Towards a Scalable Docker Registry

Michael B. Littley

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Ali R. Butt, Chair
Joseph G. Tront
Yaling Yang

May 3, 2018
Blacksburg, Virginia

Keywords: Docker Registry, Scalability, Failure Recovery, Caching, Virtualization

Towards a Scalable Docker Registry

Michael B. Littley

(ABSTRACT)

Containers are an alternative to virtual machines rapidly increasing in popularity due to their minimal overhead. To help facilitate their adoption, containers use management systems with central registries to store and distribute container images. However, these registries rely on other, preexisting services to provide load balancing and storage, which limits their scalability. This thesis introduces a new registry design for Docker, the most prevalent container management system. The new design coalesces all the services into a single, highly scalable, registry. By increasing the scalability of the registry, the new design greatly decreases the distribution time for container images. This work also describes a new Docker registry benchmarking tool, the trace player, that uses real Docker registry workload traces to test the performance of new registry designs and setups.

Towards a Scalable Docker Registry

Michael B. Littley

(GENERAL AUDIENCE ABSTRACT)

Cloud services allow many different web applications to run on shared machines. The applications can be owned by a variety of customers to provide many different types of services. Because these applications are owned by different customers, they need to be isolated to ensure the users' privacy and security. Containers are one technology that can provide isolation to the applications on a single machine, and they are rapidly gaining popularity as they incur less overhead on the applications that use them. This means the applications will run faster with the same isolation guarantees as other isolation technologies. Containers also allow the cloud provider to run more applications on a single machine, letting them serve more customers. Docker is by far the most popular container management system on the market. It provides a registry service for containerized application storage and distribution. Users can store snapshots of their applications on the registry, and then use the snapshots to run multiple copies of the application on different machines. As more and more users use the registry service, the registry becomes slower, making it take longer for users to pull their applications from the registry. This will increase the start time of their application, making them harder to scale out their application to more machines to accommodate more customers of their services. This work creates a new registry design that will allow the registry to handle more users, and allow them to retrieve their applications even faster than what's currently possible. This will allow them to more rapidly scale their applications out to more machines to handle more customers. The customers, in turn, will have a better experience.

Acknowledgments

Thank you to my committee, Ali Butt, Joseph Tront, and Yaling Yang, for overseeing my work. Thank you to Ali Anwar for helping and mentoring me, and to all the other members of Virginia Tech DSSL for working with me. Thank you to SFS Cybercorps program for funding me. Thank you to IBM ResearchAlmaden researchers, Vasily Tarasov and Lukas Rupprecht for giving me feedback and suggestions on my work. Lastly, thank you to all my friends and family for providing me with all the support I need.

Contents

List of Figures viii

List of Tables xii

1 Introduction 1

1.1 Containers 2

1.2 Motivation 3

1.3 Contributions 4

2 Background 5

2.1 Docker Engine 5

2.1.1 Docker Containers 5

2.2 The docker registry 7

2.2.1 The registry API 9

2.2.2 Current Distributed Setups 11

3 Related Work 14

3.1 Work related to Docker 14

3.1.1 Docker Distribution 15

3.2 Work Related to Design . . . 17
3.2.1 Chord Distributed Hash Table . . . 18
3.2.2 FAWN . . . 18
3.2.3 Content Addressable Storage . . . 19
3.2.4 Caching and Prefetching . . . 19

4 Docker Registry Analysis **20**

4.1 The Traces . . . 21
4.2 Trace Analysis . . . 22
4.2.1 Request types . . . 23
4.2.2 Request Sizes . . . 24
4.2.3 Request Response Times . . . 25
4.2.4 Layer Popularity . . . 27
4.3 The trace player . . . 28
4.3.1 Warmup . . . 29
4.3.2 Run . . . 29
4.3.3 Simulate . . . 31
4.4 Caching Analysis . . . 31
4.5 Prefetching Analysis . . . 32

5 The New Design **36**

5.1 Proxies . . . 37
5.2 Consistent Hashing Function . . . 39
5.3 Coordination . . . 42
5.4 Storage And Caching . . . 44
5.5 Testing . . . 49
5.5.1 Caching Results . . . 50
5.5.2 Scalability Results . . . 51
5.5.3 Fault Tolerance Results . . . 52
5.5.4 Throughput and Latency Comparison . . . 53

6 Future Work **55**

7 Conclusion **57**

Bibliography **58**

List of Figures

2.1 Relationship between different parts of Docker Engine. 6

2.2 Example of manifest, name refers to the name of the repository, the tag is the identifier for the image (i.e. version), and fsLayers contains a list of digests for all the layers required to run the image. 7

2.3 Bob and Alice (left) are two users for the registry. Bob has two repositories, Redis and Wordpress, with three and two tags respectively. Alice has one repository, CentOS, with one tag, myOS. This graphic was taken from [37]. . 8

2.4 Example of a distributed registry. Users connect to a layer 7 load balancer such as NGINX, which distributes the requests to multiple registries. The registries must fetch layers from a backend object store. 12

2.5 Result experiment from sending requests to a setup with 1 NGINX load balancer and 2 registry servers. 13

4.1 Sample of an anonymized request in the traces. The "host", "http.request.remoteaddr" have been anonymized, as well as username, repository, and identifier parts of the "http.request.uri" field. 22

4.2 These graph shows the image pull vs. push ratio, manifest vs. layer pulls, and manifest vs. layer pushes. 23

4.3 CDF of the layer and manifest sizes for GET requests. 24

4.4 CDF of response time layer GET, PUT, PATCH requests and manifest GET and PUT requests. 25

4.5 CDF of the layers, manifests, and repositories by popularity. 27

4.6 The trace player consists of a master node, which reads in anonymized traces, converts them into legitimate requests, and them distributes them to the client nodes to forward on to the registry. Once the requests are finished, the clients send their results back to the master to save them. 28

4.7 Hit ratio results of the 2 layer caching simulation carried out on all the Dal traces. A hit ratio of 100% means the cache never evicted a layer. 32

4.8 Results of the prefetching simulation. Each set represents a different *watchDuration* value while each bar within the set represents the *layerDuration* value. The y-axis shows the hit ratio of layers fetched and requested over layers fetched. 34

5.1 Replica oblivious gateways require three hops from the client to the object, replica aware gateways require two hops, and replica aware clients require only one hop. 38

5.2 Percentage of memory used to store copies of layers in a replica oblivious gateway configuration. The total memory size was 110 GB. 39

5.3 Distribution of anonymized requests accross 6 nodes using consistent hashing. 42

5.4 Zookeeper Znode setup for the new design. The circles represent normal Znodes, and the hexigons represent ephemeral Znodes. Note that `/` is the root Znode, which is the parent for all other Znodes. Each registry added to the system creates an ephemeral Znode of its hostname and port number, which allows the other registries to connect to it. 44

5.5 The new design. Registries form a consistent hashing ring, and use Zookeeper to identify each other. Clients request a complete list of registries from any registry it knows of to form a copy of the ring. Clients look up layer digests in their ring to issue push and pull requests. 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 onto the next registries, or slave nodes, represented by the red lines. Clients randomly select the master or slave node to issue pull requests, represented by the dashed lines. 45

5.6 This figure illustrates both scalability and failure recovery. Registry 0xC0 went offline. Clients attempting to connect to the registry will fail, prompting them to issure registry list requests from the other registies. Once the clients have the correct ring, they may attempt to request a layer from 0x20, which has become the slave. The registry will fetch the layer from the new master and then reply to the client. For scalability, when a new node is added, it will become the master of some layers. The old master of the layers will automatically forward the layers to the new registry once registers with Zookeeper. The new registry will request the other layers it is responsible for from the layers' masters when clients request them from the new registry. 48

5.7 CDF of the layer sizes used for testing. The average layer is is 13.7 MB. . . . 49

5.8 Hit ratios on the caches accross 6 nodes. The smallest ratio is 53.6%, the largest is 63.3%, and the average hit ratio is 57.8%. 50

5.9 This graph shows the throughput and latency from increasing the number of registries while keeping the number of clients constant at 100. 51

5.10 Failure recovery test. Five registries were used to serve 10 thousand requests. Time 0 on the graph started 20 seconds into the test. At time 0, a registry was killed (failure). At time 3, Zookeeper removed the ephemeral Znode for the killed registry, allowing the registries and clients to remove the node from their rings. At time 10, the registry was brought back online. No Failure is the performance of 5 nodes without any failures. 52

5.11 Throughput and latency comparison between the new design and the current design, with Swift used as the backend object store. 54

List of Tables

4.1 Characteristics of studied traces. This table was taken from [37]. 21

Chapter 1

Introduction

Virtualization is the practice of abstracting away the computer hardware to allow multiple applications to run on a single system without interfering with each other [39]. Virtualization technologies provide the ability to isolate the applications and give them each the illusion of their own hardware. They should also have the ability to monitor the resource usage of the applications and be able to constrain their usage to prevent a single application from starving the others from a resource. Therefore, virtualized applications should prevent buggy or malicious applications from interfering or compromising other applications on a machine. The two virtualization technologies pervasive today are virtual machines and containers [54].

Virtualization technologies are widely used in industry for many different applications. They have been essential to the development cloud systems such as Amazon AWS [2], where many users share their services on the same machines and are charged based on usage. Additionally, virtual machines and containers ease application development and testing [54]. Finally, because of the isolation grantees, virtual machines and containers offer many cybersecurity applications as well. For example, many security researchers rely on virtualization to study malware [53] and set up honeypots [47].

Virtual machines such as Xen [39] and VMWare ESX [31] are the standard way to provide virtualization. They employ a hypervisor that allows multiple guest operating systems to share the same underlying hardware. The hypervisor runs on top of guest operating systems and abstracts the hardware away from each guest, allowing the guest operating system's

resources to be monitored and controlled. It also prevents the guest operating systems from learning of other guests on the same system by intercepting instructions. Hypervisors can run directly on the hardware or be run under a host operating system. The requirement of using a guest operating system incurs significant overhead, however.

1.1 Containers

Containers are a lightweight alternative to virtual machines [54]. They rely on kernel features such as cgroups and namespaces that allow all containers to run without the need of separate guest operating systems. Linux cgroups provide resource monitoring and control over groups of processes, which make up all the dependencies of a given containerized application running on the host [38]. By providing the ability to constrain the resources of the processes, cgroups can prevent buggy or malicious containers from affecting others on the system. Linux namespaces, on the other hand, provide isolation between containers by wrapping global resources into abstractions that make changes to the resource visible to other processes in the same namespace and invisible to others that are not [23]. This provides the equivalent privacy that running different guest operating systems for each application would, without the overhead of running the guest operating system.

Because containers run on the host operating system, they incur significantly less overhead than virtual machines [54]. The decreased overhead allows more containers to run on a single machine compared to what is possible with virtual machines. This has also allowed for the growth of microservices, which is the practice of implementing large applications as a set of smaller, simpler processes that communicate via lightweight mechanisms such as REST, RabbitMQ, or gRPC [17, 22]. This allows each separate microservice making up an application to be developed, tested, distributed, and versioned separately from each other,

which in turn can speed up the development and maintenance of the application as a whole.

Container management systems such as Docker [8] or CoreOS [7] enable microservices and containers to be shared in general by providing distribution systems. These systems allow users to push their containers to a container registry which can be pulled by other users for their own use.

1.2 Motivation

Containers are quickly increasing in popularity due to their decreased overhead. Containers have been steadily growing every year by 40% since 2015, making it the fastest growing, cloud enabling technology [34]. Following the current trajectory, containers will become a 2.7 billion dollar market by 2020. Much of the market is held by Docker, which is by far the most popular container management system. Docker supports more platforms than any other container management system and is paving the way for container industry [8]. Therefore, Docker is the best container management system to study. This work is focused on improving Docker by improving container startup times, which will further reduce container overhead, making it more lightweight and improving what can be done with virtualized applications.

According to Harter et al. [44], 76% of a Docker container startup time is from pulling the container from the Docker registry. As Docker increases in popularity, more users will start storing their images in the registry, and more Docker clients connect to the registry in parallel. This can cause Docker pulls to take even longer as current Docker registry implementations rely on complicated, deep stacks that are difficult to scale. Furthermore, reducing container distribution and startup time can provide new abilities like providing rapid bug patching on production systems [51]. With this in mind, the goals of this work are as follows:

1. Increase the scalability of the Docker registry.
2. Increase the maximum throughput of requests to the registry.
3. Decrease latency of the registry fetches.
4. Increase the resource efficiency of distributed registry setups.

