Bolt: Towards a Scalable Docker Registry via Hyperconvergence

Michael Littley[1], Ali Anwar[2], Hannan Fayyaz[3], Zeshan Fayyaz[4],
Vasily Tarasov[2], Lukas Rupprecht,[2], Dimitrios Skourtis[2], Mohamed Mohamed[2],
Heiko Ludwig[2], Yue Cheng[5], and Ali R. Butt[1]

[1]*Virginia Tech,* [2]*IBM Research–Almaden,* [3]*York University,* [4]*Ryerson University,* [5]*George Mason University*

***Abstract*—Docker container images are typically stored in a centralized registry to allow easy sharing of images. However, with the growing popularity of containerized software, the number of images that a registry needs to store and the rate of requests it needs to serve are increasing rapidly. Current registry design requires hosting registry services across multiple loosely connected servers with different roles such as load balancers, proxies, registry servers, and object storage servers. Due to the various individual components, registries are hard to scale and benefits from optimizations such as caching are limited.**

In this paper we propose, implement, and evaluate BOLT—a new hyperconverged design for container registries. In BOLT, all registry servers are part of a tightly connected cluster and play the same consolidated role: each registry server caches images in its memory, stores images in its local storage, and provides computational resources to process client requests. The design employs a custom consistent hashing function to take advantage of the layered structure and addressing of images and to load balance requests across different servers. Our evaluation using real production workloads shows that BOLT outperforms the conventional registry design significantly and improves latency by an order of magnitude and throughput by up to $5\times$. Compared to state-of-the-art, BOLT can utilize cache space more efficiently and serve up to 35% more requests from its cache. Furthermore, BOLT scales linearly and recovers from failure recovery without significant performance degradation.

I. Introduction

Container management frameworks such as Docker [10] and CoreOS Container Linux [4] have given rise to a rapid adoption of containers [30], [24]. Compared to virtual machines, containers do not require their own operating system but rather share the underlying kernel. This allows fast software deployment and low performance overhead [21]. Besides their performance benefits, the lightweight nature of containers has also enabled *microservice* architectures as a new model for developing and distributing software [25]. In this architecture, individual software components, focusing on small functionalities, are packaged in container *images* that include the software and all its dependencies. These *microservices* can then be deployed separately and combined to construct larger, more complex architectures [29].

Container images are kept in an online store called *registry*. A registry is a storage and content delivery system, holding named Docker images split into one or more *layers* and available in different tagged versions. Some popular Docker registries are Docker Hub [6], Quay.ioi [9], or IBM Cloud Container Registry [7]. As the registry is a central access point for users to publish and retrieve images, it becomes a critical component in the lifecycle of a container [22], [26]. Recent work has shown that pulling images from a registry of such scale can account for as much as 76% of the container start time [22].

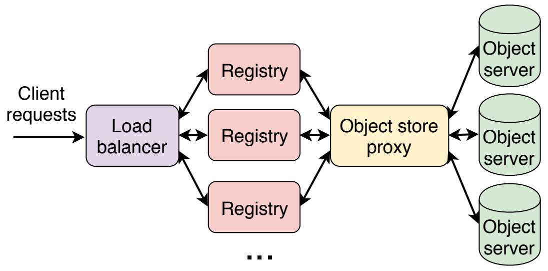


Figure 1: Example for a production level registry.

The current Docker registry software server is a single-node application. To serve many requests concurrently, organizations typically deploy a load balancer in front of several independent registry instances. All the instances store and retrieve images from a shared backend object store. The object store often requires its own load balancer and proxy servers to scale. Figure 1 illustrates a typical example of a production registry deployment. We identify three main problems with the existing registry design: 1) low cache efficiency, 2) high complexity of scaling, and 3) high latencies caused by request hopping.

Cache efficiency. In its current design, the registry only caches relevant metadata in memory but does not cache the corresponding image data. This can lead to the registry becoming a bottleneck when large volumes of hot images are retrieved. Recent work proposed a registry-level image cache [15]. However, registries in the existing design are scaled using a load balancer and hence are not aware of each other. As a result, container images may be cached redundantly by different registry instances. To demonstrate this drawback we conducted an experiment using production level traces from the IBM Cloud container registry and the

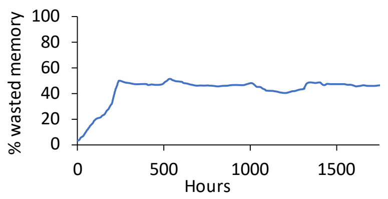


Figure 2: Percentage of registry cache used by redundant copies of image layers as time goes.

trace replayer from [15]. Figure 2 shows that at any given time around 50% of the registry cache is wasted due to caching the same layers across different registry nodes.

