage space. However, this saving virtue fades out for hybrid configurations, and when the cluster runs 80% of custom containers the performance gets closer to that of the Random policy [10]. We conclude that knowing which images are running on the nodes does not help in the case of (realistic) microservice applications with many custom containers. In contrast, the Layer Locality policy nearly eliminates the scale-out problem in any configuration. Recall that the optimal performance, i.e. the minimum storage footprint, is that for a "cluster" of a single node. With Layer Locality, this optimum value is practically maintained as the cluster size increases and, therefore, a fine-grain, layer-level knowledge of the storage status can really help save disk space.

For the case of 80% custom containers, Fig. 6 (right) also shows the performance obtained by the Multi-Constraint Layer Locality policy described in Sec. 3.5. The additional generic constraint (c_1) clearly reduces the ability to select

9. We made this simplification because it is difficult to make assumptions about the size of the new layers. In addition, these layers are custom and thus cannot be shared with any other container. Consequently, their impact on storage is the same for any policy, so neglecting them from the storage calculation does not alter the policy comparison we want to undertake.

10. For this reason, the analytical model of the Random policy of Sec. 2 can be considered as an approximation of what would happen in a real cluster, since the Image Locality policy is used in the current Kubernetes codebase.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3104662

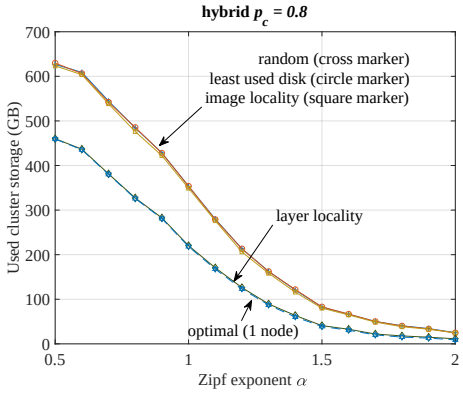

Fig. 8: Used cluster storage space for 3000 containers in a cluster of 16 nodes versus Zipf α parameter of image popularity, hybrid image selection

the best storage node, however, the performance reduction compared to the plain Layer Locality policy is very limited, suggesting that its integration into multi-constraint scheduling problems should not considerably reduce its effectiveness.

Fig. 7 shows the storage space used per node, for different schedulers, in the case of 80% of custom containers. All policies use node storage rather equally [11]. Except for Layer Locality, for the other schedulers, the storage occupancy per node decreases less than linearly as the number of nodes increases. This sub-linear reduction leads to an increase of the total cluster storage utilization, as shown in Fig. 6. In contrast, in the case of Layer Locality, we notice an almost linear reduction as the number of nodes increases, and because of this, the cluster storage occupancy remains almost insensitive to the cluster size.

The previous performance was obtained considering the popularity of the images related to Docker Hub pull frequency (Fig. 3). We also considered a model of image popularity that follows the Zipf distribution, as in [8], and conducted a sensitivity analysis with respect to the Zipf parameter α. Fig. 8 shows the result of this analysis for a cluster with 16 nodes, with 3000 containers of which 80% are custom. For low values of α, most containers use a different image, so the storage footprint is larger because layers reuse is limited. As α increases, more and more containers use the same image, layers reuse increases, and storage utilization consequently decreases. In any case, the Layer Locality policy provides the lowest storage utilization, close to the optimal value (dashed line). This is the only analysis where we use Zipf image popularity, in the following we continue to use Docker Hub image popularity.

Fig. 9 shows the storage utilization for a cluster with 16 nodes versus the number of containers. Again, the Layer Locality policy almost achieves the optimal performance, which is that achieved in the case of a single-node cluster, as also shown in the figure.

We now analyze what happens when we introduce replication with anti-affinity constraint. If there are enough nodes in the clusters, the containers of a replica-set are spread over different nodes; otherwise, some node is reused to complete the deployment (see algorithm 1). Fig. 10 shows the performance in case of 750 replica-sets, each made of four replica containers ($R = 4$). In particular, we can identify two distinct behaviors of the curve: before and after $M = R$.

