2017 IEEE International Conference on Cloud Engineering

CoMICon: A Co-operative Management System for Docker Container Images

Senthil Nathan* , Rahul Ghosh† , Tridib Mukherjee† , Krishnaprasad Narayanan†
* IBM India Research Lab † Conduent Labs India (formerly Xerox Research Center India (XRCI)).
snatara7@in.ibm.com, {rahul.ghosh, tridib.mukherjee, krishnaprasad.narayanan}@xerox.com

***Abstract*—Docker containers are becoming an attractive implementation choice for next-generation microservices-based applications. When provisioning such an application, container (microservice) instances need to be created from individual container images. Starting a container on a node, where images are locally available, is fast but it may not guarantee the quality of service due to insufficient resources. When a collection of nodes are available, one can select a node with sufficient resources. However, if the selected node does not have the required image, downloading the image from a different registry increases the provisioning time.**

Motivated by these observations, in this paper, we present *CoMICon*, a system for co-operative management of Docker images among a set of nodes. The key features of *CoMICon* are: (1) it enables a co-operative registry among a set of nodes, (2) it can store or delete images *partially* in the form of layers, (3) it facilitates the transfer of image layers between registries, and (4) it enables distributed pull of an image while starting a container. Using these features, we describe—(i) high availability management of images and (ii) provisioning management of distributed microservices based applications. We extensively evaluate the performance of *CoMICon* using 142 real, publicly available images from Docker hub. In contrast to state-of-the-art full image based approach, *CoMICon* can increase the number of highly available images up to 3× while reducing the application provisioning time by 28% on average.

I. Introduction

Operating system (OS) containers [1], [2] are becoming increasingly popular among the cloud and DevOps community with emerging open source container management technologies (e.g., Docker [3], CoreOS [4]). Major cloud providers (e.g., Google [5], IBM [6], Microsoft [7], Amazon [8]) have recently announced container based cloud services to cater for this popularity. Each week more than 2 billion container instances [9] are spawned in Google data centers to package and run services like Google Search and Gmail. Such containers are becoming critical for internal private cloud operations as they facilitate fast development, testing, and delivery of enterprise class cloud services. Another key reason behind the popularity of containers is that they enable the vision of microservices style of software development [10]–[12] where software applications are designed as suites of independently deployable services.

In contrast to the approach in monolithic software, microservices are loosely coupled where each of them has a definite purpose organized around business functions. The execution of a microservice as a container automatically enables the goals of independent deployment and management. Thus, the container is not only the option to enable microservices style of software development but also a natural and attractive choice for the deployment and execution. Consider the scenario when a microservice-based application is implemented using Docker containers [13] and the application consists of a number of containers. When provisioning such an application, Docker container instances need to be created from individual container images.

*This work was done when the author was a PhD student doing his internship at Xerox Research Centre India (XRCI), Bangalore.

There are three approaches to provision a Docker container instance from a Docker image registry: (a) from a local registry, i.e., the container instance and the image registry are on the same host OS, (b) from Docker public image registry [14], and (c) from a private image registry (e.g., [15]). While the local image registry reduces the container provisioning time, the desired image may not be available on the local registry. A locally unavailable image may be found in Docker public registry but downloading an image over wide area network can significantly increase the provisioning time. Maintaining an image registry on a separate host, within a private network may reduce the network latency. A recent proposal has further used lazy loading of images from the local registry [16] to reduce provisioning time. However, none of these solutions have considered High Availability (HA) of images—a key requirement to ensure that there is no or minimal downtime while provisioning containers (and corresponding microservices) under registry failures. A straight forward way to achieve HA can be by hosting the registries on distributed file storage system such as Amazon S3 [17], Azure Blob Storage [18], HDFS [19], etc. This approach however increases the operational cost.

The focus of this paper is to reduce the provisioning time while considering the HA requirements for container images at no additional storage cost. In this regard, we propose a new *co-operative* image management approach for Docker container based microservices. The main idea is to create a pool of local Docker hosts that can easily share the container images among each other. Note that, the usefulness of sharing images among the Docker hosts have already been recognized within the Docker community [20]. While there are *adhoc* and isolated attempts to solve this problem, this paper presents a comprehensive solution that facilitates such co-operative image management. Specifically, the three major contributions of this paper are as follows:

1) We **introduce a co-operative registry for Docker** that enables modular container image sharing across hosts. This is enabled by *storing and pulling an image partially at the granularity of only the desired layers within an image*. Instead of the entire image, selective storage of only the desired layers reduces the disk space on the host as well as the network traffic.
2) We present *CoMICon*, a **novel system** that performs **management of Docker container images** on the **co-operative** registry. Specifically, *CoMICon* supports dynamic image distribution among Docker hosts to ensure HA and provisioning of container instances from the co-operative registry. *CoMICon* allows any Linux based Docker host to register and subsequently manages the container images of a microservice based application across hosts.
3) *CoMICon* is extensively evaluated for both HA and provisioning. Using a private cloud setup in our lab, we compare the performance of *CoMICon* w.r.t. state-of-the-art image registry. Our results, using real world images from the Docker hub, show that *CoMICon* **reduces the provisioning time** by 28%. Further, compared to a full image replication, **the number of highly available images is increased** up to 3×.

The rest of the paper is organized as follows. Section II presents the related work followed by the background and motivation in Section III. Section IV introduces the co-operative registry for Docker. Section V then describes *CoMICon*. Section VI presents the evaluation of *CoMICon*, and finally, Section VII concludes the paper.

II. Related Work

High availability (HA) of virtual machine (VM) images. In [21] and [22], Cidon *et al.* present *Tiered Replication* and *Copyset Replication* respectively, to minimize the probability of data loss in face of failures. Muralidhar et al. [23] describe a system *f4* to increase the efficiency of replication by examining the underlying access patterns of large storage objects. Nadgowda et al. [24] propose a disaster recovery solution *I2Map* that maintains a mapping between VM instances and the master image from which it was created. In [25], Wu et al. present a runtime model-based configuration framework for an HA mechanism in the cloud. In [26], the authors present an image library *Mirage* which allow to persist images in a format that indexes the filesystem structure. Applying the traditional HA algorithm(s) is not suitable in *CoMICon* for the following three reasons: (1) image and its layer sizes are different for different images; (2) enabling multiple replications leads to duplicate replication of the common layers, and (3) it is difficult to find different nodes for the entire image to fit in. We overcome these drawbacks by introducing our own HA aware image distribution in a co-operative registry, which allows both image and layer level replication for a given set of images and their corresponding layers.

VM image management for fast provisioning. Optimization of VM image management for provisioning is a well studied area of research [27], [28]. Nicolae *et al.* [29] minimize the provisioning time via collaboration of unrelated VM instances. VMThunder [30] enables on-demand data download during the provisioning process. A similar lazy deployment scheme is presented in [31]. In [32], [33] and [34], peer-to-peer (P2P) streaming is used to reduce the VM startup time. Many improvements in VM image management can also be applicable for container image management at the concept level. However since VM images are not modular, these approaches cannot enable partial management of images across peers.