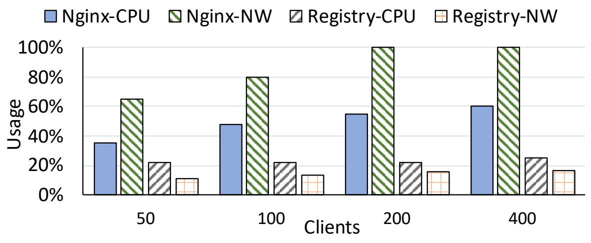
Complexity of scaling. Due to its complex architecture, any component in the registry is prone to become a bottleneck depending on the workload. We illustrate this problem by using the trace replayer to send requests to a simple setup of two registry servers behind an NGINX load balancer. In two experiments the trace replayer generates two distinct workloads: one that requests 1 MB sized layers and one that requests 1 KB sized layers. The experiment results are illustrated in Figure 3. In the first workload we observe that the network on the NGINX load balances becomes the bottleneck as it limits the achieved throughput. In the second workload, the registry server becomes CPU-bound because small layer sizes makes the workload compute intensive. These results show that scaling an existing registry is not as easy as adding resources and requires a careful and cumbersome analysis.

High latency. Finally, current registry deployments require multiple hops to process every request. For example, in Figure 1, every request needs to traverse up to four distributed registry components. This causes Docker clients to experience unacceptably high latencies. We demonstrate this issue in detail in section V-A.

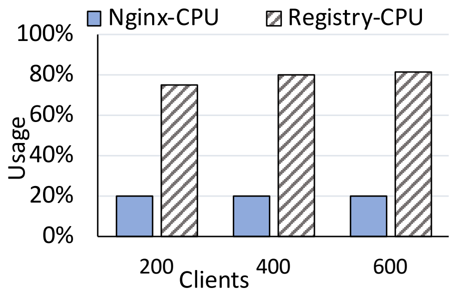
In this work, we present BOLT, a distributed Docker container registry. BOLT aims at providing a scalable registry design with a native image cache for better request performance. Specifically, BOLT follows a *hyperconverged* archictecture [18] to overcome the problems of the existing registry design. BOLT moves storage to individual registry nodes and eliminates the need for a separate storage backend. Additionally, BOLT improves registry scalability *storage-aware* by employing a distributed design based on consistent hashing and Zookeeper [11]. Finally, BOLT pushes load-balancing logic into clients and thereby removes the additional load balancing layers. Clients directly send requests to registry nodes to retrieve images, which reduces latency and simplifies the overall deployment.

In summary, in this paper, we make the following contributions:

1) We demonstrate the shortcomings and limitations of the existing Docker registry design using production-level traces from the IBM Cloud container registry.
2) We propose BOLT, a new hyperconverged design for container registries, which overcomes the problems of the existing registry design.
3) We implement a prototype of BOLT, based on the current Docker registry, which supports caching natively, allowing a higher cache space utilization, and scales linearly.



(a) 1 MB requests.



(b) 1 KB requests.

Figure 3: Result experiment from sending requests to a setup with 1 NGINX load balancer and 2 registry servers.

We evaluate BOLT on an 11-node testbed and show that it outperforms the conventional design by improving latency by an order of magnitude, and throughput by up to 5×.

II. Background

The Docker container management system is an open source software consisting of a Docker engine running on a client machine and a remote registry for storing and distributing Docker container images. The components of the Docker system are illustrated in Figure 4. In the following, we describe different Docker components in more detail.

A. Docker engine

The Docker engine consists of a daemon process and a command line interface (CLI) client that interacts with the daemon via a REST API. The daemon is responsible for creating and managing containers as well as fetching container images from the Docker registry. Additionally, the daemon sets up network and file system functionality for its containers.

B. Docker containers

A container image consists of the executables, dependent libraries, and configuration files required to run the applica-

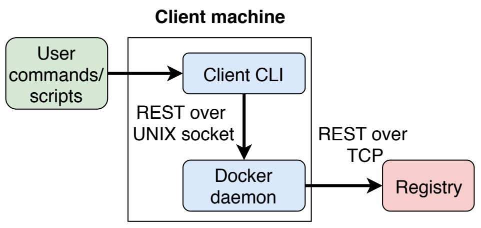


Figure 4: Relationship between different parts of Docker Engine.

tion in an isolated manner. The files are organized into read-only file systems that are compressed into tarballs called *layers*. These layers are fetched from the Docker registry using SHA256 based, content-addressable storage. The hashes of the layers, along with other container metadata, are stored in a JSON file called the *manifest*.

Starting a container consists of several steps. For example, if the user executes the `docker run ubuntu` command using the CLI, the request is forwarded to the daemon running on the local machine. The daemon fetches the manifest file and layers of the Ubuntu container image from the registry, if they are not available locally. The Docker daemon then creates a container from the image layers by setting the proper Linux namespaces [8] and control groups [28] configurations. Next, a writable file system is allocated for the container and a network interface defined for it. Finally, the daemon starts the container.