1.3 Contributions

To increase the scalability of the Docker registry, the production registry layers were coalesced together into a single, distributed registry. A fine-grained benchmark was also developed to test the scalability, throughput, and reliability of both the current registry setups and the new distributed registry. An explanation of the Docker system is detailed in Chapter 2, followed by a review of the related work in Chapter 3. Next an analysis of a current popular Docker registry distribution is discussed in Chapter 4, then the new system is detailed in Chapter 5. Lastly, there is a discussion of future work and conclusion in Chapters 6 and 7.

Chapter 2

Background

The Docker container management system is an open source project written in Go [14] made and maintained by Docker [8]. The system consists of a Docker engine running on a client machine and a remote registry for storing and distributing Docker containers. The following sections explain what Docker containers are and the role of the Docker engine and registry. The following description of Docker components comes from the documentation.

2.1 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 CLI allows users to interact with the daemon with either commands or scripting. The daemon is responsible for creating and managing containers as well as fetching any necessary container components from the Docker registry. Additionally, the daemon provides network and file system functionality for its containers. The components of the Docker system are illustrated in 2.1.

2.1.1 Docker Containers

A Docker container is a containerized application and is stored as a Docker image. An image consists of all the executables, dependencies, and configuration files required to run

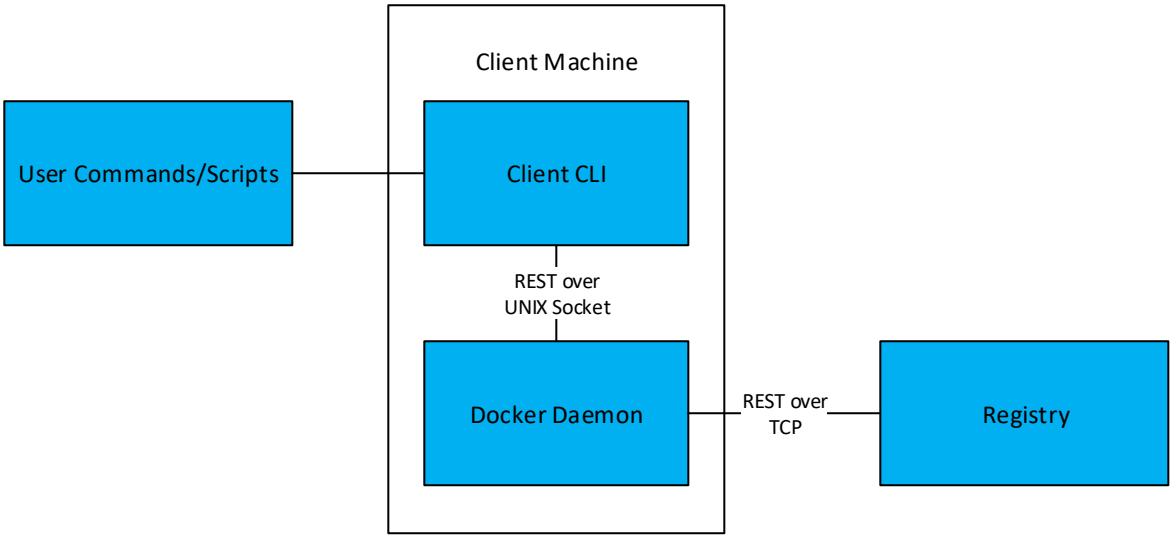


Figure 2.1: Relationship between different parts of Docker Engine.

the application 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 meta data, are contained in a JSON file called a manifest, shown in Figure 2.2. To increase the security of Docker, image manifests can be signed, providing authenticity and integrity to Docker images. Docker images are also stateless; any change to an image must result in a new image. Lastly, Docker images are usually built on top of other images. For example, an image for NGINX will require dependencies from an Ubuntu image. When creating a container image for NGINX, the Ubuntu layers must be included.

Starting a container consists of several steps. For example, if the user enters the command "docker run -i -t Ubuntu /bin/bash" into the CLI client, the request is forwarded to the daemon running on the local machine. If a registry is set up with the daemon, the daemon will request any layers from the Ubuntu image not held locally from the registry. The Docker

```
{
    "name": <name>,
    "tag": <tag>,
    "fsLayers": [
        {
            "blobSum": <digest>
        },
        ...
    ],
    "history": <v1 images>,
    "signature": <JWS>
}
```

Figure 2.2: Example of manifest, name refers to the name of the repository, the tag is the identifier for the image (i.e. version), and fsLayers contains a list of digests for all the layers required to run the image.

daemon will then create a container from the image layers by setting the proper namespaces and cgroup configurations as needed. Next a read-write file system will be allocated for the container and a network interface will be defined for it. Finally, the daemon will start the container.

2.2 The docker registry

The Docker registry is a stateless, centralized service which provides image storage and distribution. Users can store multiple versions of their images in repositories by denoting each version with a special tag. For example in Figure 2.3, the user Bob has a repository Redis with three tags: v2.8, v2.6, and latest. Provided that Bob has made his repository public, any user can pull any of the three images specified by their tags.

The registry provides both a simple authorization service as well as a delegated one. Both services rely on TLS for server side authentication. The simple authorization service uses a

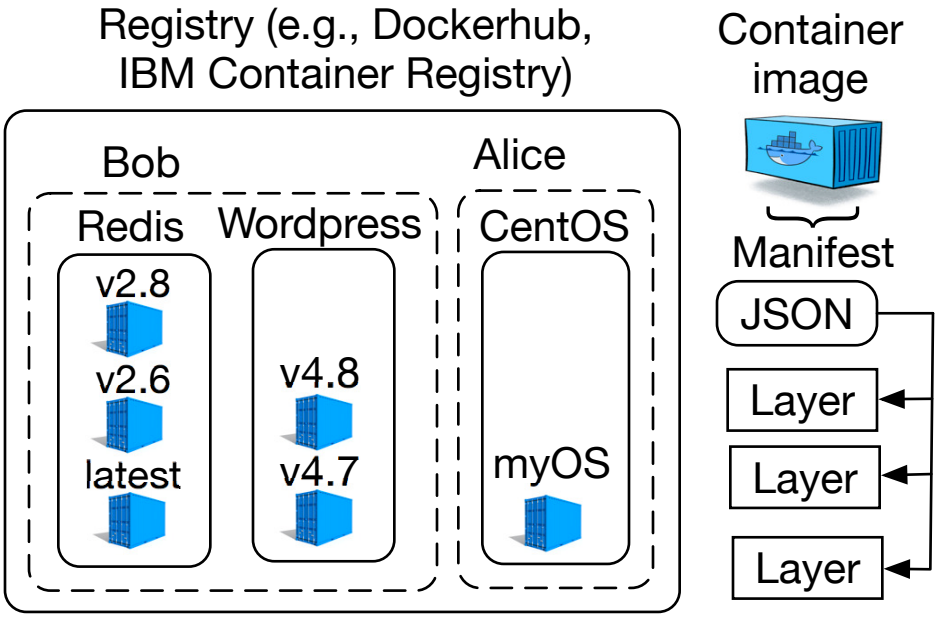


Figure 2.3: Bob and Alice (left) are two users for the registry. Bob has two repositories, Redis and Wordpress, with three and two tags respectively. Alice has one repository, CentOS, with one tag, myOS. This graphic was taken from [37].

password based system handled by the registry server directly. The delegated service allows the registry to redirect users to a dedicated, trusted authorization service for authentication.

For storage, the registry supports the following drivers:

- in memory
- filesystem
- Amazon Simple Storage Service [27]
- Microsoft Azure Blob Storage [3]
- Openstack Swift [28]
- Aliyun OSS [24]

- Google Cloud Storage [12]

The in memory driver is reserved for small, registry test benches, while the filesystem driver is meant for single registry instances. All the other drivers use backend objects stores.

2.2.1 The registry API

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

Pull Requests

Image pulls are when the daemon requests an image from the registry to run or store locally. To pull an image, the manifest is requested first using the following request:

```
GET https://<registry URL>/v2/<username>/<repository name>/manifests/<image tag>
```

The username refers to the owner of the repository where the image is kept. The registry will respond with the error code 404 if the image does not exist in the registry. If the images exist, then the daemon issues requests for all the layers contained in the manifest that it does not have locally. The daemon uses the following request to fetch layers:

```
GET https://<registry URL>/v2/<username>/<repository name>/blobs/<digest>
```

The registry checks the permission on the repository to determine whether the user is authorized to pull the layer or manifest. Once a layer is fetched, a hash of its contents is compared to its digest to ensure its integrity. The image pull is complete once all the layers are fetched. It is important to note that a Docker pull is attempted anytime the Daemon receives a run command for an image not locally available.

Push requests

Push requests occur when new container images are created and added to the registry. The sequence of requests required to push an image happens in the reverse order as a push request, pushing each layer individually and finishing by pushing the manifest. To push a layer the daemon first issues the following head request to check for the layer's existence:

```
HEAD https://<registry URL>/v2/<username>/<repository name>/blobs/<digest>
```

If the layer does not exist in the registry, then the upload process begins with the following request:

```
POST https://<registry URL>/v2/<username>/<repository name>/blobs/uploads
```

If successful, the registry will respond with a location to begin uploading. This location contains a <uuid>for the registry and daemon to keep track of the upload. The daemon can then begin the uploading process. The registry supports both chunked uploads and monolithic uploads. To upload the layer monolithically, the layer is uploaded with the following request:

```
PUT https://<registry URL>/v2/<username>/<repository name>/blobs/uploads/<uuid>
&digest=<digest>
Content-Length: <size of layer>

<layer content>
```

To upload the layer in chunks, the daemon uses this request:

```
PATCH https://<registry URL>/v2/<username>/<repository name>/blobs/uploads/<uuid>
Content-Length: <size of chunk>
Content-Range: <start of chunk>-<end of chunk>

<chunk content>

PUT https://<registry URL>/v2/<username>/<repository name>/blobs/uploads/<uuid>
&digest=<digest>
Content-Length: <size of last chunk>
Content-Range: <start of chunk>-<size of layer>

<last chunk content>
```

The PUT request signals the end of the content. When received, the registry compares the hash of the layers content with the digest provided by the client. If the hashes match then the upload is successful and the registry moves the layer into content addressable storage. Using content addressable storage provides the registry with deduplication on the layers, regardless of user or repository.

2.2.2 Current Distributed Setups

The Docker registry is open sourced so users, companies, or organizations can set up their own registries for their own networks. Alternatively, users can opt for using an established registry service for convenience. Some examples of registries are Docker Hub [9], IBM Cloud container registry [18], Quay.io [25], Google Container Registry [15], and Artifactory [1].

These registries are responsible for storing millions of images and serving them thousands of

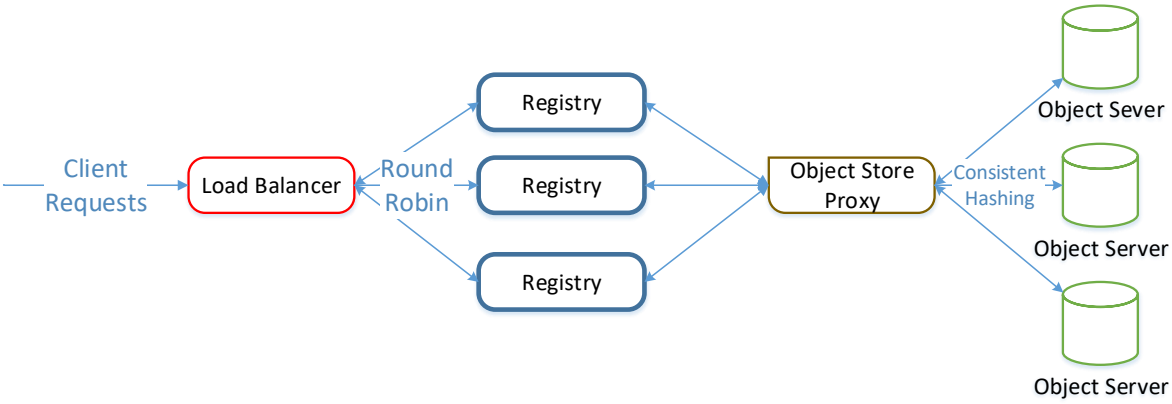


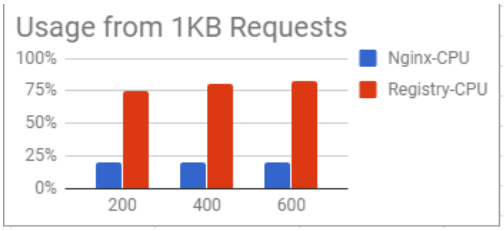
Figure 2.4: Example of a distributed registry. Users connect to a layer 7 load balancer such as NGINX, which distributes the requests to multiple registries. The registries must fetch layers from a backend object store.

clients simultaneously. In order to accommodate these demands, companies must host their registry services across multiple nodes, resulting in a complicated stack. A typical example of what is required to host a production level registry is shown in Figure 2.4. To serve many requests concurrently, different load balancers have to be deployed in front of several instances of the registry, and the registries have to depend on a back-end object store service such as Swift for the image layers and manifests. These object stores often require their own load balancers and proxies to serve the registries' manifest and layer requests.