Improvement in Container Provisioning Management. Harter *et al.* [16], present a system called *Slacker* that provides efficient deployment for Docker containers. Besides leveraging existing methods, such as lazy propagation of images, they introduce new Docker specific optimization techniques to speed up the provisioning time of containers. Hegde *et al.* [35] describe *SCoPe*, a provisioning decision management system for containers at large scale. However, *Slacker* and *SCoPe* does not leverage a co-operative registry. In addition, they do not take into consideration the HA for container images. One can host the registry in distributed storage services from Amazon S3, Azure blob storage, etc. *at an additional operational cost* to ensure HA along with *Slacker*.

Distributed pull of Docker images. In addition to the research studies, we highlight a few projects proposed by the developers of open source communities. Examples of P2P download of images include [36], [37] and [38]. These P2P systems however do not use the information about partial images (layers) while retrieving the image. *CoMICon is a **first-of-a-kind** system, which: (a) provides an ability to store, copy and retrieve images partially within a registry; and (b) jointly addresses the issues of reducing the container provisioning time and HA of container images, through a novel co-operative image registry across Docker hosts, without any additional storage cost.*

III. Background & Motivation

A. Image Management in Docker

In the paper, we use the term node to indicate the compute platform where a Docker container is provisioned and run. Examples of nodes include physical machines (PMs) or VMs. A node consists of a Docker engine and a Docker registry. A Docker container is instantiated from a container image using the `docker run` command. The images typically reside in a Docker registry and are copied from the registry to the Docker engine during instantiation. Physically, the Docker registry and the Docker engine can be on the same or different node(s). Container images consist of read-only layers each containing a set of directories and files. The layers of an image are stacked together where the top-most layer in the stack is a child of the second layer, which is a child of the third layer from the top, and so on. Most of the Docker storage drivers e.g., AUFS, Btrfs, provide the ability to manage images in a layered manner [39]. For the purpose of demonstration and evaluation of our proposal, we use AUFS as a representative storage driver in the rest of the paper.

The Docker registry stores the directories and files of each layer in an image as a compressed `tar` file. For each stored image, a `manifest` file is created which contains the following information for each layer of that image: (i) the `sha256` hash of layer's content, (ii) size of the layer, and (iii) information about the parent layer.

A Docker engine can download an image from the registry using `docker pull` command. This command takes the registry IP address and the required image name as inputs, and accesses the corresponding manifest file (using registry APIs). The image layers (i.e., the tar files) are then downloaded and stored in the file system. Notably, only the layers, which are not already present in this directory, are downloaded from the registry to the Docker engine. The layer contents are extracted (untarred) in the aforementioned directory. The docker storage driver performs union of directories and files in each layer to provide a file-system for that container.

B. Limitations of Current Image Management

The downloading of an image from the registry is a resource intensive task. Depending on where the registry is maintained, there can be major implications on the container provisioning time and network overhead. Figure 1 shows a conceptual diagram of state-of-the-art Docker registry. Here, $Container_1$ and $Container_2$ are provisioned on $node_2$ from $image_1$ and $image_2$, respectively. While the $image_1$ has three layers, the $image_2$ consists of two layers. There is one common or duplicate layer between these two images. All the layers of $image_1$ are downloaded first. Hence, for the $image_2$, only the unavailable layers are downloaded.

While Docker Hub [14] is a popular choice for downloading the images during provisioning, it incurs a high network delay. The cluster management tool such as Google Kubernetes [15] maintains a private registry at the local network to avoid long network delays. We evaluate the impact of common layers between images on the provisioning time, in our private cloud. We create 2 VMs (each with 4 CPUs @2.3 GHz, 8 GB of memory, and 3 Gbps network link between them)— (i) one VM hosts a private registry that contains the Epinion online transaction processing (OLTP) image [40] (with 12 layers and an aggregate size of 1.169 GB), and (ii) another VM acts as the Docker node. Table I presents the provisioning time with and without the presence of 4 (common) layers of Epinion already on the node. The presence of common layers on the node significantly reduces the provisioning time.

TABLE I
IMPACT OF COMMON LAYERS ON THE PROVISIONING TIME (SEC) WHEN EPINION IMAGE IS PULLED FROM A PRIVATE REGISTRY.

Test case	Average (sec)	Std. dev. (sec)
w/o common layers	28.5	0.47
w/ common layers	16.69	0.34

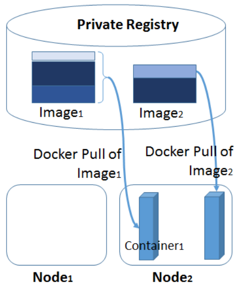


Fig. 1. Image pull from a private registry. Different layers of an image is color-coded.

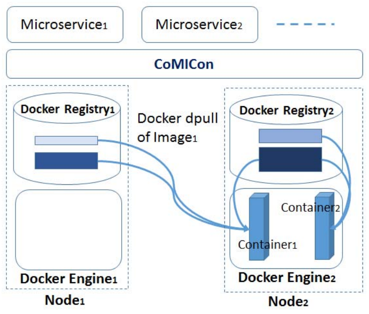


Fig. 2. Our proposal: image management across a co-operative registry.

Similar provisioning time is further observed when the registry is hosted on a distributed storage file system. We do not show the exact results to avoid repetition. In general, we label a conventional Docker registry as *non-cooperative registry* (irrespective of whether it is hosted in a centralized or a distributed storage).

In this paper, we introduce the concept of *co-operative registry*. Consider a scenario as shown in Figure 2, where node 1 and 2 have local Docker registries. By having a co-operative registry, the images from node 1 can be copied to node 2. However, copying all the images may not be possible because of limited storage capacity on a node. One option is to selectively store different images across nodes, such that an image is always locally available for provisioning. Still, it may be undesirable to provision a container locally on a node when the available resources are insufficient to satisfy the QoS. To demonstrate such a scenario, we start an Epinion OLTP application in a container running on a VM where a disk intensive application called Vdbench [41] (referred as noise) is also executing. As shown in Table II, the average provisioning time of Epinion OLTP container increases by more than 5× while its runtime performance, i.e., throughput reduces by more than 60%.

TABLE II
IMPACT OF RESOURCE CONSTRAINT WHEN A CONTAINER IS PROVISIONED FROM AN IMAGE AVAILABLE AT LOCAL REGISTRY.