C. Docker registry

The Docker registry is a stateless, centralized service that provides image storage and distribution. Users may store multiple versions of their images in repositories by denoting each version with a special tag. For storage, the registry supports multiple backends such as in-memory, local file system, Amazon S3, Openstack Swift, Google Cloud Storage, etc. The in-memory driver is reserved for small registry test bed while the local file system is meant for single node registry deployments. All other drivers use backend objects stores for scalability.

The daemon connects to the registry via a RESTful interface. The main requests served by the Docker registry are related to image push and pull operations:

Pull requests. To pull an image, first, the manifest is requested using a `GET` request. Next, the daemon issues a `GET` request for each layer referred to in the manifest that does not exist locally. Once a layer is fetched, a hash of its contents is compared to its digest to ensure layer integrity. The image pull is complete once all the layers are fetched.

Push requests. The sequence of requests required to push an image happens in the reverse order as that of the pull request. The daemon first issues a `HEAD` request to check for the layer's existence in the registry. If the layer is not found, the daemon will upload it to the registry using a `POST` request. When a layer is received, the registry compares the hash of the layers content with the digest provided by the client. If the hashes match, the upload is successful and the registry moves the layer into content addressable storage. Once the upload process finishes for all layers, the manifest file is uploaded using a `PUT` request.

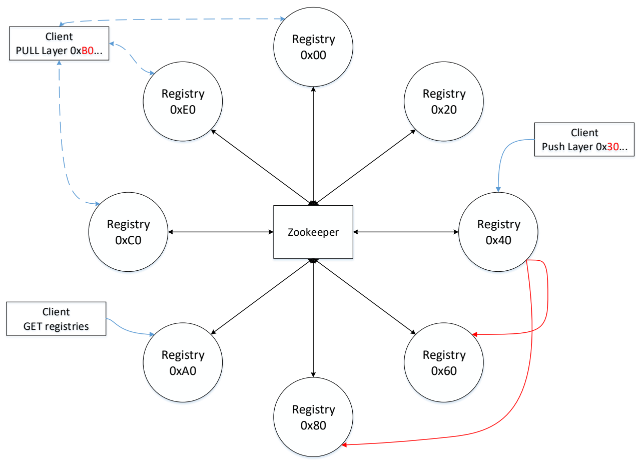


Figure 5: BOLT hyperconverged design.

III. THE BOLT REGISTRY DESIGN

In this section, we describe BOLT's design. After discussing its overall architecture, we explain main system components and their interactions.

A. BOLT *Overview*

BOLT follows a *hyperconverged* archictecture [18] to overcome the problems of the existing registry design. Therefore, BOLT incorporates three main design decisions:

First, it moves storage to individual registry nodes and eliminates the need for a separate storage backend. This simplifies the scaling process for administrators: to increase registry performance they just need to add additional nodes that provide both extra storage and compute resources.

Second, it uses a distributed design based on consistent hashing and Zookeeper [11]. This makes registry nodes *storage-aware* and allows for simpler and more efficient caching strategies as now each registry node is responsible for a dedicated subset of image data.

Third, it pushes load-balancing logic into clients and thereby, removes the additional load balancing layer. Clients directly send requests to registry nodes to retrieve images, which reduces latency and simplifies the overall deployment.

Figure 5 illustrates the design. The individual registry nodes form a consistent hashing ring and use Zookeeper to identify each other. To become aware of all nodes, clients can request a list of all registries from any registry to form a copy of the ring. Clients look up layer digests in their ring to issue push and pull requests to the responsible nodes. Clients must push layers to the master node of the layer, defined as the node immediately greater than the layer digest. On a push, the master node forwards the layer to a set of other registries (slave nodes) for replication. Clients can direct pull requests randomly to the master or any of the slave nodes.

B. Storage-aware Registry Nodes

BOLT eliminates the reliance on a backend object store completely by allowing registry nodes to persistently store the layers. As shown in [15], the entire three months of IBM workload can be held in less than 11 TB of SSD so this feasible. However, by eliminating the backend object store, its duties such as providing reliability and consistency now have to be performed by the registry itself.

To achieve this, BOLT uses a master/slave topology. Each layer has a master node, which is responsible for storing the layer. The master node is decided by proximity in the consistent hashing ring. The slave nodes are the next N nodes in the ring, where N is usually set to 2.

Pushing a layer to the registry works as follows: During a layer push, the data is first stored in a temporary file. Once the layer upload is complete, the registry verifies the hash of the file with the digest provided by the PUT request. If there are no errors, the registry moves the temporary file into a directory named by layer's digest, at which point the layer is ready to be served. To facilitate replication in BOLT, while the file is being verified, the registry checks whether it is a master or slave for the layer. If the registry is the master, it forwards the layer to the slave registries to create replicas. If the registry is a slave it proceeds as if it were a standard registry. If the registry determines that it is neither the master or the slave, it deletes the file and returns an error to the client that issued the push.