The complexity of the stack means a single request from a client results in many hops from the load balancer to the storage node containing the manifest or layer requested. The registry does provide distributed caching options for manifests such as Redis [26] or Memcached [21], however, this option is not available for layers. Without caching for layers, the latency of layer pulls can easily become the bottleneck when starting Docker containers.

The complexity of the stack in current setups also makes scaling difficult, as each layer of the stack can become a bottleneck. To illustrate this, an experiment was run using the trace

(a) Results of sending 1 MB requests



(b) Results from sending 1 KB requests

Figure 2.5: Result experiment from sending requests to a setup with 1 NGINX load balancer and 2 registry servers.

player from Anwar et al. [37] explained in Section 4.3 to send requests to a small setup of two registry servers behind one NGINX load balancer. The trace player machines first sent requests for 1MB layers, then sent requests for 1KB layers to gauge the resource usage of the NGINX and registry servers. All three machines had 8 core CPUs, 16GB of RAM, and 10Gbps network cards. The results of the experiment are shown in Figure 2.5. When sending 1MB requests, the network of the load balancer limited the throughput of the setup, but when sending 1KB requests, the registries' CPUs were the most utilized resource. This shows how the maintainers of a production registry have to identify one of the potentially many bottlenecks while attempting to scale their registry.

Chapter 3

Related Work

This chapter provides an overview of the current work related to Docker, as well as works related to other aspects of the new design. Section 3.1 is based on and builds on the discussion from Anwar et al. [37]. Section 3.2 focuses on works related to the new Registry design.

3.1 Work related to Docker

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 on the Docker daemon. Many images contain layers that are either identical, or contain identical files within them, and Docker relies heavily on Copy-on-Write (COW) storage techniques to allow many containers to be run simultaneously with minimal data churn. Tarasov et al. [56] provides a study on the impact of different storage drivers on different types Docker workloads. Alternatively, Dell EMC [42] provides a layer between the driver and the Daemon which improves performance for dense container workloads. However, this thesis is focused on improving the Docker registry so works like these can be easily integrated together.

3.1.1 Docker Distribution

These works are focused on improving Docker's performance through changing how containers are distributed and are thus more related to this thesis.

Slacker

Harter et al. [44] introduced Slacker, which improves container start-time performance and distribution by incorporating a network file system (NFS) server into both the Docker daemon and registry. All container data is stored in the NFS server which is shared between all the worker nodes the registry. This causes Docker pushes and pulls to become much faster as newly created layers become known to all nodes as they are created, eliminating the need to actually push and pull them. Furthermore, only the layers necessary for starting containers are fetched from the NFS server at the startup for the container, and all other layers are only fetched when needed by the container.

This design has several drawbacks. 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 start-times times. However, with Slacker, the necessary layers must be fetched from the NFS every time the container starts. The advantage of slacker occurs in situations where images are updated frequently or are long-running. Slacker also tightly couples the worker nodes with the registry. Docker clients also need to maintain a constant connection with the NFS server to fetch additional layers as they need them which can present a challenge to services like Docker Hub.

Harter et al. [44] also introduced HelloBench [16], a benchmarking tool for Docker registry pushes and pulls. Their benchmark consists of 57 containers which require minimal configuration and do not depend on other containers. These images provide a variety of containers,

each with varying numbers of layers, and can be run with the standard Docker commands for pushing and pulling. Unlike the Trace Player discussed in 4.3, HelloBench analyzes the client-side performance of image pulls and pushes, while the Trace Player focuses on the registry performance. Furthermore, HelloBench's dataset is small and is not influenced by real-world work flows.

CoMICon

Nathan et al. [49] proposes a peer to peer (p2p) network, CoMiCon, for Docker clients to share Docker images. 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. The system adds an image and node metadata manager which stores information about all the nodes in the system, all the images, and the locations of all the layers on the nodes. The metadata manager is used by a master registry to dictate which registry slave should store which layer. CoMiCon also adds a new pull command to the Docker Daemon which allows the clients to pull images from the registry slaves. This command works by first consulting the master for the locations of each layer in the image and then pulling from the correct registry slave.

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 lacks backwards compatibility with the current Docker Registry. Finally, using a p2p network to exchange layers offloads the work of the registry onto the Docker client nodes. It's the goal of this work to increase the scalability of the Docker registry without increasing the burden of Docker clients. However, running a p2p system on the clients along side the new design could easily be achieved if users would be willing to do so.

FID

Kangjin et al. [45] integrates the Docker daemon and registry with BitTorrent [5]. 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 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.

Dragonfly

Dragonfly [10] is a p2p based file distribution system recently open sourced by Alibaba, which is compatible with Docker. It advertises extreme increase in bandwidth and decrease in registry latency. However, to my knowledge, this system has not been implemented or studied anywhere outside Alibaba. Because it is p2p application, the same considerations from the other p2p works apply as well.

3.2 Work Related to Design

These works relate to aspects of the new registry design described in Chapter 5.

3.2.1 Chord Distributed Hash Table

Chord [55] is a p2p distributed hash table which leverages consistent hashing [46] to distribute data among a set of nodes. In consistent hashing, a standard hash function such as SHA1 is used to hash both keys and node identifiers, forming a mathematical ring of (mod 2^m) where m is the length of the hash function. The node identifiers' hashes are placed in the ring and the keys' hashes are used to find the closest node identifier hash equal to or greater than the key hash. The value associated with the key can then be stored or retrieved from the node corresponding to the node identifier. The length of the hash m should be selected to minimize the probability of collisions in the node identifiers, and several node identifiers should be assigned to a single node to ensure an even distribution of keys.

Each Chord node maintains a finger-table of a certain number of successor nodes. Whenever a node leaves or enters the ring, the finger-tables of all other nodes must be updated. Nodes replicate their data by forwarding them onto successor nodes so if they leave the data will not be lost. Furthermore, whenever a node is added, some preexisting data will be reassigned to the node, so the node which was originally responsible for data must populate the new node with the old data.

The new design uses similar principals for storing layers using consistent hashing. However, unlike Chord, the new design relies on a meta-data server for maintaining the ring rather than finger tables. This is because Chord is focused on a decentralized design while the new design is centralized.

3.2.2 FAWN

Andersen et al. [36] implements a similar system to the new design. However, their focus is on providing speed and energy efficiency with cheaper hardware. While FAWN is designed

for key-value stores, our work applies their design principals to the Docker Registry. Another key difference is FAWN's reliance on a proxy, which the new design eliminates.

3.2.3 Content Addressable Storage

Because layers are read-only, content addressable storage (CAS) was a natural choice for Docker for storage. There have been numerous works and technologies which offer CAS services [13, 19, 20, 52, 57]. The Docker registry leverages its own CAS services for layers and the new design contributes to this by distributing the layers among registry nodes.

3.2.4 Caching and Prefetching

Both caching and prefetching are both tried and true methods for improving performance [40, 41, 43, 48, 50, 58, 59]. To my knowledge, the new design is the first work to incorporate layer caching into the registry design. A prefetching algorithm for Docker was also created for [37] is discussed in Chapter 4.

Chapter 4

Docker Registry Analysis

This chapter is on the work presented in Anwar et al. [37]. In order to further improve the Docker registry, the collection and analysis on long-span, production level workloads was carried out on five IBM data centers. These data centers are located in Dallas, Texas; Montreal, Canada; London, England; Frankfurt, Germany; and Sydney, Australia [18]. The workload traces collected cover seven availability zones within the data centers. There are four production level zones, two test zones, and one zone used internally by IBM.

The production level availability zones are located in Dallas (Dal), London (Lon), Frankfurt (Fra), and Sydney (Syd), and serve customers ranging from individuals, small to medium businesses, all the way up to large enterprises and government institutions. The two testing availability zones, prestaging (Prs) and development (Dev), are located in Montreal. Finally, the staging (Stg) availability zone is used internally at IBM, and is located in Dallas. Each production zone and Stg has six registries behind NGINX load balancers, while the testing zones use three registries each. All zones except for the testing zones use Swift as their backend object store, and is located in Dallas. The Dallas production zone is the oldest, while the Sydney zone is the newest, starting during the collection period. The collection period lasted 75 days covering all the zones, and consists of over 38 million requests and 181.3 TB of data transferred.

Aavailablity Zone	Dal	Lon	Fra	Syd	Stg	Prs	Dev	TOTAL
Duration (Days)	75	75	75	65	65	65	55	475
Trace data (GB)	115	40	17	5	25	4	2	208
Filtered (GB)	12	4	2	0.5	3.2	0.5	0.2	22.4
Requests (millions)	20.85	7.55	1.80	1.03	5.90	0.75	0.34	38.22
Data ingress (TB)	5.50	1.70	0.40	0.29	2.41	0.23	0.01	10.54
Data egress (TB)	107.5	25.0	3.30	1.87	29.2	2.45	1.44	170.76
Images pushed (1,000)	356	331	90	105	327	65	15	1289
Images pulled (1,000)	5,000	2,200	950	360	1,560	140	15	10280
Up since (mm/yy)	06/15	10/15	04/16	04/16	-	-	-	-

Table 4.1: Characteristics of studied traces. This table was taken from [37].

4.1 The Traces

The requests were captured from the registry, filtered, anonymized, and stored as JSONs. Table 4.1 shows the basic characteristics of the data collected. Dal was by far the most popular availability zone, accounting for more than half of the data collected. The filtered and anonymized traces are 22 GB total, and are available at [30]. The anonymized traces are made up of layer GET, PUT, PATCH, and HEAD requests and manifest GET, PUT, and HEAD requests, which all correspond to image pushes and pulls. A sample of the anonymized trace is shown in 4.1.

Each trace contains 10 fields: `host`, `http.request.duration`, `http.request.method`, `http.request.remoteaddr`, `http.request.uri`, `http.request.useragent`, `http.response.status`, `http.response.written`, `id`, and `timestamp`. The `host` field contains the anonymized address of the registry server that handled the request. The `http.request.remoteaddr` contains the anonymized address of the Docker client which made the request. The `http.response.written` and `http.request.duration` contains the size of either the layer or manifest being pushed or pulled and the length of time in seconds that the request took. The `http.request.uri` contains the anonymized username, repository, and identifier for the request, as well as whether

```
{
  "host": "786b3803",
  "http.request.duration": 0.913932322,
  "http.request.method": "GET",
  "http.request.remoteaddr": "f95a1151",
  "http.request.uri": "v2/1a5446cf/9b1aa2f6/blobs/6f7bafa4",
  "http.request.useragent": "docker/1.12.1 go/go1.6.3 git-commit/23
     cf638 kernel/3.16.0-67-generic os/linux arch/amd64 UpstreamClient(
     python-requests/2.18.1)",
  "http.response.status": 200,
  "http.response.written": 4656,
  "id": "b091ccbf29",
  "timestamp": "2017-08-04T06:00:00.525Z"
},
```

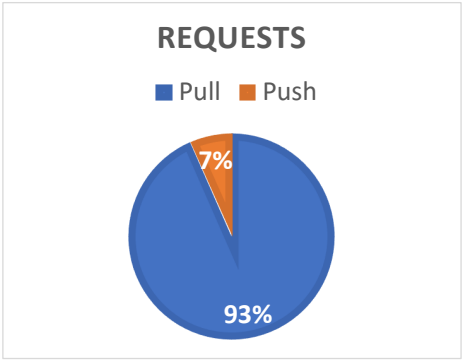

Figure 4.1: Sample of an anonymized request in the traces. The "host", "http.request.remoteaddr" have been anonymized, as well as username, repository, and identifier parts of the "http.request.uri" field.

the request is for a layer or manifest. Lastly, the `timestamp` field contains the time when the request began.