As long as the cluster size is less than or equal to the number of replicas ($M \leq R$), the policies do not differ from each other and the storage utilization grows linearly. This happens because each policy first tries to guarantee the anti-affinity constraint. Therefore, the R containers of a replica-set are distributed in a round-robin fashion on each node to avoid, as much as possible, their presence on the same node. Since nodes are less than the R containers, each node executes at least one container, stores a copy of its image, which can also be used by the other containers of the replica-set that the node is possibly running. Consequently, each node stores the complete set of images/layers of containers executed in the cluster, and these images occupy $S_n(full)$ bytes of a node's disk. This behavior holds for $1 \leq M \leq R$ and, in this interval, the memory occupancy of the cluster S_c grows linearly as $M\ S_n(full)$. After the initial interval, the scheduler recovers the ability to make different decisions about where to deploy containers, and as a result, differences between the policy capabilities arise. Once again, the Layer Locality policy achieves the optimal performance for any cluster size. In fact, when the linear growth due to the anti-affinity constraint ends, the policy is able to hold the storage footprint at that minimum value.

So far, we have varied the number of nodes in the cluster, or the number of containers, separately. In a final evaluation, we keep the ratio η between the number of containers and nodes constant as discussed in Sec. 2 and consider greater clusters up to 200 nodes [9]. This analysis allows us to evaluate policies in an evolving cluster infrastructure that is scaled out to follow container demand [12]. Fig. 11 shows that when the number of containers and nodes are increased together ($\eta = 16$), Random, Least Used Disk and Image Locality, perform almost linearly, as foreseen by Eq. 11 The Layer Locality policy behaves better because by increasing the number of nodes, it has more possibilities to efficiently distribute containers while respecting the fairness constraint. And it effectively exploits these possibilities to reduce the linear growth of storage footprint. For a cluster of 200 nodes, the Layer Locality policy provides a storage footprint reduction of 2.5 and 2.35 compared to the Random and Image Locality policies, respectively. Similar results have been obtained for $\eta = 200$ and $\eta = 110$, where the latter value is the maximum number of containers (Pods) per node on standard Kubernetes clusters. For these greater values of η, the improvement of the Layer Locality policy with respect to Image Locality is greater than 2.35, e.g., about 2.6 for $\eta = 110$ and 32 nodes. In fact, with more containers per node, the Layer Locality policy makes better use of its ability to reuse layers.

We conclude the performance evaluation by reporting some results we obtained using a real Kubernetes (v1.21.2) cluster running on virtual machines in an Azure data center. The cluster runs 300 containers using vanilla Docker Hub

11. For Layer Locality we used the fairness factor $\gamma = 1.5$

12. We are also assuming that as the cluster infrastructure increases, running containers are redistributed across nodes. This condition can actually happen, albeit in the long term, due to the deployment of new versions of services/containers, as is the case with Kubernetes rolling updates.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3104662

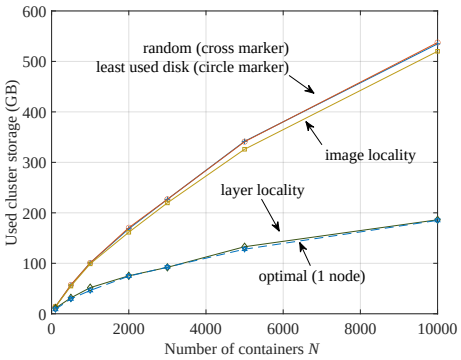

Fig. 9: Used storage in a cluster of 16 nodes vs the number of containers, hybrid selection with 80% of custom containers

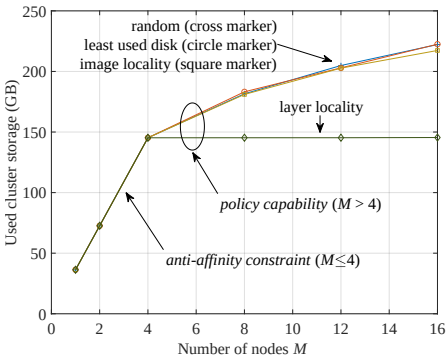

Fig. 10: Used cluster storage for 750 replica-sets with four containers versus cluster size, hybrid selection with 80% of custom containers

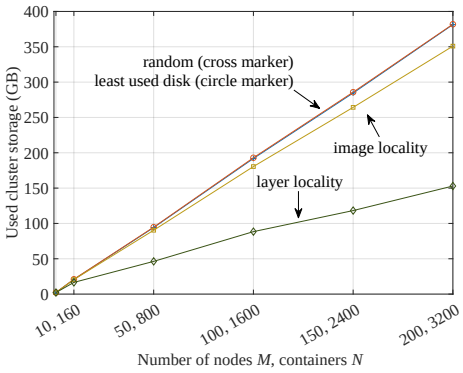

Fig. 11: Used cluster storage versus number of containers/nodes for $\eta = N/M = 16$, hybrid selection with 80% of custom containers