The replication provides reliability and availability for layers. Users can choose between strong consistency provided using chain replication protocol or eventual consistency. As all layers are content addressable and hence, each layer update generates a new layer digest, BOLT do not have to deal with inconsistencies among replicas due to layer updates.

C. Distributed Coordination

BOLT implements a distributed coordination mechanism to improve scalability. To deal with scale out/in, determine the correct replication targets, and allow clients to directly request layers from responsible nodes, registry nodes must be aware of each other and be informed of node additions/removals. To meet this requirement, BOLT uses Zookeeper as a metadata service to monitor the health and availability of all registries in the system. Zookeeper also facilitates distributed coordination.

Zookeeper itself is a distributed service that organizes data into *Znodes*. It accepts new Znodes or updates to existing Znodes via a consensus of the nodes. Zookeeper clients are able to set watches on Znodes that notify them of changes to the Znode. Znodes are organized into file system like structures, where one root Znode acts like a directory for all other Znodes. However, unlike normal file systems, Znodes can have data associated with them as well as child Znodes. Zookeeper also supports ephemeral Znodes, which exist as long as the session that created them is active, and are deleted when the session is terminated.

BOLT makes use of ephemeral Znodes to register active registries. When a registry is created, it first registers itself with Zookeeper by creating an ephemeral node under a root Znode. If the root does not exist, the registry creates it and then creates its ephemeral Znode. It then establishes a watcher on the root, which provides the registry with the addresses of all other registries in the system, and notifies of any changes. Because the registries use ephemeral Znodes, all other registries are notified in case the registry goes offline and its Znode is removed. The registries use their knowledge of each other to populate their consistent hashing rings, and to provide clients with a list of active registries.

Clients can request the list of registries in the system from any registry it has previously known about, or learns about through another means, such as DNS. Docker clients can issue this request when they first come online, or when a layer request fails (see Section III-B), allowing the clients to update their consistent hashing rings.

D. Replica-aware Clients

Replica-aware clients can directly query registry nodes for desired data. However, in this design, the clients (Docker daemons in this case) need to know which registries store which layers. Therefore, the daemons use a specialized consistent hashing function on the layers to identify the location of the layers and load balance their requests across registry nodes.

As described above, BOLT employs consistent hashing for determining the location of a layer and for load balancing layer requests across the registry nodes. BOLT uses a custom consistent hashing function to take advantage of how layers are already addressed. As layers are identified by their digests (a SHA256 hash of the layer content), the digests can be used directly to map a layer to a registry. As common in consistent hashing, each registry uses a set of pseudo identities to help distribute the layers across the node.

Two different hash functions are used for layer pushes and pulls. For layer pushes, the write hash function returns the first registry whose pseudo identity is immediately greater than the layer's hash. This is the master node for the pushed layer. Algorithm 1 shows that the master node ID for the pushed layer digest is determined efficiently via binary search.

Algorithm 1: Write Hash Algorithm.

```
Input: $Layer$: SHA256 hash of layers. $nodeIDs$: List of
       SHA256 psuedo IDs. $Nodes$: map of pseudo ID to node.
1 begin
2     $ID \leftarrow BinarySearch(nodeIDs, Layer)$
3     $return\ Nodes[ID]$
```

Algorithm 2: Read Hash Algorithm.

```
Input: $Layer$: SHA256 hash of layers. $nodeIDs$: List of
       SHA256 psuedo IDs. $Nodes$: map of pseudo ID to node.
1 begin
2     $ID \leftarrow BinarySearch(nodeIDs, Layer)$
3     $ReplicaNodeList.append(Nodes[ID])$
4     $index \leftarrow nodeIDs.indexOf(ID)$
5     while $Length(ReplicaNodeList) < NumberOfReplics$ do
6         $index \leftarrow index + 1\%length(nodeIDs)$
7         $ID \leftarrow nodeIDs[index]$
8         $node \leftarrow Nodes[ID]$
9         if $node\ not\ in\ ReplicaNodeList$ then
10            $ReplicaNodeList.append(node)$
11    $return\ RandomChoice(ReplicaNodeList)$
```