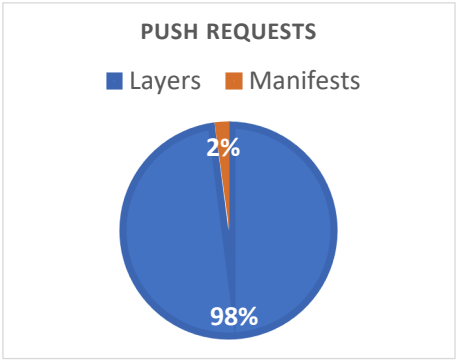
4.2 Trace Analysis

The anonymized traces from all availability zones were analyzed to determine possible registry optimizations. The analysis was focused on uncovering the general registry workload, including request types, sizes, and response times. It also focused on investigating the spatial and temporal properties of requests and whether any correlations between them could be discovered. Because Dal is the oldest and most popular registry, this section will only focus on those traces for request types, sizes, response times, and popularity to provide justification for the registry design changes. Anwar et al. [37] gives a more extensive analysis over all availability zones.

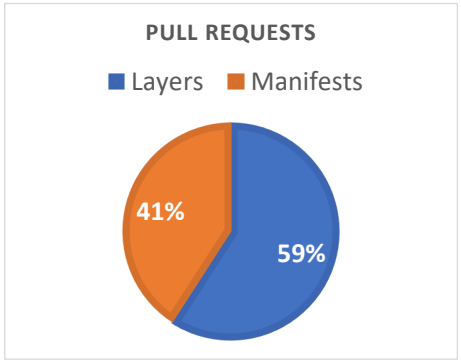
4.2.1 Request types



(a) Image Push vs. Pull



(b) Push Layers vs. Manifests



(c) Pull Layers vs. Manifests

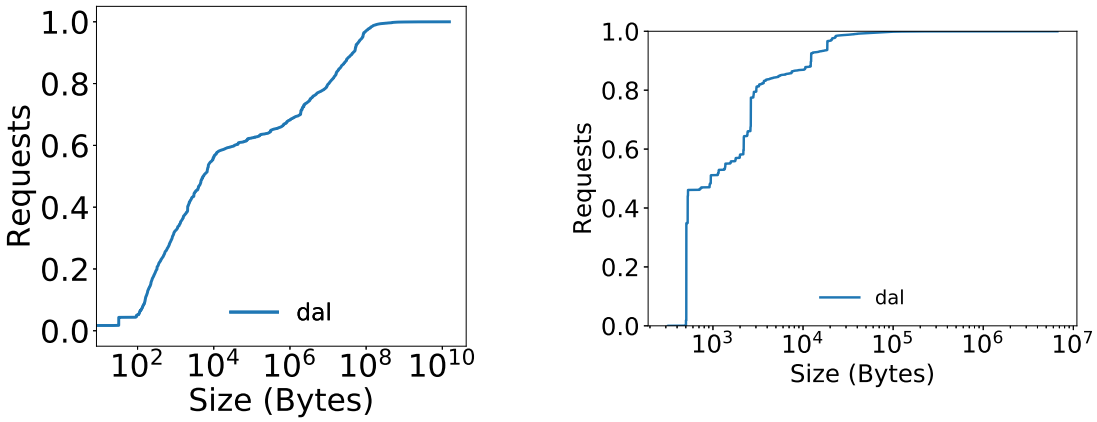
Figure 4.2: These graph shows the image pull vs. push ratio, manifest vs. layer pulls, and manifest vs. layer pushes.

Because image manifests must be fetched during image pulls, and manifests must be pushed during image pushes, the image push/pull ratio can easily be extracted from the traces, shown in figure 4.2. The Docker registry is a very read heavy service, as 93% of the requests are pull requests. One reason why pull requests are much more prevalent than push requests is because the Docker client issues a pull request automatically when a container is started. This also explains why manifest requests make up 40% of the pull requests, as clients always issue requests for manifests during pulls, but only issue requests for layers they do not have

locally. Finally, the number of layer push requests dominates manifests because images often consist of many layers and a single manifest. It's important to note however, that although all layers are pushed during an image push, generally only one layer has to actually be sent, as most layers for the image will already exist in the registry. The graph shows only attempted layer pushes, given by the amount of HEAD requests.

4.2.2 Request Sizes

The sizes of layer and manifest GET requests are shown in 4.3. GET requests were selected as they most accurately represent the actual sizes of the layers and manifests. As expected, manifests are small, with the majority being under 10 KB. About 65% of the layers fetched are under 1MB, 80% are less than 10 MB, and more than 99% of the layers are less than 1GB, with a total layer average size of 13 MB. This also makes sense as layers are created with stacking in mind, so most layers will contain a only couple dozens of files. This also proves that layers are small enough for caching.

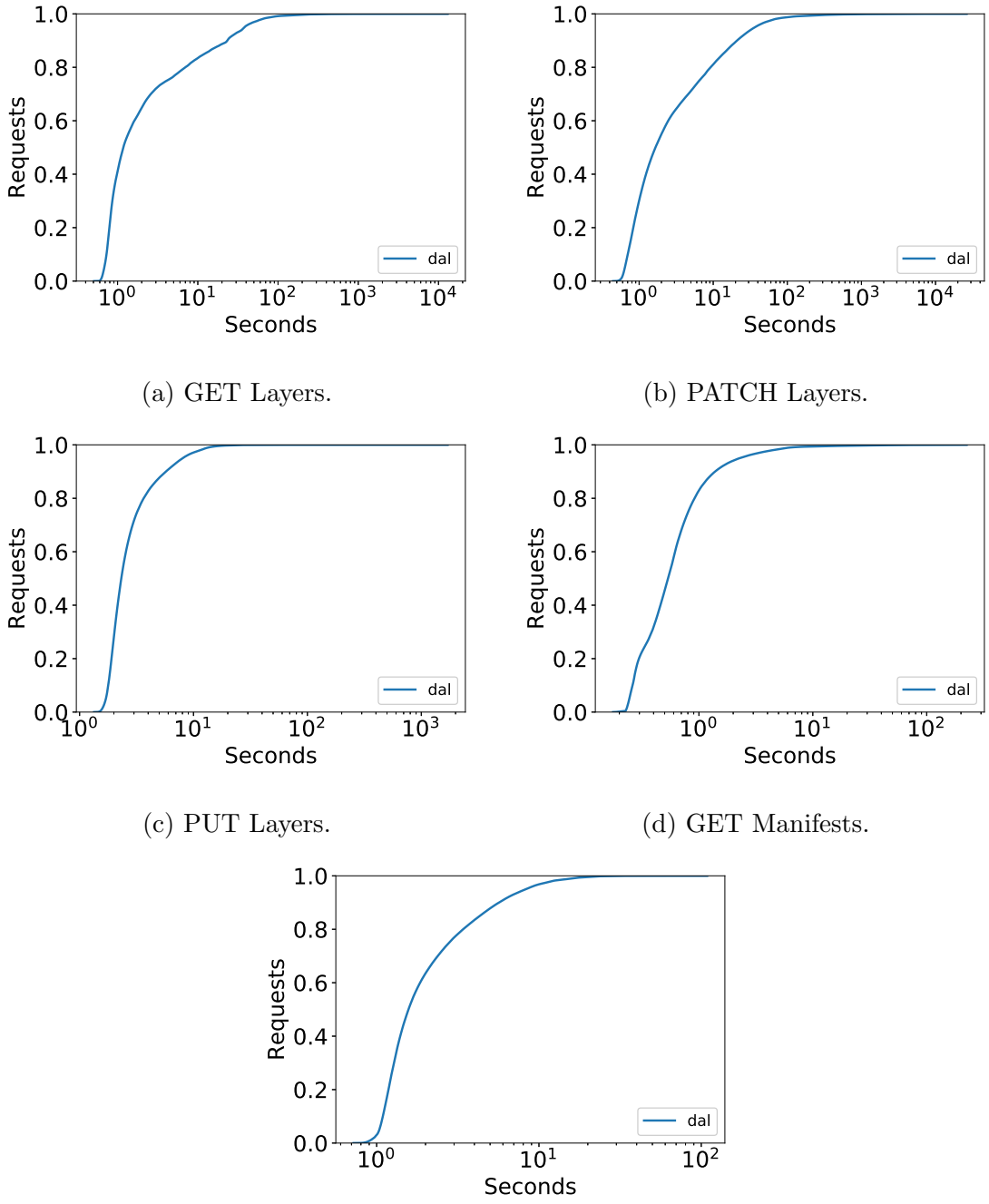


(a) Layer Sizes

(b) Manifest Sizes

Figure 4.3: CDF of the layer and manifest sizes for GET requests.

4.2.3 Request Response Times



(a) GET Layers.

(b) PATCH Layers.

(c) PUT Layers.

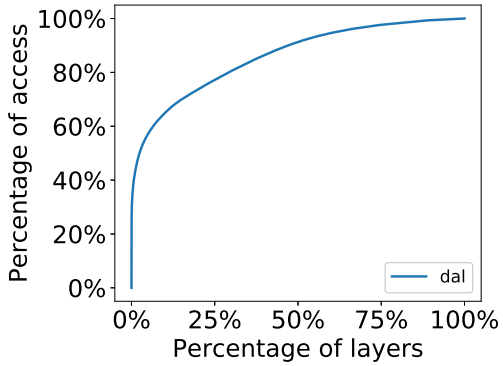
(d) GET Manifests.

(e) PUT Manifests.

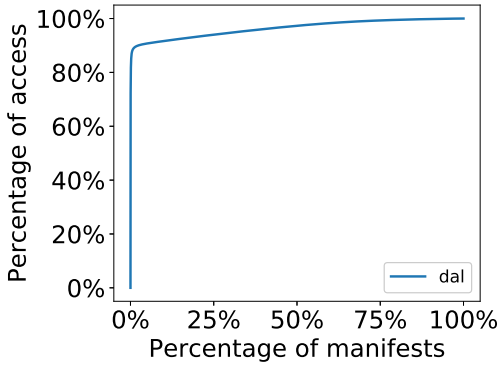
Figure 4.4: CDF of response time layer GET, PUT, PATCH requests and manifest GET and PUT requests.

Figure 4.4 shows the average response times for GET, PATCH, and PUT requests for both layers and manifests. Layer HEAD and POST requests are not shown as they do not carry any payloads. The vast majority of the response times for manifests are under a second, which is expected given the typical manifest sizes. Twenty-five percent of the layer GET requests take ten seconds or longer. Finally, Layer PATCH requests take longer than PUT requests as clients typically upload most of the layer data in the PATCH requests. This illustrates the need to focus on improving the response time for layer requests while improving registry performance.

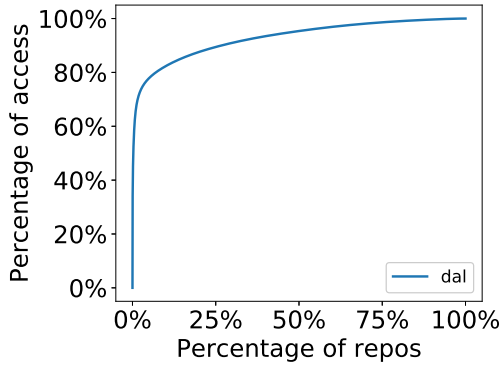
4.2.4 Layer Popularity



(a) Layer Popularity



(b) Manifest Popularity



(c) Repository Popularity

Figure 4.5: CDF of the layers, manifests, and repositories by popularity.

Figure 4.5 demonstrates heavy skew in layer, manifest, and repository access for the registry. About 1% of the layers account for 42% of all layer accesses. Manifests, which represent images as a whole, show even more skew. The single most popular image accounts for 20% of the requests. Repositories also reflect images, however manifests are finer grained as they represent versions of an image unlike. This further emphasizes the potential benefit of caching layers in memory or entire images, rather than just manifests.