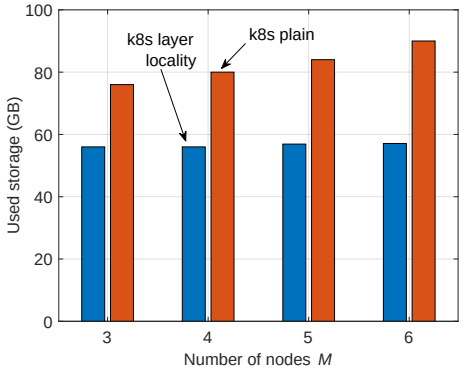

Fig. 12: Used storage space of a Kubernetes (k8s) cluster for 300 vanilla containers

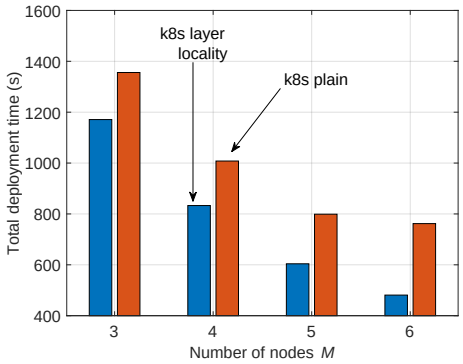

Fig. 13: Total deployment delay of 300 vanilla containers in a Kubernetes (k8s) cluster

images. We requested Kubernetes to deploy the 300 containers all at once. Fig. 12 shows the storage occupancy of the entire cluster both using the plain Kubernetes scheduler and the Layer Locality policy [13]. This real experiment confirms that the Layer Locality policy is insensitive to the number of nodes in the cluster (as in Fig. 6), while the cluster storage utilization increases in the case of the plain Kubernetes scheduler that is unaware of layers, even though it implements Image Locality as a scoring function [14].

Fig. 13 shows the time required to deploy the full set of 300 containers. This graph supports that a reduced storage footprint is only one side of the benefit that a layer-level knowledge of the storage can bring. The other aspect, which may be even more important as discussed in the introduction, relates to the reduction in the time required to start a container. With the Layer Locality policy, a container finds all or part of its layers on the node where it is being executed. As a result, the container engine does not have to either download missing layers or decompress them, and this reduces the container startup time compared to that with a layer unaware scheduling policy, such as the one currently used by Kubernetes.

In any case, increasing the number of nodes reduces the deployment time because the network and CPU capacities offered by the nodes are used in parallel to download and decompress the layers, and finally execute the containers. However, the reduction rate is lower in the case of the plain Kubernetes policy because there is the contrasting effect of increasing the storage space (Fig. 12), i.e., the number of bytes that must be downloaded from the network and decompressed to run containers increases at cluster scale out. This contrasting effect is not there for the Layer Locality policy, which therefore takes full advantage of having more nodes in the cluster and, in fact, the deployment time decreases faster with respect to the plain Kubernetes.

5 Related Work

The growing adoption of the microservices architectures [10] has led Docker, and containers in general, to be a pivotal component of modern enterprise applications [11]. Several aspects of performance have been addressed by the research, and of these, some studies have focused more on issues related to container images.

The papers [12] [13] [14] [15] analysed and made proposals for improving performance (speed) of container file systems or registry architecture [16]. Other works have provided solutions to speed up the distribution of images

13. We used the simulator to compute the execution nodes for the Layer Locality policy. Then we applied these choices in the real cluster using Kubernetes `nodeSelector` constraint.

14. The performance achieved is a sort of best case for the plain Kubernetes scheduler, because its Image Locality scoring function can take advantage of the fact that we are using containers based on vanilla images.

from a registry to execution nodes, which is the most time-consuming step in the process of deploying a container [17] [18] [19], [20] [21]. A common idea behind these solutions is to create a shared local registry by using different approaches (DHT, network file systems, etc.).

Some works, like ours, have been interested in how to optimize the storage footprint, motivated by the fact that Docker and container-based virtualization technologies, in general, are storage hungry. In [7], the authors observed that the current layer sharing approach still leads to significant file duplication between different layers. This finding paves the way for research of new sharing approaches that target file-level deduplication, which can greatly reduce the amount of storage space used, and would be particularly useful for large-scale registries such as Docker Hub. In this direction, [22] inspected how the process of building container images can be modified to optimize storage via an overall rearrangement of the layer structure itself.