Test case	Average provisioning time (sec)	Average throughput (request/sec)
w/o noise	3	3646
w/ noise	17.78	2219

Thus, if a container can not be provisioned on the node where the image is present, the image will be copied to another node for provisioning. This leads to the same issue of high provisioning time as with the non-cooperative registry. We therefore argue that it is important to have the ability to store and retrieve images partially in the form of layers to improve the provisioning time. Furthermore, when a set of nodes are already available within a cloud network, a co-operative image sharing mechanism among the nodes can significantly reduce

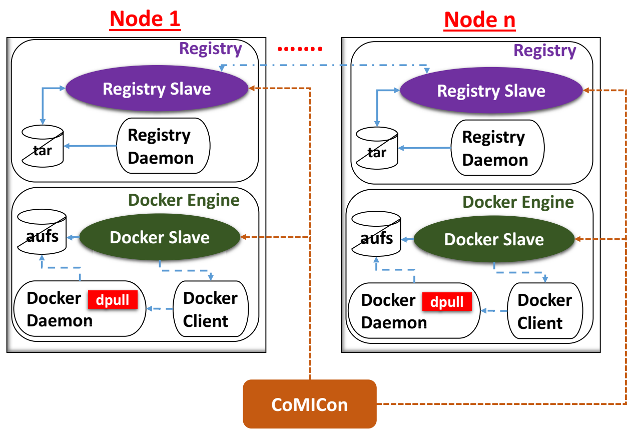


Fig. 3. Co-operative Docker Registry Architecture.

the provisioning time. Figure 2 conceptually shows this vision, where the `docker pull` command is modified to `docker dpull` (i.e., distributed pull). To manage the image sharing and distributed container provisioning across the nodes, a system, *CoMICon*, is needed that is placed between the nodes and a set of microservice based applications. In the next section, we describe how we enable image sharing across nodes by using co-operative registry.

IV. Co-operative Docker Registry

The proposed co-operative Docker registry is shown in Figure 3. Each registry can store different set of layers. We have modified the Docker source code and added a slave daemon at each registry called `registry slave` to perform the following three tasks which are necessary to dynamically redistribute images when needed: (i) store a partial image (i.e., not all layers) by appropriately creating a manifest file, (ii) copy layers between registries, and (iii) delete layers from a registry. Next, we present the implementation details for each of the above three tasks.

Store a Partial Image. The default `manifest` file of an image contains `sha256` hash value of all layers associated with that image (as a registry stores full image by default). When we store a partial image (i.e., only a few set of layers), we cannot use the default `manifest` file. Hence, we modified the `manifest` file structure to add a flag called `present` to denote whether a layer is stored in the current registry. By downloading an image's `manifest` file from a registry, we can identify the list of image's layers present in that registry.

Copy a Layer between Registries. In order to dynamically re-distribute layers among registries, we enable registry slaves to transfer / receive layers among them. Given a `sha256` hash of a layer, the source registry slave locates the layer and transfers it to the destination registry slave along with all the image names that use the layer at source. At destination, on receiving a layer and its associated image names, first, the registry slave stores the layer as per content addressable storage mechanism. Second, the registry slave tries to locate respective image's `manifest` file. If the manifest file is available, it sets the `present` flag for the received layer. Otherwise, it downloads the manifest file from source registry slave and resets `present` flags for all layers except the received one.

Delete Layers from a Registry. In order to either increase free storage capacity or to remove the unused images, we enable the deletion of layers. On receiving a `sha256` hash value of a layer, the registry slave deletes the layer and resets the `present` flag in all manifest files that use the corresponding layer. If no `present` flags in the manifest files are set, registry slave removes that manifest file.

Distributed Pull. We introduce a command called `docker dpull` in `docker daemon` to enable the download of an image from co-operative registries. The `docker dpull` takes a list of `<IP address of a registry, sha256 hash of an image's layer>` as input and downloads each layer from the corresponding registry and creates the file system for the container. Note that, we provide only unavailable layers of the required image as input to the `dpull` command. The `docker dpull` performs the following steps to download an image: (i) downloads layers from the respective Docker registry (using the given IP address), (ii) extracts the downloaded layer to perform union operations on directories and files provided that the parent of this layer is already extracted. The provisioning of containers using the new `docker dpull` command needs a co-operative management of images across the registries.

Key Benefits of Co-operative Management. With a co-operative Docker registry, it is possible to store an image partially (i.e., subset of layers of an image) in a registry. Hence for HA, the replication can be done at a layer level instead of a full image level. If there is little free space remaining in the image registries across multiple nodes, it may not be possible to find multiple nodes where a number of replicas of a full image can fit in. With the layer level modularity, such replication may still be feasible. Moreover, if there are common layers across different container images, a full image replication will lead to duplicate replication of the common layers. This may in turn reduce the available registry space and thus, making many images HA infeasible. Layer based modularity can avoid this duplication problem.

Note that, when a full image is replicated across registries and a node fails, the recovery of an image requires the transfer of the entire image. Layer level modularity can ensure the transfer of only the required layers that were present in the failed registry. This can significantly reduce the amount of data transfer and consequently reduce the recovery time. Additionally, depending on how the images and their layers are distributed across multiple registries, it is possible to provision an image locally (i.e., from the registry associated with the node) as much as possible. Furthermore, if the different layers of an image are distributed in different registries, it is possible to perform parallel transfer of the layers—thus further reducing the provisioning time.

Key Issues. The key issues to address in this regard are:

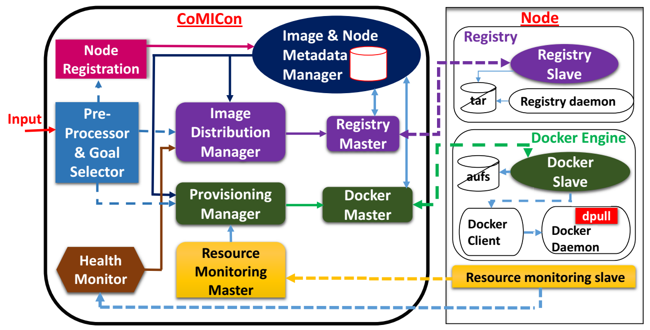


Fig. 4. CoMICon system.

(i) where to place the images (and their layers) across a set of registries with fixed registry sizes to ensure HA and fast provisioning? (ii) in the case of registry failures, which registries to select for recovering lost image layers so that recovery time can be reduced as much as possible? (iii) on which node (destination) a container is provisioned? and (iv) from which registries (sources) the image layers are fetched? We now discuss how these issues are addressed using *CoMICon*.

V. Description of *CoMICon*

The components of co-operative image management system (*CoMICon*) are shown in Figure 4. The **input** to *CoMiCon* is one of the following:

1) *Participating nodes*. Any new node can be added to the co-operative registry by providing the IP address along with the login credentials and the resource capacity as input.
2) *Images*. These images correspond to microservices and are to be distributed across nodes.
3) *Applications to provision*. The set of container (microservices) images and the respective resource requirements are provided as input for provisioning.