4.3 The trace player

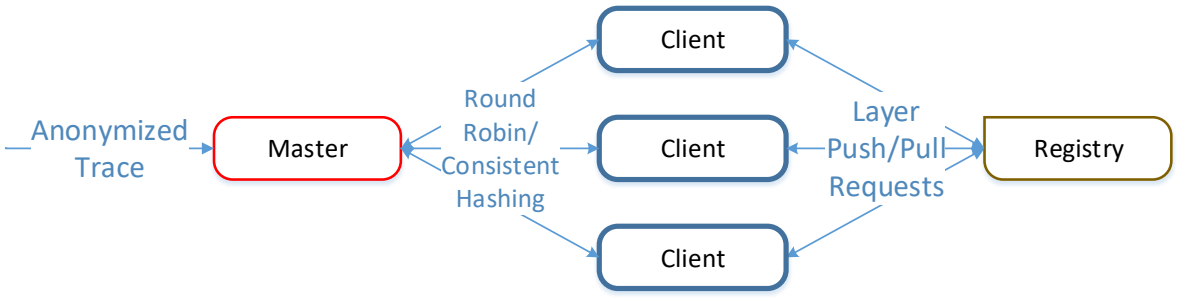


Figure 4.6: The trace player consists of a master node, which reads in anonymized traces, converts them into legitimate requests, and them distributes them to the client nodes to forward on to the registry. Once the requests are finished, the clients send their results back to the master to save them.

In order to further study the docker registry, a distributed trace player [29] was developed to both benchmark registry set ups and perform simulations using the anonymized traces. Figure 4.6 shows the structure of the trace player. It consists of a master node that reads in anonymized traces, organizes, and then distributes them to client nodes that issue layer push and pull requests to the registry. The system consists of three main modes, with additional options and features for each mode. The additional options, along with the names of the trace files to use, are contained in a configuration file that must be specified at runtime. Both the master and client are implemented in Python, using the DXF [11] library to issue Layer requests.

The trace player's modes of operation are Warmup, Run, and Simulate. Of these modes, only Run mode requires the client nodes, while all others are carried out by the master. For each mode, the master must read in the traces. The traces are filtered so that only layer GET and PUT requests are read. The `timestamp`, `http.request.method`, `http.request.uri`,

`http.response.written`, `http.request.duration`, and `http.request.remoteaddr` fields are extracted for each mode.

4.3.1 Warmup

In Warmup mode, the master creates real layers to map to the anonymized layers and pushes them to the specified registry. By default, the layers created are null strings equal to the length of `http.response.written`, however, if the user specifies randomness, then the last eight bytes of the string are randomized. Without specifying randomness, the amount of layers pushed to the registry will potentially decrease as different layers of the same length will be treated as the same. The mapping between the anonymized `http.request.uri` field and the SHA256 hash of the generated file is stored as a JSON file for Run mode. Lastly, only layers for GET requests are generated, and users can specify how many processes the master should use to generate and forward files to the registry.

4.3.2 Run

Run is the main mode of operation, in which the master distributes layer GET and PUT requests to the Docker registry. When the trace player is started in this mode, the master reads in the anonymized trace files and converts the GET layer requests to the layers generated during Warmup mode. Depending on the configuration, the master can either distribute requests to the client nodes in a round robin fashion, or by consistent hashing the `http.request.remoteaddr` field to ensure the consistency between the anonymized Docker clients and the client nodes sending the requests. The master organizes all the requests for each client node together with a return port number, thread count, control flags, and the list of registries to test into a JSON that is forwarded on to each client. Finally, the master

opens a server at the specified return port number and listens for the clients' responses.

The client runs a HTTP server using the Python Bottle library [6]. It waits for incoming requests from the master node. When a request is received, The client extracts the number of threads to use to send requests, the layers to send, and the control flags from the JSON. Because the client is implemented in python, the client creates processes equal to the number of threads specified by the master and then distributes the requests out to them. The two control flags concern creating files for push requests and whether the client needs to send the requests at the same relative time that they were forwarded in the traces. For push requests, the client follows the same procedure as Warmup mode, creating random files if the control flag is set, and null strings otherwise. To ensure requests can still follow the same timing as they do in the traces, the client distributes its requests based on the requests' start times and duration, illustrated in Algorithm 1, where the start time is the number of seconds between when the first request was issued and when the current request was issued in the traces, and the duration is the value from the `http.request.duration` field.

Algorithm 1: Layer Scheduling Algorithm.

Input: *Requests*: The list of layers to pull or push. Each layer in the list contains a $start_time$ and a *duration*. *Threads*: Struture of lists for requests for each process to send. Each list also contains a counter, *time* for scheduling.

```
1 begin
2     for request in Requests do
3         process_list ← min_time(Threads)
4         if process_list.time < request.start_time + request.duration then
5             process_list.time = request.start_time + request.duration
6         else
7             process_list.time = process_list.time + request.duration
8         process_list.append(request)
```

If the client needs to ensure the requests are sent at the same time, it will wait for the required amount of time before issuing the request, otherwise it will issue the requests as fast as possible. It will store whether the request was issued on time, whether an error occurred while issuing the request, the layer digest and size, the duration of the request,

and the time the request was started. Once all the requests have been issued, the client will send all the information it collected back to the master node. Once the master has all the information from all the clients, it will store them as traces for comparison and evaluation. It will also calculate the throughput, average latency, and the number of failed requests automatically.

4.3.3 Simulate

Simulate is the simplest mode in the trace player, which allows the trace player to perform analysis on the anonymized traces offline. It works by reading in the trace and then handing it off to a user defined function along with any other additional arguments for the user's needs. Both prefetching and caching analysis were carried out in this mode.

4.4 Caching Analysis

The effectiveness of caching was studied by simulating a situation where the registry nodes use a two level cache using memory and an SSD. Smaller layers (under 100 MB) are cached in main memory, while larger layers are only cached on the SSD. Also, when layers are evicted from memory, they are moved to the SSD, and when layers are evicted from the SSD, they are removed. Cache misses cause the registry to fetch the layer from the backend object store Least recently used (LRU) is the eviction policy for both levels of cache. Because layers are content addressable, there's no need to account for consistency, as updates to layers are treated as new layers.

For the simulation, 2% to 10% of the ingress of the registry was used for the memory size. The sizes of the SSDs that were simulated were 10, 15, and 20 times the memory size. For

Dal, this means the memory tested was between 110 GB to 550 GB, and SSD sizes of 1.1 TB to 11 TB. The results of the test are shown in Figure 4.7. The hit ratio of just memory varied between 46% to 61%, while the larger second layer cache sizes did not evict any layers. Furthermore, using the trace player on a real registry deployment with caching added showed that for layers under 1 MB, fetching from an SSD was 10 times faster than fetching from swift and fetching from memory was 10 times faster than fetching from the SSD. This shows that even simple caching policies can greatly increase the performance of the registry.

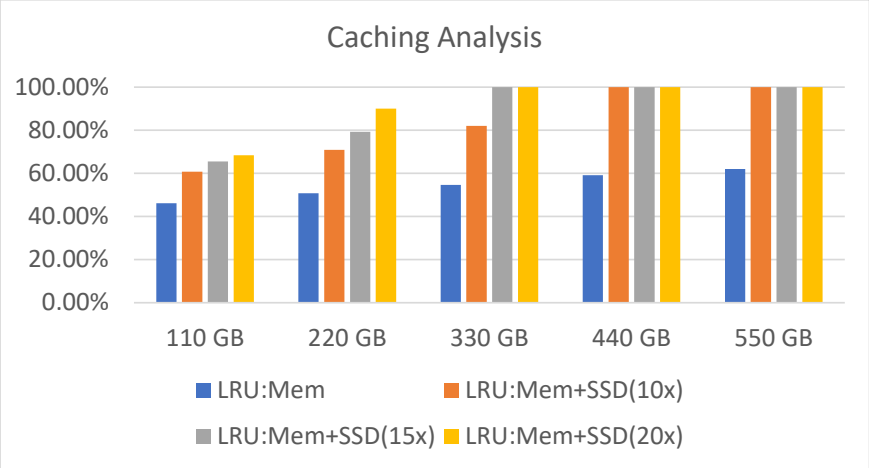


Figure 4.7: Hit ratio results of the 2 layer caching simulation carried out on all the Dal traces. A hit ratio of 100% means the cache never evicted a layer.

4.5 Prefetching Analysis

Another simulation was conducted to determine the effectiveness of prefetching layers based on the following observations from the traces:

1. On an image pull, the Docker client will issue a GET request for the manifest, and then issue GET requests for any layers that the client does not possess.

2. 80% of GET manifest requests are not followed by a GET layer request from the client within 1 minute from the GET manifest request.

3. When considering manifest requests that are issued after a layer PUT request to the repository, 75% have corresponding GET layer requests from the repository from the same clients.

From these observations, a prefetching algorithm as described in Algorithm 2 was developed and simulated. When a PUT layer request is received by the registry, its digest, repository, client's address, and arrival time are added to a lookup table. The table holds entries for a certain threshold, *watchDuration*, at which point they are evicted. GET manifest requests from new clients that arrive before *watchDuration* will cause the registry to fetch the layer in the lookup table from the object store and hold it in memory for another specified amount of time, *layerDuration*. The client address from the GET manifest request will also be added to the lookup table, preventing that client from initiating another fetch. GET manifest requests from new clients will reset the timer on layers held in memory, and the layers are evicted once *layerDuration* has been reached.

The algorithm was tested on all the traces, over *watchDuration* and *layerDuration* values of 1 hour, 12 hours, and 1 day. The hit ratio of the algorithm is the number of times a layer that was fetched has been requested divided by the total number of layers that are fetched. Therefore, a hit ratio of 1.0, would mean that every layer fetched was requested by a client exactly once. From the graph, the smallest metric, of 1 hour for both *watchDuration* and *layerDuration*, had a hit ratio of 0.92 while the largest of 1 day resulted a hit ratio of 5.4. For all simulations, the memory usage never exceeded 9.3 GB.

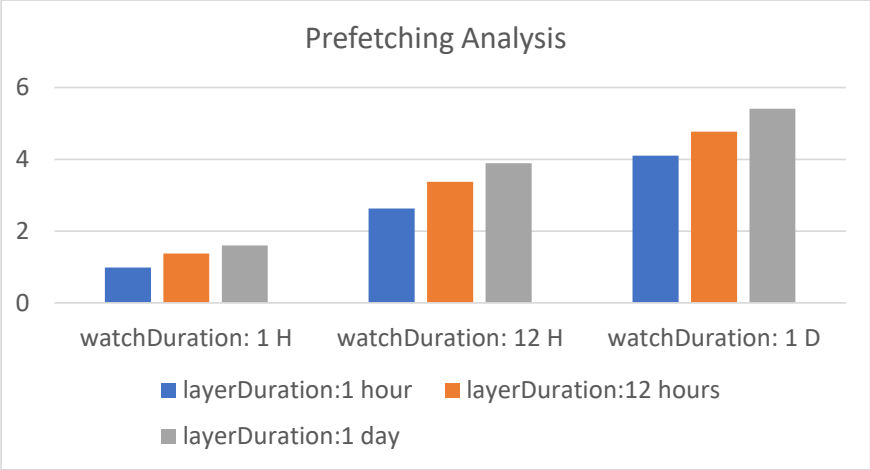


Figure 4.8: Results of the prefetching simulation. Each set represents a different *watchDuration* value while each bar within the set represents the *layerDuration* value. The y-axis shows the hit ratio of layers fetched and requested over layers fetched.

Algorithm 2: Layers Prefetching Algorithm.