Within the arena of scheduling-related papers, in [8], the authors explored the use of scheduling policies that take into account available images and layers on nodes to reduce the storage footprint and startup latency of containers in a Kubernetes cluster. After a test campaign using containers based on vanilla Docker Hub images and a real implementation of their policies in Kubernetes, they concluded that scheduling policies that leverage layer-level or image-level knowledge provide a valuable improvement in startup latency and storage footprint compared to a storage agnostic policy (our random one). However, the benefit provided by a, deeper, layer-level knowledge over an image-level one was not found to be as high in their study (1.6x); in fact, only a scheduling policy based on image-level knowledge was eventually adopted in the main Kubernetes codebase. To some extent, our work can be considered as a follow-up to [8]. Unlike their study, our work revealed that when running containers are not based on vanilla Docker Hub images but on custom images (which we believe is very realistic), the benefit provided by a scheduler with image-level knowledge vanishes dramatically, while having layer-level knowledge still significantly improves the storage footprint (e.g., 2.35x) and, consequently, container startup latency. In this sense, our work revamps the value of policies that take into account layer locality. Besides this conclusion, in our work, there is a broader discussion of the underlying phenomena (considering also the effect of replication) and sensitivity analysis with respect to clusters and workload parameters. Compared to the layer-matching policy proposed in [8], our Layer Locality policy takes into account a storage fairness constraint, and we have also shown that the policy provides good performance even when considering additional generic constraints, which was a question mark in [8]. Finally, at the best of our knowledge, our work is the first one proposing an analytical model of storage utilization in a cluster running containers, which highlights the problem of storage usage growth as the cluster size increases.

Many other papers related to the scheduling issue (e.g., [23] [24] [25] [26] [27] [28]) formulate a multi-objective scheduling problem by focusing on heterogeneous optimization aspects that include storage, such as the papers [29], [30], and [9]. For these solutions, the storage requirement is an input parameter of the request, a fixed amount of bytes. Our work suggests that, for container-based environments, it is better not to consider the storage demand as constant because, due to layer sharing, the real storage demand depends on the containers that are already running on the cluster nodes. By taking this layered nature of container images into account, the scheduler is able to make much better storage-saving decisions. Therefore, future works could include revising these multi-objective scheduling solutions to change the storage demand from constant to that proposed in Eq. 17.

6 Conclusions

Within a single server, the layering of images allows them to be reused by many containers, resulting in a significant reduction in storage (disk) utilization. However, the storage-saving tends to fade out in a cluster, where containers are distributed across many nodes in a storage-agnostic way. In fact, for a given number of containers, their cumulative storage footprint becomes higher when distributed in larger clusters.

This storage inefficiency when the cluster scales out can be mitigated by scheduling policies that reward storage-saving deployments. To compute such rewards, a scheduling policy must necessarily have a level of knowledge about the storage status of the cluster. In our analysis, a layer-level knowledge results to be the only one that a scheduler can effectively leverage to overcome the storage inefficiency. The scheduler must be aware of which image layers are available on each node and which layers form the image of the containers it is going to deploy. Such fine-grained knowledge allows the implementation of a Layer Locality policy that identifies the best nodes where to run containers to minimize their impact on storage and also, as a result, make them faster to start up. The resulting storage footprint is (nearly) equal to the minimum one for any cluster size and microservice configuration.

A coarse-grained knowledge, for example, about what the storage utilization of each node is, or what images are available on each node has not been very useful, especially with custom containers. In fact, the corresponding scheduling policies, Least Used Disks, and Image Locality, respectively, achieved performance almost close to that of a zero-knowledge Random policy.

Clearly, a fine-grained knowledge such as that at the layer-level requires more effort in scheduling and signaling, but we believe this to be definitively feasible with current technologies, and indeed a similar Kubernetes implementation has been already proposed in [8]. We also note that, a layer-level scheduler must query Docker Hub to find out which layers make up an image that it sees for the first time. This information is then cached for later reuse. The query delay may be a few hundreds of ms [8], but such delay penalty is only experienced once, and thus not during the frequent scale out operations that can occur in the lifetime of a microservice.

Acknowledgment

This work is supported in part by the Italian MIUR PRIN Liquid_Edge project.

REFERENCES

[1] C. Richardson, *Microservices Patterns*. Manning Publications, 2019.

[2] Docker. [Online]. Available: https://www.docker.com/

[3] Open container initiative. [Online]. Available: https://opencontainers.org/

[4] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[5] Kubernetes (k8s): Production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[6] Docker. Docker swarm. [Online]. Available: https://docs.docker.com/engine/swarm/

[7] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-scale analysis of the docker hub dataset," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–10.