On receiving an input from the user, the **Pre-Processor and Goal Selector** scans the input format and identifies the type of the given request. Specifically, the first input type is forwarded to the **Node Registration**. The last two types of inputs are handled by: (i) **Image Distribution Manager (IDM)** and (ii) **Provisioning Manager (ProM)**. They are described in Sections V-A and V-B respectively.

The Node Registration adds information about the new node in a database called **Image and Node Metadata Manager**. The database stores information about each registry and Docker engine such as available storage capacity and list of stored layers. Further, it stores information about each placed image such as name of the image, size of the image, number of layers associated with each image, `sha256` hash of each layer, size of each layer, and the location of each layer on the co-operative registry. The goal of IDM is to generate a mapping of individual image *layer to registry*. Once the mappings are decided by the manager, the output is sent to the **Registry Master** for realizing the mapping. The Registry Master communicates with the **Registry Slave** at each node to store the layers (i.e., partial image), copy layers between registries (in the case of re-distribution), and delete layers (in case the layer is not needed). Further, the Registry Master contacts the Image and Node Metadata Manager to update the database accordingly. The goal of the ProM is to minimize the startup time of a distributed container based application. For a set of images, ProM decides a set of destination nodes and tries to provision containers from the locally available image layers on the destination nodes as much as possible. In case some layers are missing, a set of source nodes are identified where the missing layers are present. For each destination node, a mapping of missing layer to source node is then passed to **Docker Master** to perform the distributed pull of missing layers. Specifically, Docker Master communicates with the **Docker Slave** of individual destination nodes to provide the input for `docker dpull`, i.e., a list of *missing layer to registry IP address* mapping and initiates distributed pull. Once the `dpull` completes the download successfully, Docker Master contacts Image and Node Metadata manager to update the database (i.e., layer to node mapping).

The **Resource Monitoring Master** interacts with the **Resource Monitoring Slave** of each node to collect the node's resource utilization. The Resource Monitoring Slave uses `nmon` tool [42] to collect this data and then periodically transfers it to the Resource Monitoring Master. The resource utilization profile of each node is processed at the master to provide information about available resources to the Provisioning Manager. The **Health Monitor**, on the other hand, checks if any registry has failed and initiates the image distribution manager to recover layers (stored at the failed node) in case of a failure. Specifically, this component periodically pings all the registries. A registry is failed if there is no response from it for more than a pre-defined duration. The list of layers stored in the failed node is retrieved from Image and Node Metadata Manager and given as input to the Image Distribution Manager.

A. Image Distribution Manager (IDM)

The input to IDM is a list of images (along with its layers) that need to be distributed, and the output is a list of *layer to registry* mapping. This mapping is used to store the layers on a co-operative registry. IDM uses a set of heuristics to determine the images for which redundant copies (typically 3 or more) can be created across a set of nodes. Initially, all the layers of input images are sorted in increasing or decreasing order of the layer size. From the sorted list, the top-most layer is retrieved and is mapped on a set of nodes. The mapping is done by sorting the nodes in increasing or decreasing order of the available size of the corresponding image registries. Note that, based on the ordering of layers and nodes, four heuristic variants can be created. We evaluate them extensively in Section VI. An image is not HA feasible, if the desired level of redundancy can not be achieved for any layer that belongs to that image. When a layer can not be mapped onto a set of nodes with a desired level of redundancy, all images that need this layer become infeasible for HA. For HA infeasible

TABLE III
CHARACTERISTICS OF REAL WORLD IMAGES UNDER CONSIDERATION.

Total #images	142
Total size of images	58.03 GB
Total #layers	2607
Total #unique layers	1284
Total size of unique layers	45.27 GB

images, all layers which are either already mapped or yet to be mapped are removed. Finally, all layers of HA feasible images along with their corresponding mapped nodes are aggregated and provided as the output.

B. Provisioning Manager (ProM)

The inputs to ProM are: (i) a set of containers that need to be provisioned along with its resource requirements, (ii) the available resources at each node, (iii) the mapping of layers on registries. To provision containers from a set of images, ProM takes a two step approach. First, it tries to place the containers on destination nodes such that: (a) the amount of layer's data to be pulled over network is minimized and (b) the required resources are satisfied. Next, if the destination node does not have all the necessary image layers in its local registry, ProM provides a set of source nodes from which the missing layers can be fetched such that the provisioning time is minimized. Initially, given a set of input images, ProM sorts the images in increasing order of their size. For each image, feasible nodes with enough resource capacity are selected. The destination node is then decided by selecting a feasible node that has the maximum resource capacity and maximum sum-total size of the image layers. Note that, a destination node may have all the layers of an image (i.e., full image) or only a subset of layers (i.e., partial image). ProM then determines the set of missing layers on a selected destination node for a given image. Finally, for each destination node, a unique set of missing layers for all images are provided as the output.

Once the set of destination nodes are decided, ProM determines the set of source nodes for each of the missing layers of every destination node. The key intuition is to create a load-balanced mapping between source and destination nodes. To this effect, for each missing layer, ProM creates a bi-partite graph between a set of source and destination nodes. The edges of the graph are determined for two cases - (i) case-I: when the number of destination nodes is less than or equal to the number of source nodes and (ii) case-II: when the number of destination nodes is greater than the number of source nodes. In case-I, an edge is created between the first destination node (selected in random order) and a source node, which has the largest total size of all the missing layers. For the next destination node, an edge is created with the source node that has the second largest size and so on. In case-II, when the set of source nodes are exhausted while creating the edges, the process is again repeated from the source node with the largest size. The output of ProM is a bipartite graph for each unique missing layer.

TABLE IV
HEURISTIC ALTERNATIVES

Alternatives	Layer/Image Ordering	Node Ordering
LL	Descending	Descending
LS	Descending	Ascending
SL	Ascending	Descending
SS	Ascending	Ascending

VI. EVALUATION

In this section, first, we discuss the experimental setup and the container images that we use to evaluate the performance of *CoMICon*. Second, we present the efficiency of *CoMICon* for high availability and provisioning management.

A. Experimental Setup

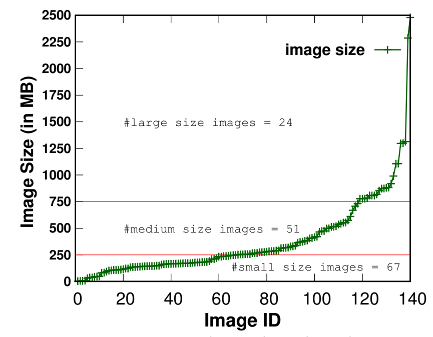
Our setup consists of a PM with 64 `Intel Xeon CPU E5-2698` cores, each running @ 2.3 GHz, 322 GB of memory, and 5 local hard disks each with 10000 RPM. This PM runs on a Ubuntu 14.04 OS with linux `kernel v3.13.0-24` and `QEMU-KVM v2.4` is hosted as a UNIX process in the background. Each set of Docker engine / registry is run on a separate VM created using `QEMU-KVM` on the host PM. In total, we use 15 VMs each hosting a pair of Docker engine `v1.10.1`/registry `v2.4`. Each VM is allocated 4 vCPUs (1:1 mapping with physical CPU), 8 GiB of memory, 50 MB/s disk read, 50 MB/s disk write, and 200 Mbps of network bandwidth. The storage capacity of registry (which stores compressed layers) and Docker engine (which stores uncompressed layers) is set to 15 GB and 100 GB respectively.