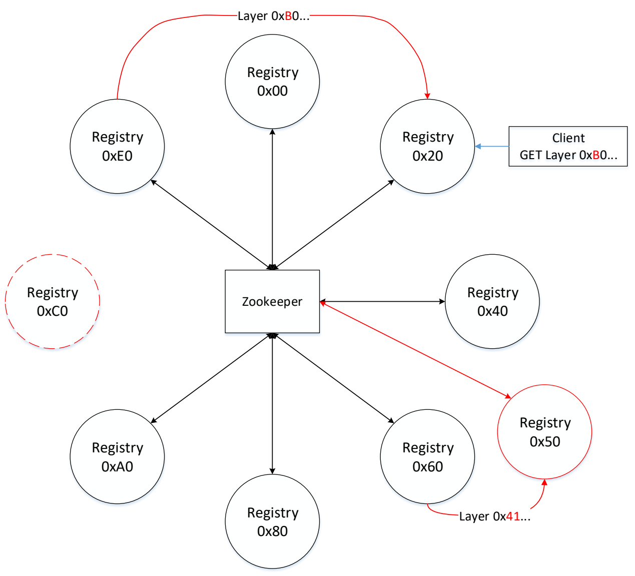


Figure 6: Scalability and failure recovery in BOLT.

For pull requests, the read hash function selects a target node from a set of registries containing a replica of the layer, i.e. the slave nodes. The read hash is described in Algorithm 2. First, the master node ID is determined (Line 2) and added to the list of target nodes (Line 3). The list of nodes is then searched for the replica holders by going clockwise around the consistent hashing ring (Lines 5–10). Finally, a random node in the complete list is returned (Line 10). Compared to the existing, replica-oblivious design, replica-aware clients require less request hops for better performance and no separate load balancing infrastructure for better manageability.

IV. IMPLEMENTATION

Next, we describe the implementation details of BOLT for its improved caching and scalability. BOLT is written in Go and based on the Open Source Docker registry [5].

A. Caching

BOLT implement efficient layer caching in registry nodes. Due to the hyperconverged architecture, layers are not cached redundantly, which allows for more layers to be cached. As manifest files are small and are handled by services like Redis in the existing registry design, we only focuses layer caching.

In BOLT, layers are cached using the Bigcache [2] library for Go. This library provides a fast LRU cache for larger objects with little to no overhead by eliminating Go's garbage collection for the cache. Although the cache is configurable in the design, limiting the size of the layers cached to 1 MB typically improves performance as it allows more layers to be held in memory.

Due to the content addressable nature of layers we do not have to worry about cache invalidation as any update in the layer also updates the layer's digest, which automatically invalidates the cached layer. In total, we added 905 lines of code to the registry to introduce the caching functionality.

B. Scalability

BOLT supports dynamic addition and removal of registry nodes to existing setups, allowing for easy scale out/in of the registry deployment. Once a registry node detects a new node via Zookeeper, it loops through all locally available layers to detect if the new registry is the master of any of them. If so, it forwards those layers to the new registry.

Registries can fetch layers from the master registry if they act as their slave. This happens when clients request layers from registries that have been recently added, or for registries that become slaves as a result of another registry leaving or failing. During the fetch, the registry writes the layer to a temporary file and then renames the file to the correct name and serves it. This prevents other clients from requesting the layer during the fetch from receiving a partial layer.

Figure 6 depicts an example for a node arrival. In the figure, Node 0x50 is joining the registry. On receipt of the notification of the new arrival, node 0x60 will pass some of it's layers, e.g. 0x41 to node 0x50.

The example also illustrates the case of node departure. Node 0xC0 went offline and hence, the layers for which node 0xC0 was a master node are now owned by node 0xE0. Additionally, node 0x20 becomes a new slave node for those layers. When a client attempts to retrieve a layer from 0xC0, the request will fail, prompting the client to issue a registry list requests from the other registries. Once the clients have the correct ring, they might attempt to request a layer from 0x20, which has become the new slave. In that case, 0x20

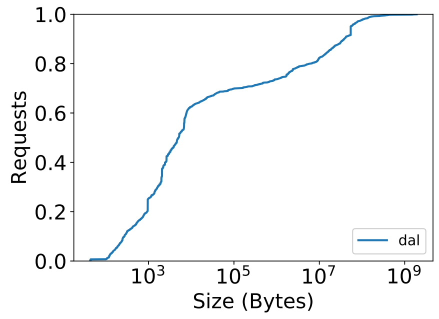


Figure 7: CDF of the layer sizes in the workload trace.

will fetch the layer from the new master and then reply to the client.

V. Evaluation

By evaluating Bolt, we aim to answer five main questions: 1) how does Bolt compare to the existing registry design; 2) how well is Bolt able to balance load; 3) how effective is Bolt's caching approach; 4) how does Bolt scale; and 5) how does Bolt recover from failures.

As a workload we use a production traces from an IBM registry from July 24th, 2017 [15]. This trace contains around 0.26 Million requests in total and holds over 1000 clients connected in parallel. Figure 7 shows the size distribution of the first 5000 requests, containing just over 1500 layers with an average layer size of 13.7 MB. The trace also contains 4% write requests.