[8] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[9] G. Fan, L. Chen, H. Yu, and W. Qi, "Multi-objective optimization of container-based microservice scheduling in edge computing," *Computer Science and Information Systems*, no. 00, pp. 41–41, 2020.

[10] 2019 container adoption survey. [Online]. Available: https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf

[11] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[12] V. Tarasov, L. Rupprecht, D. Skourtis, W. Li, R. Rangaswami, and M. Zhao, "Evaluating docker storage performance: from workloads to graph drivers," *Cluster Computing*, vol. 22, no. 4, pp. 1159–1172, 2019.

[13] Q. Xu, M. Awasthi, K. T. Malladi, J. Bhimani, J. Yang, and M. Annavaram, "Performance analysis of containerized applications on local and remote storage," in *Proc. of MSST*, vol. 3, 2017, pp. 24–28.

[14] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding performance of i/o intensive containerized applications for nvme ssds," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016, pp. 1–8.

[15] F. Zhao, K. Xu, and R. Shain, "Improving copy-on-write performance in container storage drivers," in *Storage Developers Conference*, 2016.

[16] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig *et al.*, "Improving docker registry design based on production workload analysis," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, 2018, pp. 265–278.

[17] L. Du, T. Wo, R. Yang, and C. Hu, "Cider: a rapid docker container deployment system through sharing network storage," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2017, pp. 332–339.

[18] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "Comicon: A co-operative management system for docker container images," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 116–126.

[19] N. Hardi, J. Blomer, G. Ganis, and R. Popescu, "Making containers lazy with docker and cernvm-fs," in *Journal of Physics: Conference Series*, vol. 1085, no. 3. IOP Publishing, 2018, p. 032019.

[20] C. Zhengt, L. Rupprecht, V. Tarasov, M. Mohamed, D. Skourtis, A. S. Warke, D. Hildebrand, and D. Thaint, "Wharf: Sharing docker images across hosts from a distributed filesystem," 2016.

[21] W. Kangjin, Y. Yong, L. Ying, L. Hanmei, and M. Lin, "Fid: A faster image distribution system for docker platform," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2017, pp. 191–198.

[22] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving perfect layers out of docker images," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[23] P. Townend, S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu, "Invited paper: Improving data center efficiency through holistic scheduling in kubernetes," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 156–15 610.

[24] R. Duan, R. Prodan, and X. Li, "Multi-objective game theoretic schedulingof bag-of-tasks workflows on hybrid clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 1, pp. 29–42, 2014.

[25] D. Zhang, B. Yan, Z. Feng, C. Zhang, and Y. Wang, "Container oriented job scheduling using linear programming model," in *2017 3rd International Conference on Information Management (ICIM)*, 2017, pp. 174–180.

[26] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, vol. 16, no. 1, pp. 113–135, 2018.

[27] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *Journal of Network and Computer Applications*, vol. 119, pp. 97–109, 2018.

[28] Z. Wei-guo, M. Xi-lin, and Z. Jin-zhong, "Research on kubernetes' resource scheduling scheme," in *Proceedings of the 8th International Conference on Communication and Network Security*, 2018, pp. 144–148.

[29] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud," *IEEE Access*, vol. 7, pp. 83 088–83 100, 2019.

[30] R. Duan, R. Prodan, and X. Li, "Multi-objective game theoretic scheduling of bag-of-tasks workflows on hybrid clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 1, pp. 29–42, 2014.

Ludovico Funari is a researcher at the University of Rome Tor Vergata. He received the master's degree in "ICT And Internet Engineering" in October 2019. His research activity includes IoT, Cloud and Edge computing. He has worked as a CNIT (Italian National Inter-University Consortium for Telecommunications) researcher for the EU H2020 "Fed4IoT" project. He is currently working for the "Liquid_Edge" MIUR research project at University of Rome Tor Vergata.

Luca Petrucci received the Master's Degree in Computer Science Engineering in April 2019, from the University of Rome Tor Vergata. From April 2016 to December 2019 he is a researcher at CNIT (Italian National Inter-University Consortium for Telecommunications), where he had developed his bachelor thesis and master thesis, respectively concerning the EU projects BEBA and 5G-PICTURE. He is currently working for the "Liquid_Edge" MIUR research project at University of Rome Tor Vergata.

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