In total 142 container images are downloaded from Docker Hub [14]. The characteristics of these images are summarized in Table III. The images are chosen from various domains such as monitoring, software development kit, speech/video processing, and data analytics platform. Figure 6 presents the image category along with their total size.

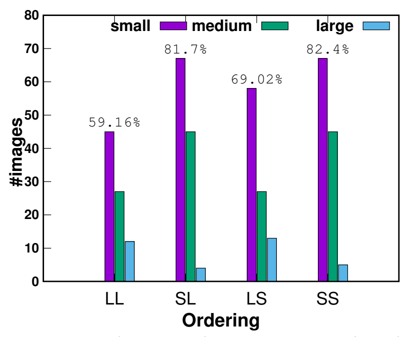
B. HA Image Management

We divide these images into three groups—`small`, `medium`, and `large` based on the size of the images (Figure 5(a)). The `small` group corresponds to images whose size is less than 250 MB, whereas `medium` and `large` groups contain images with size between 250 MB – 750 MB, and greater than 750 MB, respectively. For HA evaluation, we consider the two metrics: (i) % of images that can be placed in co-operative registry in HA feasible manner, and (ii) the time taken to recover a failed node.

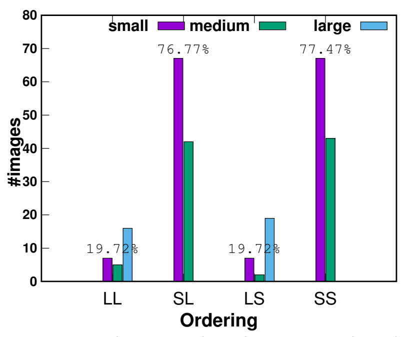
1) Heuristics evaluated: We evaluate *CoMICon* with different alternatives that are used inside IDM. Table IV shows the heuristic alternatives. These are denoted by two lettered abbreviations, where the first and second letter represent the layer and node orderings, respectively. The letter 'L' is used to denote descending order (i.e., Largest layer first or Largest registry size first) whereas 'S' is used to denote ascending order (i.e., Smallest first). We also compare *CoMICon* with image level placement where images as a whole are distributed



(a) Image Size Distribution



(b) Placed images with layer replication



(c) Placed images with image replication

Fig. 5. Feasible HA images.

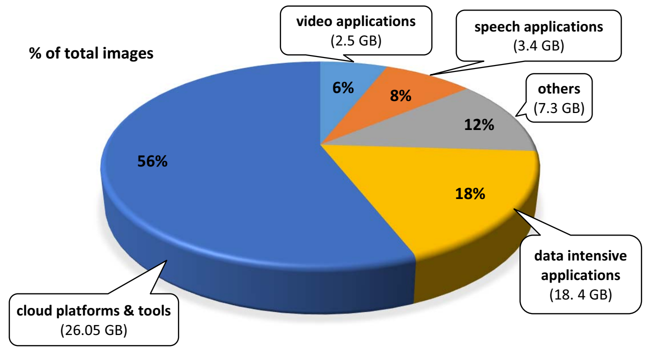
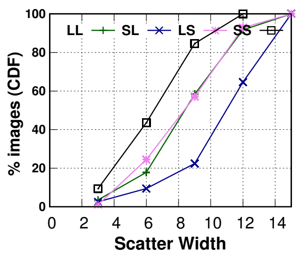
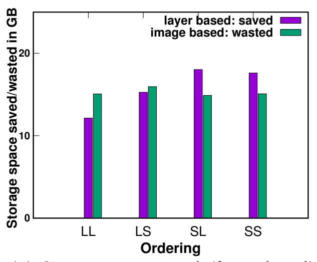


Fig. 6. Distribution of Images.



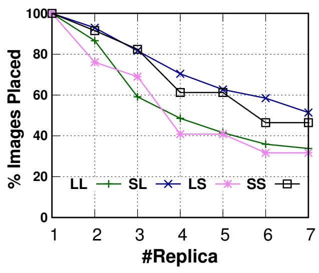
(a) Scatter width (layer based)

Fig. 7. Cumulative Distribution Function (CDF) of images placed with different scatter width (i.e., no. of nodes used to place images).

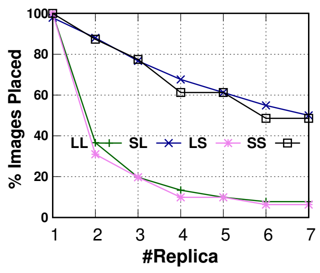


(a) Storage space saved (layer based)

Fig. 8. Amount of storage capacity saved and wasted with layer and image based approach, respectively.



(a) Layer Replication



(b) Image Replication

Fig. 9. Impact of replication factor on the percentage of placed images with respect to both layer and image based replication.

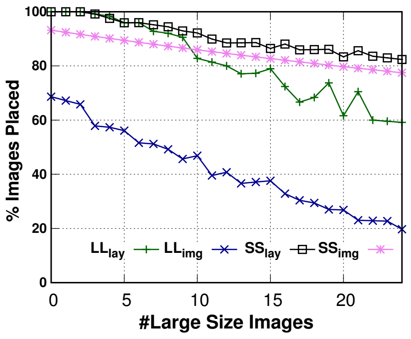
in different registries. For a fair comparison, we apply the same heuristics for image level replication as well.

2) Results: The figures 5(b) and 5(c) show the number of different sized images for the layer and image level replication respectively. The placement is done assuming a 3 level replication for HA. The number of images placed in HA feasible manner is shown in bar chart for the different heuristic alternatives. The bar charts also show the overall % of images placed in a HA feasible manner (e.g., 59.16% for LL). Comparing the Figures 5(b) and (c), it is clear that *CoMICon* consistently leads to larger number of HA feasible image for the same heuristic (e.g., 59.16% vs 19.72% for LL).

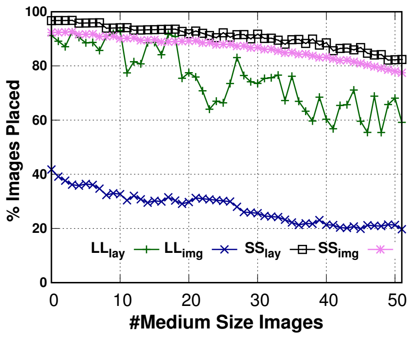
For LL and LS, *CoMICon* leads to 3× or more HA feasible placement than image level distribution. This is primarily attributed to the high number of small images placed for layer level distribution. With a higher modularity at the layer level, a small image can get partially placed in separate registries, allowing the entire image to be placed. Figure 7 shows the CDF of images placed on different number of nodes, also referred as the scatter width.