Input: *watchDuration*: Duration to watch for layers to be prefetched, *layerDuration*: Duration to keep prefetched layer in memory, $RL[]$: Lookup table for layers to prefetch, $PL[]$: Layer prefetching cache

```
1  begin
2      while true do
3          r ← request received
4          if r = PUTLayer then
5              RL[] ← r.repo
6              RL[r.repo] ← (r.client, r.layer)
7              RL[r.repo] ← set watchTimer
8          else if r = GETmanifest then
9              if r.client in RL[r.repo] then
10                 continue
11             else if r.repo in PL[] then
12                 PL[r.repo] ← reset LayerTimer
13                 continue
14             else if r.repo in RL[] then
15                 PL[r.repo] ← prefetch_layer(RL[r.repo])
16                 PL[r.repo] ← set LayerTimer
17                 RL[r.repo] ← (r.client)
18                 prefetch++
19         else if r = GETlayer then
20             if r.repos in PL[] and r.layer = PL[r.repo] then
21                 serve from PL[r.repo]
22                 prefetch_hit++
23             else
24                 serve from object store
25         for repo in RL[] do
               /* Evict expired layers from repo list                    */
26             if RL[repo].watchTimer > watchDuration then
27                 RL[repo] ← Evict
28         for repo in PL[] do
               /* Evict expired, prefetched layers                       */
29             if PL[repo].LayerTimer > layerDuration then
30                 PL[repo] ← Evict
```

Chapter 5

The New Design

This chapter describes the new registry design based on the characterization of IBM's registry. Because manifests are small and handled by services like Redis, the new design focuses on how layers should be stored and requested. From Section 2.2.2, the current distributed registry design requires multiple services layered together, which does not provide sufficient scalability for increasing client loads. Furthermore, the new design seeks to add caching to the registry to improve performance. However, even though layers are small enough to cache, many layers are still bigger than current distributed caches like Redis are designed to handle. Additionally, from Figure 4.7, it can be noted that the entire three month IBM workload can be held in less than 11 TB of SSD, so the design eliminates the reliance of a backend object store completely by allowing registry nodes to share layers between them.

By eliminating the backend object store, its duties are inherited by the registry itself. These duties are to provide consistency and failure recovery for layers. Because layers are content addressable, consistency is not an issue. Backend object stores also typically require the use of specialized proxies to locate the objects requested by clients. The next section will cover the three choices of proxy, and justify the new design's choice.

5.1 Proxies

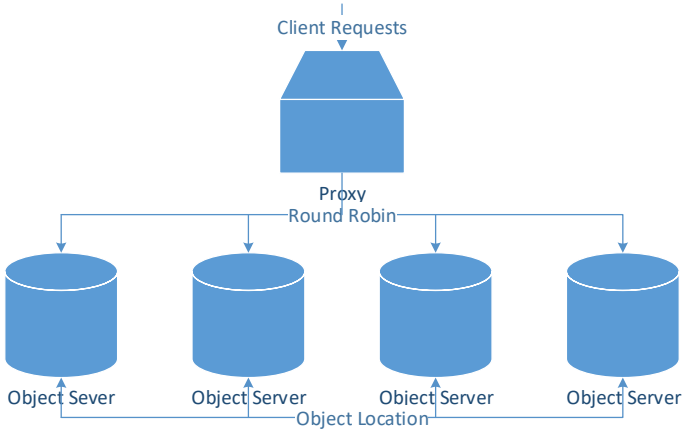
There are three choices of proxy in any given distributed system: replica oblivious gateway, replica aware gateway, and replica aware client [35].

A replica oblivious gateway is unaware of the location of the object on the storage nodes or cache. This requires two additional hops from the proxy to an object server, and then from the object server to the location of the object. This configuration is shown in Figure 5.1a. The advantage of replica oblivious gateways is the ability to use existing solutions such as NGINX out of the box with minimal configuration.

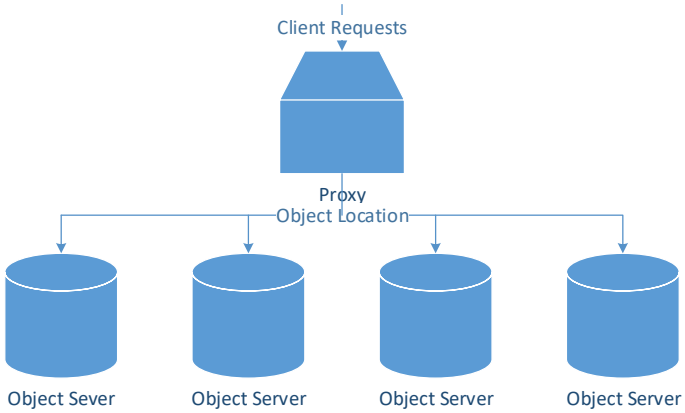
A replica aware gateway knows the location of the object, so forwards the request directly to the object server with the object, allowing for only one additional hop. Unlike a replica oblivious gateway, a specialized proxy usually needs to be developed specifically for the system. Figure 5.1b shows this configuration.

Finally, in a replica aware client, the client is aware of the location of the object, and can request it directly from the object server in one hop. However, because this requires the clients to be aware of the internals of the object store, this option is usually not used. Figure 5.1c illustrates this method.

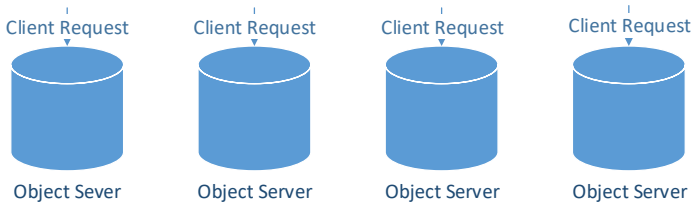
In order make the appropriate choice for the design, two experiments were run in order to gauge the memory wastage of a registry LRU cache if a replica oblivious gateway is used, and to gauge the overhead of the extra hop required for a replica aware gateway. For the first experiment, a cache simulation similar to Section 4.4 where the amount of memory that held repeat instances of layers was collected. Each registry that appeared in the anonymized traces was given its own LRU cache and served the same requests as they did in the traces. The amount of memory was recorded for every hour of trace time, resulting in Figure 5.2. Throughout the simulation, about 46% of the memory held redundant layers, cutting the



(a) Replica Oblivious Gateway



(b) Replica Aware Gateway



(c) Replica Aware Client

Figure 5.1: Replica oblivious gateways require three hops from the client to the object, replica aware gateways require two hops, and replica aware clients require only one hop.

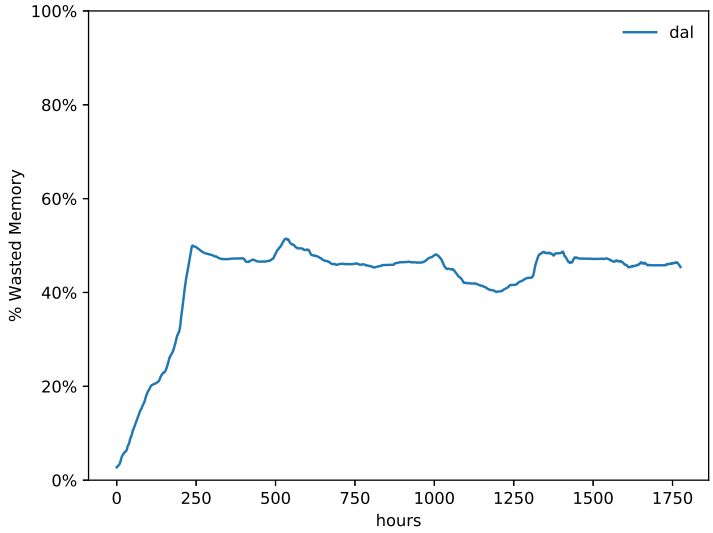


Figure 5.2: Percentage of memory used to store copies of layers in a replica oblivious gateway configuration. The total memory size was 110 GB.

effective cache size of all registries in half. This illustrates one shortcoming of the replica oblivious gateway, which makes it an undesirable choice for the new registry design.

For replica aware clients, the clients need to be aware of the structure of the object servers. In this case, the clients are the Docker daemons that need to be aware of which registries hold what layers. To make this possible, the daemons use a specialized consistent hashing function on the layers to both identify the location of the layers and load balance their requests across the registry nodes. The next section describes the hashing function.

5.2 Consistent Hashing Function

Consistent hashing is used in the new design both for determining the location of the layer and to load balance layer requests evenly across the registry nodes. As stated previously in Section 3.2.1, consistent hashing maps an object to a node in such a way that prevents

unnecessary remapping when nodes are added or removed. The way this is achieved is by hashing the object servers' identities using a standard hashing function, and then organizing the hashed identities into a mathematical ring. To locate an object, its identifier is hashed and mapped to one of the hashes of the ring. This is done efficiently by doing a binary search for the object hash on the list of node hashes. Consistent hashing functions also allow for easy failure recovery, as an object server can store replicas of their objects in the next node along the ring. Then, if that node fails, the consistent hash function will return the next node in the ring with the replica.

An alternative approach to consistent hashing is rendezvous hashing [33], which involves appending each node ID to the object identifier, hashing them, and then determining which node produces the greatest hash. This is less efficient than consistent hashing for large numbers of nodes, however, because consistent hashing does not need to compare every node like rendezvous hashing does.

For the new design, a custom consistent hashing function was created to take advantage of how layers are already addressed. Because layer digests are SHA256 hashes used to identify them, the digest can be used directly to map it to a registry. Each registry uses a set of pseudo identities to help distribute the layers across the node. The layers are also replicated to multiple nodes to help mitigate the load of very popular layers. The details of the replication are in Section 5.4.

Two different hashing 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 node is known as the master node for the layer in question. For pull requests, the read hash function selects from a set of registries which contains the replica of the layer. The registries which hold the replicas of the layer are known as slave registries for that layer. Algorithms 3 and 4 detail the hash functions. The consistent hashing function

was implemented in both Go and Python, using 479 lines of code in Go and 129 lines of code for Python.

Algorithm 3: 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 ← BinarySearch(nodeIDs, Layer)
3      return Nodes[ID]
```

Algorithm 4: 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 ← BinarySearch(nodeIDs, Layer)
3      ReplicaNodeList.append(Nodes[ID])
4      index ← nodeIDs.indexOf(ID)
5      while Length(ReplicaNodeList) < NumberOfReplics do
6          index ← index + 1%length(nodeIDs)
7          ID ← nodeIDs[index]
8          node ← Nodes[ID]
9          if node not in ReplicaNodeList then
10             ReplicaNodeList.append(node)
11     return RandomChoice(ReplicaNodeList)
```

To generate pseudo identities for the registry nodes, a hash chain is used. The first pseudo identity is found by SHA256 hashing the hostname and port number of the registry node, and the rest of the identities are found by hashing the previous identity that was calculated. By default, the new design uses 50 pseudo identities and 3 replicas for layers. To determine the effectiveness of this consistent hashing algorithm, the anonymized traces were used to determine the layer distribution across six nodes: thor9:5005, thor10:5005, thor11:5005, thor19:5005, thor20:5005, and thor21:5005. The `http.request.uri` field was hashed to provide a digest to represent the layer. All the layer GET and PUT requests were hashed, providing the results shown in Figure 5.3. The layers from the traces distribute themselves almost evenly on the nodes, with no more than 3% deviation from an ideal distribution.

Lastly, in order for clients to be able to use consistent hashing, they must be aware of all

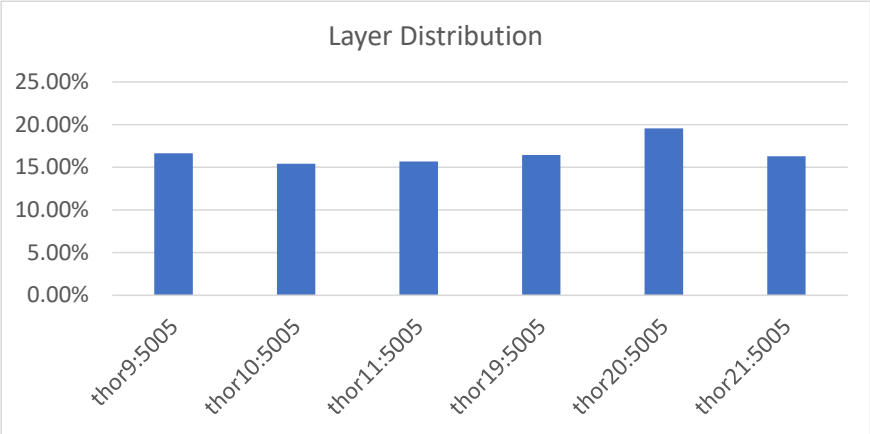


Figure 5.3: Distribution of anonymized requests accross 6 nodes using consistent hashing.

nodes currently in the system and any nodes that are added or removed. Registries hold the same requirement. To meet the requirement, a metadata service is used to monitor the health and availability of all registries in the system.