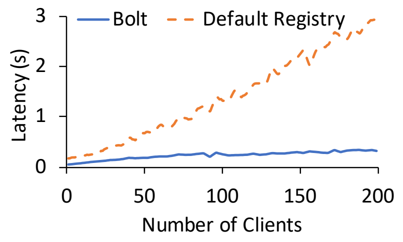
To replay the workload, we use the corresponding trace re-player [15]. We modify the trace re-player to add consistent hashing to the warm-up mode, i.e. when layers are assigned to registry nodes and the clients.

Our testbed consists of 11 machines, each of them having 8 cores, 16 GB of RAM, 512 GB of SSD storage, and 10 Gbps networking. Bolt uses one node as a dedicated Zookeeper node. Every experiment uses 50 pseudo identities per registry node and 3 replicas per layer.

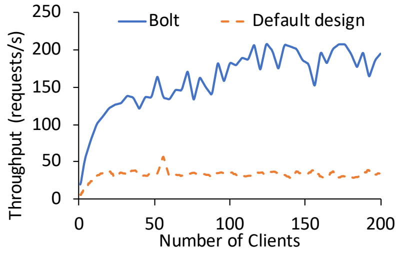
A. *Throughput and Latency Comparison*

We start by comparing Bolt to the conventional distributed registry. Both registries are deployed on 6 nodes. Bolt is configured to have 8 GB of RAM per node. The conventional distribution uses Swift as its backend object store, which is co-located with the registry nodes. The default registry uses a dedicated node as a NGINX load balancer. Each registry connects to a Swift proxy which is running on the same node to prevent a single proxy bottleneck. Both Swift and Bolt are configured to have 3 replicas.

We compare the performance of both systems by increasing the number of clients connecting to the registry. Bolt uses consistent hashing for load balancing, whereas



(a) Latency



(b) Throughput

Figure 8: Throughput and latency for Bolt and the default registry.

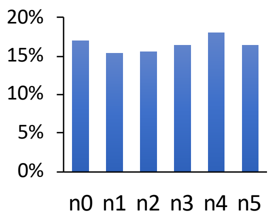


Figure 9: Request distribution.

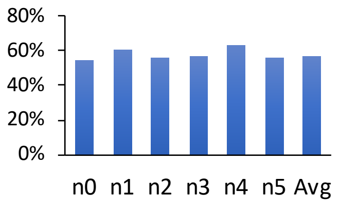


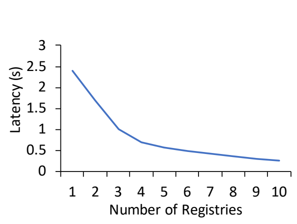
Figure 10: Cache hit ratios.

the conventional registry setup uses round robin. In each experiment, the clients send 5000 requests.

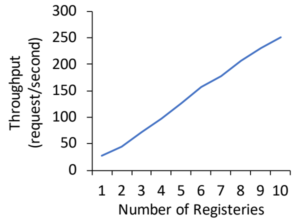
Bolt outperforms the conventional registry design significantly and improves latency by an order of magnitude and throughput by up to 5× (see Figure 8). This is due to both Bolt's caching of layers and the clients' ability to request the layer with one network hop. In the regular setup, clients must communicate via NGINX, which load balances the requests to the 6 registries, and the registries use the Swift proxy to fetch the layer from one of the object nodes, which causes additional hops between nodes and reduces latency. As the regular setup does not cache layer data, throughput is significantly lower.

B. *Load Balancing*

To determine the load balancing capabilities of Bolt's consistent hashing algorithm, we look at the layer distribution for a 6-node deployment of Bolt. We hash the `http.request.uri` field across all layer `GET` and `PUT` requests to represent the layers as we do not have the actual



(a) Latency



(b) Throughput

Figure 11: Throughput and latency for increasing number of registry nodes.

layer contents. As shown in Figure 9, the layers from the trace are distributed almost evenly on the registry nodes, with a maximum discrepancy between two nodes of 3%.

C. *Caching Effectiveness*

To evaluates BOLT's caching effectiveness, we use the trace re-player with 4 client nodes to send 5000 requests to a 6-node registry deployment. Each registry node uses an 8 GB cache and only caches layers that are less than or equal to 1 MB. Each trace player client node spawns 50 docker clients to send the requests. All clients send requests in parallel.

The results (see Figure 10) show that the average hit ratio is 57%. However, it's important to note that 25% of the layers requested were greater than 1 MB. Hence, the effective hit ratio for *cached* layers is 76%. The request latency for cached layers is an order of magnitude less than uncached layers.

Compared to previous work on registry caching [15], we observe that BOLT can serve up to 35% more requests from its cache. This is because BOLT uses replication-aware clients and does not waste cache space by storing copies of already cached layers as shown in Figure 2).