Further, with a layer level placement, the common layers are replicated in 3 registries. However, in image level placement, the entire set of layers of each image are placed together in 3 registries. Hence the common layers of multiple images get duplicated as part of the corresponding images. For example, one layer common to 2 images gets replicated 6 times as part of the 3-replication placement for the individual images. Such duplications are not possible in *CoMICon*. Figure 8 shows the registry storage space saved by *CoMICon* as well as the wasted storage by the image level placement because of duplication of common layers across images.

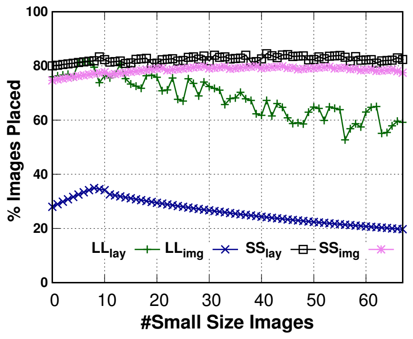
For SS and SL, image level placement does not place any large image in an HA feasible manner (figure 5(c)). This is because there are a large number of small and medium images that get placed first. In SS and SL, the large images do not get selected at all for placement while the registries get filled up by the small and medium images only. *CoMICon* however does allow some large images to get placed, since the layers (of a large image) that are common with smaller and medium images, may not need to be placed again.



(a) Impact of large size images



(b) Impact of medium size images



(c) Impact of small size images

Fig. 10. Impact of varying the number of images of a given type (`small/medium/large`) on the percentage of placed images with respect to both layer (lay) and images (img) based replication.

Figure 9 shows the variation of image replication factor on the % HA feasible placement. As expected, the % of HA feasible placement is decreased with increase in replication factor. When there is single replica placement, all the images could be placed in the set of nodes for all the heuristics alternatives. However, as the replication factor increases, the LL and LS decrease sharply compared to the SS and SL. This is because larger images are placed before the smaller images in the LL and LS alternatives, the registries get utilized more, and hence, causing unavailability of registry space for the smaller images. In the image level placement, because of the inability to partially place an image, the chances of not placing the smaller images are much higher. This leads to a much sharper decrease in the % of HA feasible placement under image level placement for higher replication factors (Figure 9(b)).

For a given type of image (`small/medium/large`), Figure 10 shows the impact of varying the number of images on the % of HA feasible image placement under 3-replication. For example, in Figure 10(a), we keep all the small and medium sized images and increase the number of large sized images from 0–24 (i.e., the total number of images from 118–142).Similarly, for Figures 10(b) and (c) we vary the number of medium and small images, respectively. For clarity of presentation, we only show the LL and SS alternatives for both layer level and image level placement. The LS and SL follow a very similar trend compared to LL and SS, respectively, and are omitted from the plots. This observation essentially means that node ordering does not impact the % of HA feasible image placement when the number of different sized images are varied.

It is further observed that for a given image/layer ordering, the layer level placement always outperforms the image level placement. Overall, with increase in number of images (irrespective of image sizes), the % HA feasible placement decreases. Interestingly, when the large sized images are less, the % HA feasible placement is high for all alternatives (Figure 10(a)). The layer level placement can ensure 100% HA feasible placement under such conditions.

In general, we summarize that layer level placement in *CoMICon* leads to higher HA feasibility than image level placement. The improvement is much more prominent when large sized images are to be placed. The modularity of layer level placement as well as the avoidance of duplicate placement of common layers lead to this improvement. It can be argued, however, that with a layer level placement in *CoMICon*, failures to nodes containing a set of common layers can lead to unavailability of all the images with these layers. However, the modularity allows fast recovery of images to maintain high availability. We perform experiments by failing a single node at a time. Recovery time is determined by the time taken to transfer all the lost layers (or images in case of image level placement) from the non-failed nodes to a new node. Figure 11 shows the recovery time for both layer and image level approaches under failure of different nodes. We notice that the time taken to recover a failed node in the layer level replication approach is lower than the image level replication by up to 34%.

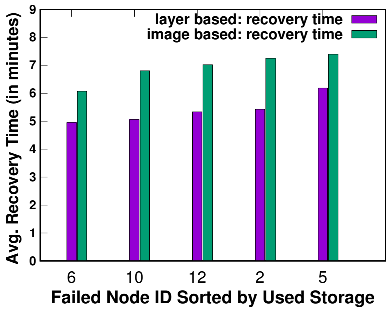


Fig. 11. Impact of layer & image based replication approach on the recovery time.

C. Application Provisioning Management

This section describes the performance evaluation of *CoMICon* with respect to application provisioning. It is important to have the images placed in the co-operative registry so that provisioning time can be minimized as compared to the non-cooperative approach. We assume that the image distribution in the co-operative registry is as per the HA placement mentioned

TABLE V
CHARACTERISTICS OF EACH APPLICATION.

App. ID	size (GB)	#images	#layers
A1	1.6	6	59
A2	2.5	8	233
A3	3.4	12	206
A4	4.7	19	239
A5	5.6	21	244
A6	7.3	17	278
A7	8.4	15	273
A8	13.9	16	413

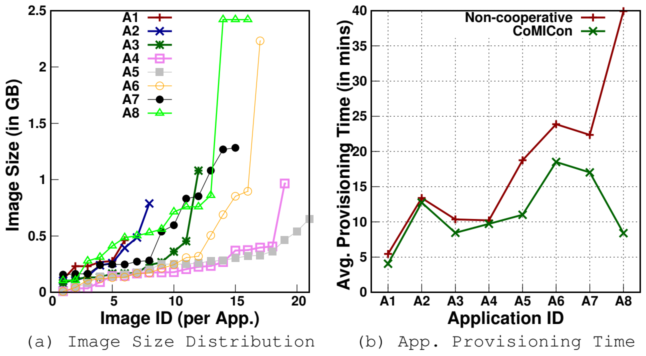


Fig. 12. Image size distribution of each application and the corresponding provisioning time with respect to non-cooperative and co-operative registry.

in the previous section. To quantify the improvement in application provisioning time with *CoMICon* over the non-cooperative approach, we consider eight microservice based applications (A1 to A8) each with different containers and image sizes. Table V presents the number of images, total size and the number of layers with each application. We create these applications by picking images from categories shown in Figure 6. Figure 12(a) shows the image size of each container in these eight applications.