5.3 Coordination

Zookeeper [32] was chosen as the meta data server for the new design as it was developed as a highly reliable service to provide distributed coordination. Zookeeper itself is a distributed service that organizes data into Znodes. Zookeeper accepts new Znodes or updates to already existing Znodes via a consensus of the zookeeper nodes. Zookeeper clients have the ability to set watches on Znodes that allow them to be notified when any changes to the Znode occur.

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 children Znodes. Zookeeper also supports ephemeral Znodes, which exists as long as the session that created them is active, and are deleted when the session is terminated.

The new design makes use of ephemeral Znodes to register active registries. When a registry is created, it first registers itself with Zookeeper by creating a ephemeral node under a `/registries` Znode. If the `/registries` Znode does not exist, the registry will create it and then create its ephemeral Znode. Figure 5.4 illustrates the design's usage of Znodes in the system. After it creates its ephemeral Znode, it will create a watcher on `/registries`, that will provide the registry with the addresses of all the other registries in the system, and give notification of any changes with the other registries. Note that because the registries use ephemeral Znodes, if the registry goes offline, its Znode will be removed and all other registries will be notified of the removal. 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. To issue the specific request, the registry server had to be modified to accept a new request, and serve a JSON with the list of active registries. To request the list of registries, the client issues the following request:

```
GET https://<registry URL>/v2/registries
```

Clients can issue this request when they first come online, or when a layer request fails. Section 5.4 details when the registry will reject requests, allowing the clients to update their consistent hashing rings. Figure 5.5 shows how clients issue requests for layers and registry lists in the new design. For simplicity, each registry is represented with one pseudo identity.

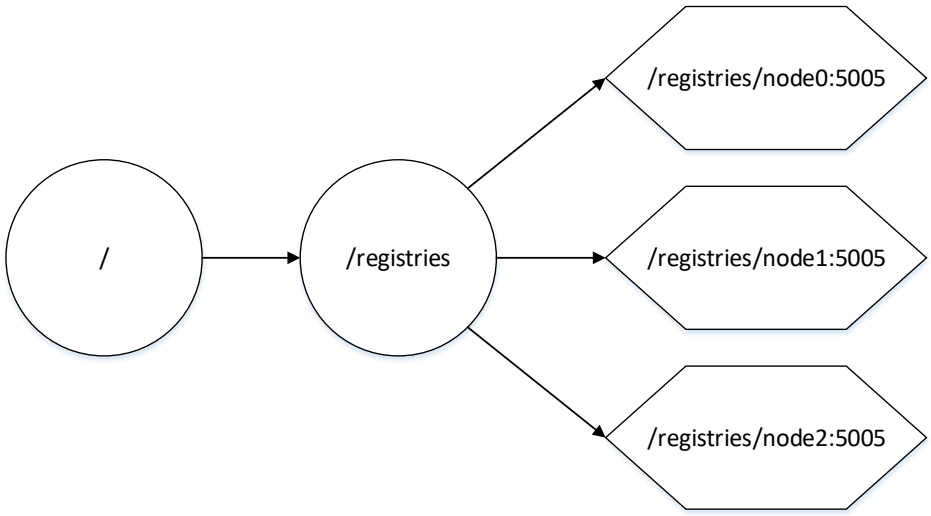


Figure 5.4: Zookeeper Znode setup for the new design. The circles represent normal Znodes, and the hexigons represent ephemeral Znodes. Note that / is the root Znode, which is the parent for all other Znodes. Each registry added to the system creates an ephemeral Znode of its hostname and port number, which allows the other registries to connect to it.

5.4 Storage And Caching

Because the new design eliminates the backend object store, the registries take the responsibility of layer storage. To achieve this, the registry relies on a custom storage driver based on the filesystem driver, but with additional functionality to request and forward layers to and from other registries, based on its consistent hashing ring. The registry forwards layers to other registries during layer pushes and when new registries are added. The registry requests layers from other registries when it is a slave for the layer being requested but does not have it locally.

During a layer push, the standard registry stores the layer data into a temporary file. Once the registry receives the final PUT request, the registry verifies the hash of the file with the digest provided by the PUT request. If there are no errors, the registry then moves the

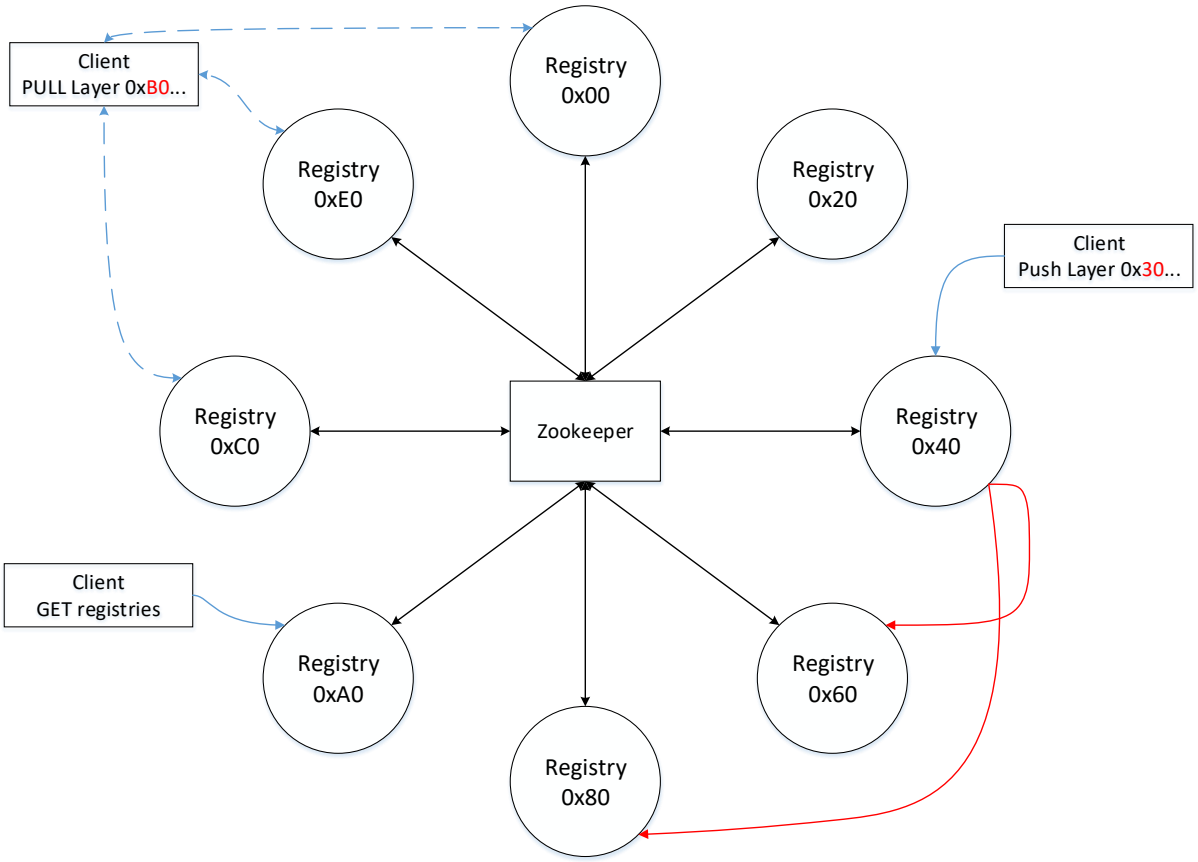


Figure 5.5: The new design. Registries form a consistent hashing ring, and use Zookeeper to identify each other. Clients request a complete list of registries from any registry it knows of to form a copy of the ring. Clients look up layer digests in their ring to issue push and pull requests. Clients 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 onto the next registries, or slave nodes, represented by the red lines. Clients randomly select the master or slave node to issue pull requests, represented by the dashed lines.

temporary file into a directory named by its digest, at which point the layer is ready to be requested. In the new design, while the file is being verified, the new registry also checks to make sure it is either a master or a slave for the layer. If the registry is the master, it will forward the layer to the slave registries to create replicas. If the registry is a slave it will proceed as if it were a standard registry. However, if the registry determines that it is neither the master or the slave, it will delete the file and return an error to the client that issued the push. Unfortunately, this is the only way to determine if the layer being pushed is being pushed to the master because of how the Docker API works.

Registries also forward layers to new registries that are added to the system. Once a registry detects a new registry via Zookeeper, it will loop through all the layers it has locally to see if the new registry is the master of any of them. If it has any, then it will forward those layers to the new registry if the it was the old master for those layers. This process will take anywhere from seconds to minutes depending on how many layers and how many other registries are in the system. The registry will also delete any layers for which it is no longer responsible. Figure 5.6 shows how scalability is handled.

Registries can fetch layers from the master registry if they are a slave for the layer. 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. If a client requests a layer while it is being fetched for another client, the registry will reply with a 404 error, causing the client to issue another request. Figure 5.6 illustrates how the registries request layers from each other to mitigate failures. Lastly, registries respond with 404 errors for any layers for which they are neither masters nor slaves. This can let the client know if its ring is outdated.

Caching is also a part of the new design. Layers are cached using the Bigcache [4] library for Go. This library provides a very 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 1MB typically improves performance as it allows more layers to be held in memory, and because layers over 1MB only receive minor improvement from being fetched from memory versus being fetched from SSD. In total, 905 lines of code were added to the registry for the new functionality.

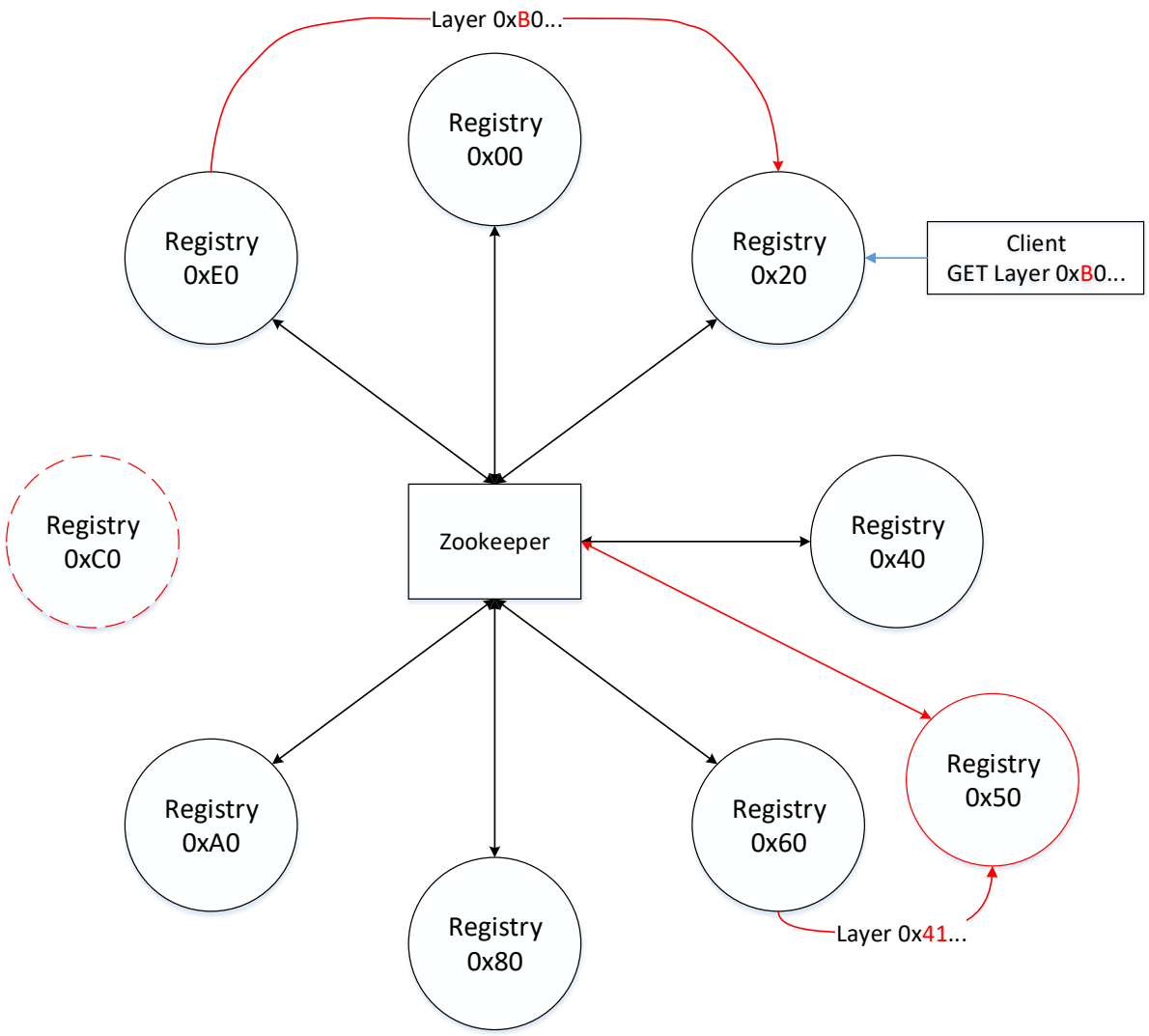


Figure 5.6: This figure illustrates both scalability and failure recovery. Registry 0xC0 went offline. Clients attempting to connect to the registry will fail, prompting them to issure registry list requests from the other registies. Once the clients have the correct ring, they may attempt to request a layer from 0x20, which has become the slave. The registry will fetch the layer from the new master and then reply to the client. For scalability, when a new node is added, it will become the master of some layers. The old master of the layers will automatically forward the layers to the new registry once registers with Zookeeper. The new registry will request the other layers it is responsible for from the layers' masters when clients request them from the new registry.