D. *Scalability*

To evaluate scalability, we increase the number of registries from 1 to 10 and measure request throughput and latency as seen by the clients. For this experiment, each registry uses 2 CPUs and 4 GB of RAM. The cache size for each registry is set to be 2 GB. We alter the clients to request each layer from each slave registry in order to fully populate newly added nodes.

The results of the experiment are shown in Figure 11. We observe a linear increase in throughput as the number of registries are scaled up. Analogously, latency also decreases for more registries. Due to it's design, scaling out BOLT only required to adding new registry nodes and registering them with Zookeeper.

E. *Fault Tolerance*

Next, we evaluate BOLT's ability to recover from faults. Therefore, we kill a registry 20 s after starting the experiment

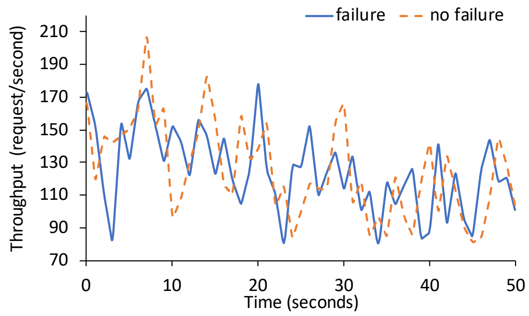


Figure 12: Throughput with and without failure of a registry node.

and bring it back up 10 s later. We use 5 registries for this experiment, and 10000 requests and measure client throughput every second.

Figure 12 shows the results and time 0 is equal to 20 s into the experiment, when one registry node was taken down. We see that the ephemeral Znode for the killed registry exists for 3 seconds after the node is killed, causing the clients' requests to fail. Once the Znode is removed, the registries update their rings, which provides the clients accurate registry lists, preventing more requests from failing. When the node is brought back up after 10 s, the other registries become aware of it within milliseconds.

VI. RELATED WORK

Improving Docker performance is a new research area that is rapidly gaining widespread attention. One area that has specifically received a lot of attention is how to best use the storage drivers for the Docker daemon. [31] provides a study on the impact of different storage drivers on different types Docker workloads. Alternatively, [20] provides a layer between the driver and the Daemon which improves performance for dense container workloads. As our work is focused on improving the Docker registry, it is orthogonal to such approaches.

Recent work proposed a registry-level image cache [15]. However, registries in the existing design are scaled using a load balancer and hence are not aware of each other. As a result, container images may be cached redundantly by different registry instances.

Slacker [22] improves container start-time performance and distribution by incorporating a network file system (NFS) server into both the Docker daemon and registry. This design has two drawbacks: First, in the standard Docker implementation, the image is fetched on container startup and stored locally, so in subsequent container startups all the layers will already be present on the machine, resulting in faster startup times. However, with Slacker, the necessary layers must be fetched from the NFS every time the container starts. Second, Slacker tightly couples the worker nodes with the registry. Docker clients need to maintain a constant connection to the NFS server to fetch additional layers as they need them which can present a challenge to services like Docker Hub.

Dragonfly [1] is a peer-to-peer (p2p) based file distribution system, recently open sourced by Alibaba, which is compatible with Docker. Similarly, CoMiCon [26] proposes a p2p network for Docker clients to share Docker images. In a p2p based approach, each client has a registry daemon responsible for storing partial images with specialized manifest files, copying layers to other nodes, and deleting layers from its node to free space. This approach is limited to only a trusted set of nodes including the Docker clients, and so is not suitable for any registry deployments outside of a single organization. Furthermore, CoMiCon creates changes to image manifests and depends on a new command, so it lacks backwards compatibility.

FID [23] integrates the Docker daemon and registry with BitTorrent [3]. When images are pushed to the registry, the registry creates torrent files for each layer of the image and then seeds them to the BitTorrent network. When images are pulled, the manifest is fetched from the registry and then each layer is downloaded from the network. BitTorrent exposes Docker clients to each other which can become a security issue when worker nodes do not trust each other. Additionally, FID only uses one registry which creates a single point of failure if the registry node goes down. This is because the registry is still responsible for handling manifests and Docker pushes.

A large body of research studied the effect of caching + prefetching [19], [27], [32], [33], [12], and resource optimization [16], [17], [13], [14]. Identifying a number of optimization opportunities, our work demonstrates that these techniques can be utilized to bring dramatic performance improvement.

VII. Conclusion

The Docker registry plays a critical role in providing containerized services. However, due to the various individual components, registries are hard to scale and benefits from optimizations such as caching are limited. In this paper, we proposed, implemented, and evaluated BOLT, a new hyperconverged design for container registries. BOLT makes different registry nodes aware of each other and improves the client-to-registry mapping by using a hash function to take advantage of how layers are addressed by the Docker clients.