1) Heuristics Evaluated: For the process of application provisioning, *CoMICon* decides on the following: (i) destination node for each container and the missing layers in the destination, (ii) which missing layers are to be pulled from which registry. The Docker master communicates with the Docker slave at all destination nodes to perform the dpull operation. For the non-cooperative approach, all the missing layers are pulled from the non-cooperative registry. At all destination nodes, the pull or dpull command is concurrently executed to enable parallel downloads of layers. Note that the provisioning time of an application is measured as the time when all container instances are started.

2) Results: Figure 12(b) shows the average application provisioning time of each application for both non-cooperative approach and *CoMICon*. On an average, using *CoMICon*, the provisioning time is reduced by 28%. Figure 13 shows the CDF of container provisioning time per application. We observe that the *CoMICon* is able to instantiate containers quickly as compared to the non-cooperative approach. For applications A1 to A4, the improvement in the average provisioning time is not significant (only 13% on an average). This is because, one of the container image is much larger compared to all other images in these applications. As a result, in both dpull and pull, smaller images are downloaded quickly leaving the larger image to the end. When both the operations fetch a single large image, the destination node becomes the resource bottleneck but not the source registry and hence, the difference in provisioning time is less. For applications A5 to A8, the improvement is very significant (42.10% on an average). This is because, most of the time, more than one image is downloaded. This creates a bottleneck at the non-cooperative registry. Figure 14 shows the average CPU utilization, disk read & write rate, network transfer & receive rate across nodes for both non-cooperative and *CoMICon* approaches. We plot the results for four applications (increasing image size). At the registry, the disk read and network transfer operations are observed due to reading of a layer tar file and transfer of the same to a destination node, respectively. At the destination node, the network receive operation is observed due to receiving of a layer tar file from registry, and the disk write operation is observed due to storing of the tar file, and then extracting and writing the layer content to AUFS. Reading of stored tar file for extraction does not increase the disk read rate because of presence of data in the OS page cache.

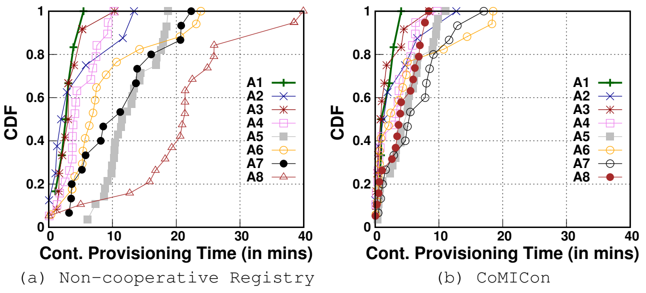


Fig. 13. CDF of container provisioning time per application.

Figure 14(a) shows the average CPU utilization. For the non-cooperative setup, one of the fifteen VMs is used as the source registry to download images. This is shown as the *Non-cooperative:registry* in Figures 14 and 15. As expected, the CPU utilization with *CoMICon* is higher as compared to non-cooperative approach as the layers are downloaded quickly. Figure 14(b) shows the average disk read rate for a *CoMICon* node and the non-cooperative registry. The disk read rate at the non-cooperative registry is higher compared to the *CoMICon* node as download requests for all layers go to the registry. Similar behaviour is observed for the network transfer rate in Figure 14(d). Figure 14(c) shows the average disk write rate for a node in both non-cooperative approach and *CoMICon*. In general, we expect higher disk write rate at the *CoMICon* node (due to distributed pull of layers) as compared to a node in non-cooperative approach. However, for two applications (A1 & A2), we observe lower rate as most of the missing layers are downloaded from local registry which

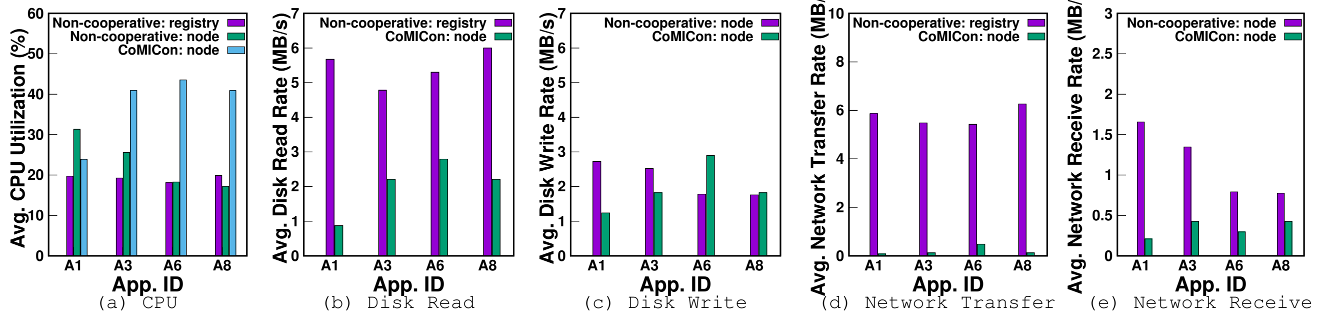


Fig. 14. Average resource utilization per node during application provisioning.

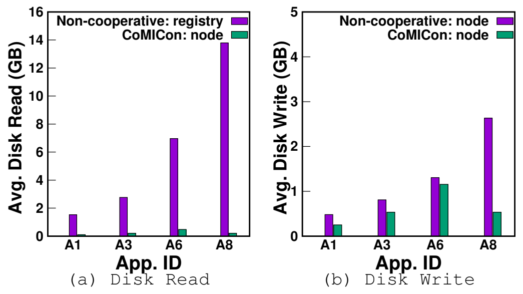


Fig. 15. Overall disk read and write per node.

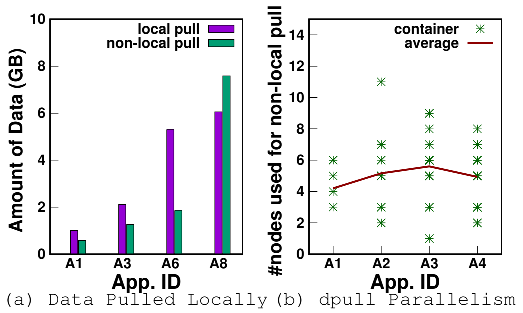


Fig. 16. Local and distributed parallel image pulling in *CoMICon*.

avoids storing of another tar file before extracting it. Figure 14(d) and 14(e) show the network transfer and receive rate for both non-cooperative approach and *CoMICon*, respectively. The network rates are lower with *CoMICon* as some layers are downloaded from local registry. Thus irrespective of the network conditions, the amount of network transfer needed is less as compared with the non-cooperative approach.

Figures 15(a) and (b) show the overall disk read and write (averaged over all nodes) for *CoMICon* and non-cooperative approach. The amount of data read at the non-cooperative registry is equal to the application size, whereas *CoMICon* distributes the data read operations to various nodes as well as to local registry. Further, the average disk write is low for *CoMICon* as downloading layers from local registry avoids storing of the tar file before extraction. Figure 16(a) shows the amount of data pulled from local registry as well as from other registries for four applications. The amount of data pulled over network is much lower as compared to non-cooperative approach. For each application, Figure 16(b) shows the number of nodes that are used in parallel to pull each container image. To summarize, all these efficiencies in the resource usage and parallel pulling in *CoMICon* leads to improvement in the application provisioning time.

VII. Conclusions and Future Research

In this paper, we describe *CoMICon*, a system for co-operative management of Docker container images. The key idea behind *CoMICon* is that, when a collection of Docker nodes are available, instead of relying on a non-cooperative registry, container images can be shared in a cooperative manner. Further, such sharing is facilitated by selectively fetching the desired image layers which in turn significantly reduces the storage requirement and network traffic. Using real images from Docker Hub, we extensively evaluate *CoMICon* for two scenarios: (i) high availability assurance for a set of images and (ii) provisioning of a distributed containerized application for next generation microservices. Results show that compared to the state-of-the-art approach, *CoMICon* can increase high availability assurance by up to 3× while reducing the provisioning time by 28% on an average. By showing the promise of such co-operative image management, we believe that *CoMICon* opens up a number of interesting research problems. For example, (i) how do we ensure the security requirements when all cooperative nodes do not belong to the same owner or provider? (ii) how do the nodes cooperate with each other when anti-colocation constraints are imposed on running containers? (iii) what would be a cost and incentive model to promote such co-operative management in a large public Cloud setup? (iv) how will the system adapt to varying network conditions such as broad- and narrow-band networks? and (v) what are the other feasible algorithms that can be added to the components: ProM and IDM. We plan on addressing some of these problems in the future.

References

[1] S. Soltesz, H. Potzl, M. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *EuroSys*, 2007.

[2] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. Shin, "Performance evaluation of virtualization technologies for server consolidation," HP Laboratories Report, 2007.

[3] "Docker," 2015, https://www.docker.com/.

[4] "CoreOS," 2015, https://coreos.com/.

[5] "Google Container Engine," https://cloud.google.com/container-engine/.

[6] "IBM Containers for packaging apps and services," October 2015, https://www.ng.bluemix.net/docs/containers/container_index.html.

[7] "Azure Container Service," 2015, https://azure.microsoft.com/en-us/blog/azure-container-service-now-and-the-future/.

[8] "Amazon EC2 Container Service," 2016, https://aws.amazon.com/ecs/.

[9] "Google Cloud Platform Blog," June 2014, https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html.

[10] M. Fowler, "Microservices," http://martinfowler.com/articles/microservices.html, 2016.

[11] C. Richardson, "Microservice architecture patterns and best practices," 2016, http://microservices.io/.

[12] "Azure Service Fabric," 2016, https://azure.microsoft.com/en-gb/services/service-fabric/.

[13] "Implementing microservices using Docker containers," August 2015, https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/.

[14] "Docker Hub," 2016, https://hub.docker.com/.

[15] "Kubernetes," 2016, http://kubernetes.io/.

[16] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 181–195. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[17] "Getting Started with Amazon Simple Storage Service," 2006, http://docs.aws.amazon.com/AmazonS3/latest/gsg/GetStartedWithS3.html.

[18] "Get started with Azure Blob storage using .NET," 2016, https://azure.microsoft.com/en-gb/documentation/articles/storage-dotnet-how-to-use-blobs/.

[19] "HDFS Architecture," https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2016.

[20] "Peer-to-peer push/pull between docker hosts," March 2013, https://github.com/docker/docker/issues/247.

[21] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 37–48. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/cidon

[22] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer, "Tiered replication: A cost-effective alternative to full cluster geo-replication," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 31–43. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/cidon

[23] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's warm blob storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 383–398. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar

[24] S. Nadgowda, P. Jayachandran, and A. Verma, "i2map: Cloud disaster recovery based on image-instance mapping," in *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*. Springer Berlin Heidelberg, Dec. 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45065-5_11

[25] Y. Wu and G. Huang, "Model-based high availability configuration framework for cloud," in *Proceedings of the 2013 Middleware Doctoral Symposium*, ser. MDS '13. New York, NY, USA: ACM, 2013, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/2541534.2541595

[26] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang, "Virtual machine images as structured data: The mirage image library," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=2170444.2170466

[27] K. Razavi, A. Ion, and T. Kielmann, "Squirrel: Scatter Hoarding VM Image Contents on IaaS Compute Nodes," in *Proceedings of the 23rd International Symposium on High Performance Distributed Computing*, ser. HPDC '14, 2014.

[28] K. Razavi, G. V. D. Kolk, and T. Kielmann, "Prebaked μ vms: Scalable, instant vm startup for iaas clouds," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, June 2015, pp. 245–255.

[29] B. Nicolae, A. Kochut, and A. Karve, "Discovering and leveraging content similarity to optimize collective on-demand data access to iaas cloud storage," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, May 2015, pp. 211–220.

[30] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu, "VMThunder: Fast provisioning of large-scale virtual machine clusters," *IEEE TPDS*, vol. 25, no. 12, pp. 3328 – 3338, 2014.

[31] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu, "Going back and forth: Efficient multideployment and multisnapshotting on clouds," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996152

[32] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein, "Vmtorrent: Scalable p2p virtual machine streaming," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413210

[33] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath, "Image distribution mechanisms in large scale cloud providers," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Nov 2010, pp. 112–117.

[34] C. M. O'Donnell, "Using bittorrent to distribute virtual machine images for classes," in *Proceedings of the 36th Annual ACM SIGUCCS Fall Conference: Moving Mountains, Blazing Trails*, ser. SIGUCCS '08. New York, NY, USA: ACM, 2008, pp. 287–290. [Online]. Available: http://doi.acm.org/10.1145/1449956.1450040

[35] A. Hegde, R. Ghosh, T. Mukherjee, and V. Sharma, "Scope: A decision system for large scale container provisioning management," in *Proceedings of the 9th IEEE International Conference on Cloud Computing (IEEE Cloud)*, San Francisco, USA, 2016.

[36] T. Rice, "TheDistributedBay," https://github.com/TheDistributedBay/TheDistributedBay/, 2015, [Online; accessed 29-Apr-2015].

[37] I. Babrou, "p2p distribution for docker images," https://github.com/bobrik/bay/, 2014, [Online; accessed 08-December-2014].

[38] L. Meng, "A image hub for rtk & docker," https://github.com/containerops/dockyard/, 2016, [Online; accessed 2016].

[39] "Select a storage driver," https://docs.docker.com/engine/userguide/storagedriver/selectadriver/, 2016.

[40] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "OLTP-Bench: An extensible testbed for benchmarking relational databases," in *VLDB*, 2013.

[41] "Vdbench," 2016, http://www.oracle.com/technetwork/server-storage/vdbench-downloads-1901681.html.

[42] "Nmon for linux," 2016, http://nmon.sourceforge.net/pmwiki.php.