We evaluated BOLT to test its performance, caching effectiveness, scalability, and its ability to recover from faults. Our analysis shows that BOLT outperforms the conventional registry design significantly and improves latency by an order of magnitude and throughput by up to $5\times$. Compared to state-of-the-art, BOLT can utilize cache space more efficiently and serve up to 35% more requests from its cache. Furthermore, BOLT scales linearly and recovers from failure recovery without significant performance degradation.

VIII. Acknowledgment

This work is sponsored in part by the NSF under the grants: CNS-1405697, CNS-1615411, and CNS-1565314/1838271.

References

[1] alibaba Dragonfly. https://github.com/alibaba/Dragonfly, visited 2018-02-12.

[2] Bigcache. https://github.com/allegro/bigcache, visited 2018-02-12.

[3] BitTorrent. http://www.bittorrent.com/, visited 2018-02-12.

[4] CoreOS. https://coreos.com/, visited 2018-02-12.

[5] Docker-Registry. https://github.com/docker/docker-registry, visited 2018-02-12.

[6] Dockerhub. https://hub.docker.com, visited 2018-02-12.

[7] IBM Cloud Container Registry. https://console.bluemix.net/docs/services/Registry/index.html, visited 2018-02-12.

[8] Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, visited 2018-02-12.

[9] Quay.io. https://quay.io/, visited 2018-02-12.

[10] What is Docker. https://www.docker.com/what-docker, visited 2018-02-12.

[11] Zookeeper. https://zookeeper.apache.org/, visited 2018-02-12.

[12] ANWAR, A. *Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning*. PhD thesis, Virginia Tech, 2018.

[13] ANWAR, A., CHENG, Y., GUPTA, A., AND BUTT, A. R. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop* (2015), ACM, pp. 7–12.

[14] ANWAR, A., CHENG, Y., GUPTA, A., AND BUTT, A. R. Mos: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (2016), ACM, pp. 177–188.

[15] ANWAR, A., MOHAMED, M., TARASOV, V., LITTLEY, M., RUPPRECHT, L., CHENG, Y., ZHAO, N., SKOURTIS, D., WARKE, A. S., LUDWIG, H., HILDEBRAND, D., AND BUTT, A. R. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA, 2018), USENIX Association, pp. 265–278.

[16] ANWAR, A., SAILER, A., KOCHUT, A., SCHULZ, C. O., SEGAL, A., AND BUTT, A. R. Cost-aware cloud metering with scalable service management infrastructure. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on* (2015), IEEE, pp. 285–292.

[17] Anwar, A., Sailer, A., Kochut, A., Schulz, C. O., Segal, A., and Butt, A. R. Scalable metering for an affordable it cloud service management. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on* (2015), IEEE, pp. 207–212.

[18] Azeem, S. A., and Sharma, S. K. Study of converged infrastructure & hyper converge infrastructre as future of data centre. *International Journal of Advanced Research in Computer Science 8*, 5 (2017).

[19] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst. 14*, 4 (Nov. 1996), 311–343.

[20] Dell EMC. Improving Copy-on-Write Performance in Container Storage Drivers. https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf, visited 2018-02-12.

[21] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS* (2015).

[22] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 181–195.

[23] Kangjin, W., Yong, Y., Ying, L., Hanmei, L., and Lin, M. FID: A Faster Image Distribution System for Docker Platform. In *IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)* (Tucson AZ, 2017), IEEE, pp. 191–198.

[24] Menage, P. B. Adding generic process containers to the linux kernel. In *Proceedings of the Linux symposium* (2007), vol. 2, Citeseer, pp. 45–57.

[25] Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[26] Nathan, S., Ghosh, R., Mukherjee, T., and Narayanan, K. CoMICon: A Co-Operative Management System for Docker Container Images. In *IEEE IC2E* (Vancouver Canada, 2017), pp. 116–126.

[27] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *ACM SOSP* (1995).

[28] Rosen, R. Resource management: Linux kernel namespaces and cgroups. *Haifux, May 186* (2013).

[29] Singh, V., and Peddoju, S. K. Container-based microservice architecture for cloud applications. In *2017 International Conference on Computing, Communication and Automation (ICCCA)* (2017), IEEE, pp. 847–852.

[30] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM EuroSys* (2007), ACM, pp. 275–287.

[31] Tarasov, V., Rupprecht, L., Skourtis, D., Warke, A., Hildebrand, D., Mohamed, M., Mandagere, N., Li, W., Rangaswami, R., and Zhao, M. In search of the ideal storage configuration for Docker containers. In *IEEE AMLCS* (Tucson, AZ, 2017), IEEE.

[32] Wiel, S. P. V., and Lilja, D. J. When caches aren't enough: data prefetching techniques. *Computer 30*, 7 (Jul 1997), 23–30.

[33] Zhang, Z., Kulkarni, A., Ma, X., and Zhou, Y. Memory resource allocation for file system prefetching: From a supply chain management perspective. In *ACM EuroSys* (2009).