5.5 Testing

The trace player was used to test and benchmark the new registry. In order to test it, consistent hashing was added to Warmup mode and to the client. When the client starts its processes, the processes first request the list of layers from the registry provided by the master. If a request fails, the client will attempt to request an updated list from all the registries it knows about until the request is successful. The client will then rehash the layer and send the request again. The client will attempt three requests before marking the request as failed.

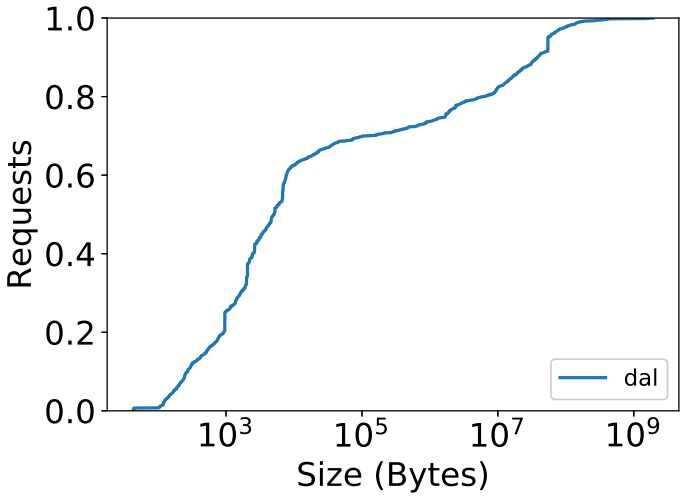


Figure 5.7: CDF of the layer sizes used for testing. The average layer is is 13.7 MB.

For testing, one trace was used from July 24th, 2017, which was selected because it held over 1 thousand clients connected in parallel. Figure 5.7 shows the size distribution of the first 5 thousand requests, containing just over 15 hundred layers. Although the trace differs slightly than the entire 3 month period, the average size remains the same. The trace also contains 4% write requests.

All tests were carried out on 8 core machines with 16GB of RAM, 512GB of SSD, and 10Gbps network cards. In total 11 machines were used. For the new design, one node was used as a dedicated Zookeeper node. Lastly, every test used 50 pseudo identities per registry node and 3 replicas per layer. Caching, scalability, and failure recovery were all tested. Additionally, the new design has also been compared to the conventional registry setup.

5.5.1 Caching Results

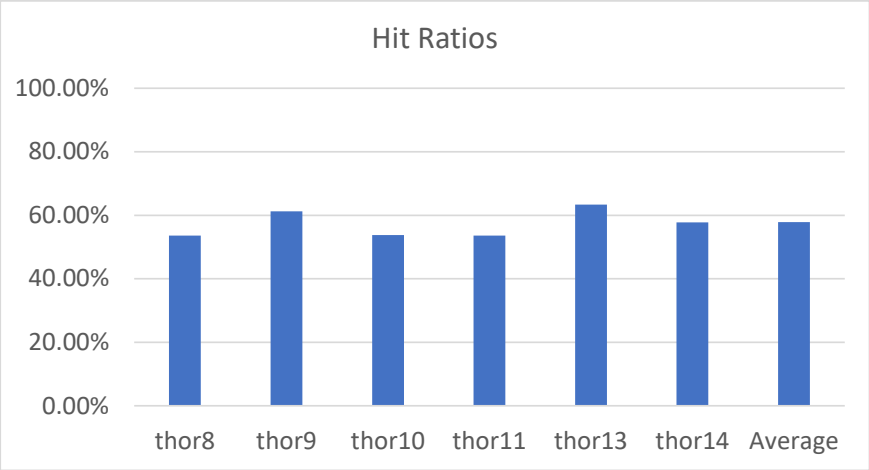


Figure 5.8: Hit ratios on the caches accross 6 nodes. The smallest ratio is 53.6%, the largest is 63.3%, and the average hit ratio is 57.8%.

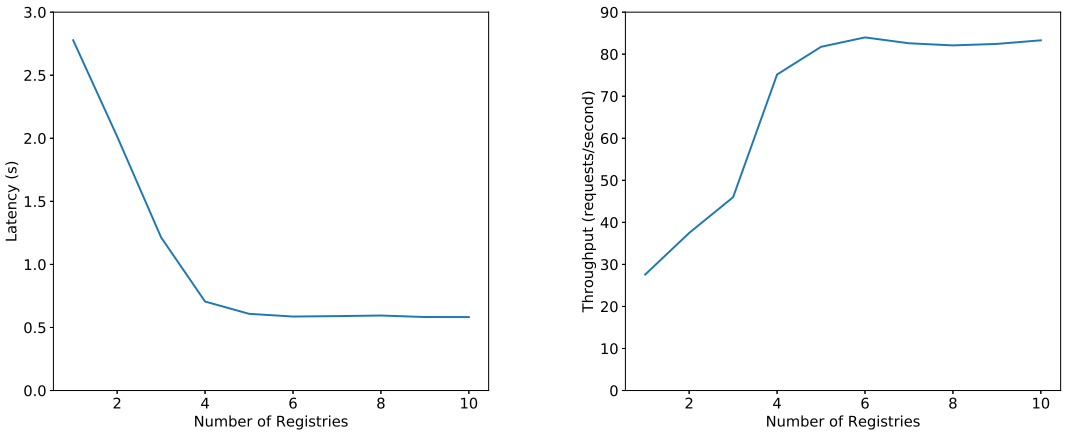
To test caching, the trace player used 4 client nodes to send 5 thousand requests to a 6 node registry setup. Each registry node used an 8GB cache, and only cached layers less than or equal to 1MB. Each trace player client spawned 50 clients to send requests for a total of 200 clients sending requests in parallel. To fill the cache of each node, the experiment was run

twice, with the caching data collected on the second run.

The average hit ratio was 57%. However, it's important to note that 25% of the layers requests was greater than 1MB, so the effective hit ratio is closer to 76%. Finally, the cached layers' latency was an order of magnitude less than uncached layers. These results are very similar to the caching simulation explained in Section 4.4.

5.5.2 Scalability Results

For testing scalability, the number of registries was increased from 1 registry to 10, while keeping the number of clients constant at 100. To give a fair comparison between testing the number of registries, a special script was written to request each layer from each slave registry in order to fully populate newly added nodes. The registries were run as containers to constrain their resource usage. Each registry used 2 CPUs and 4GB of RAM. The cache size for each registry was 2GB.



(a) Latency (b) Throughput

Figure 5.9: This graph shows the throughput and latency from increasing the number of registries while keeping the number of clients constant at 100.

The results of the test are shown in Figure 5.9. Because 3 replicas were used, the first 3 registries were responsible for storing and serving all the layers. However, once another registry was added, the load on each registry was significantly decreased. After 5 registries, the performance increase tapers as the bottleneck moves from the registry to the number of clients issuing the requests.

5.5.3 Fault Tolerance Results

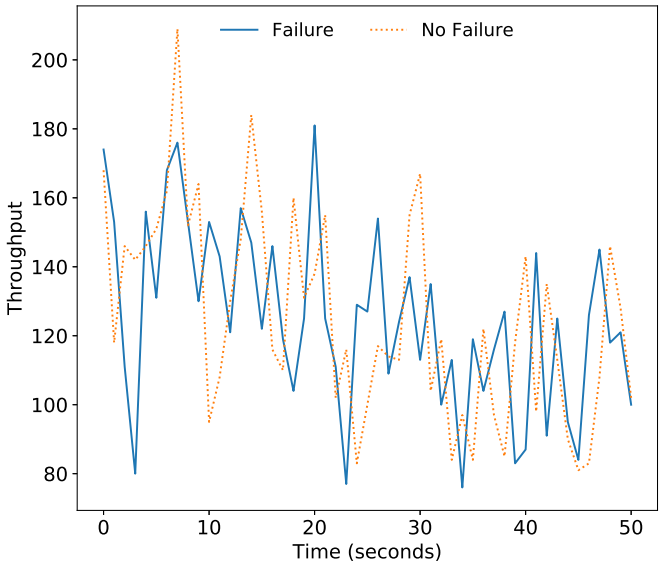


Figure 5.10: Failure recovery test. Five registries were used to serve 10 thousand requests. Time 0 on the graph started 20 seconds into the test. At time 0, a registry was killed (failure). At time 3, Zookeeper removed the ephemeral Znode for the killed registry, allowing the registries and clients to remove the node from their rings. At time 10, the registry was brought back online. No Failure is the performance of 5 nodes without any failures.

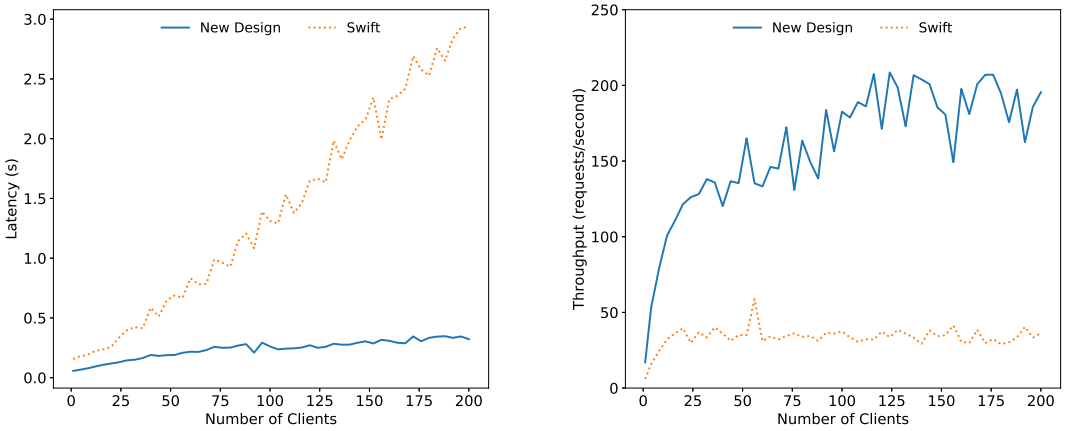
To test fault tolerance, a registry was killed during the test and brought back up 10 seconds later. Five registries were used for this experiment, and 10 thousand requests were sent. Figure 5.10 shows the results of this test. The ephemeral Znode for the killed registry exists

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. Ten seconds after the node went down it was brought back up. The registries became aware of the new registry within milliseconds of the registry joining.

5.5.4 Throughput and Latency Comparison

The last test conducted compares the performance of the new design over the conventional distributed registry. Both registries were deployed on 6 nodes, and the new design was configured 8GB of RAM. The conventional distribution used Swift as its backend object store, which was installed on the same nodes as the registries. It also used NGINX as a load balancer, which was run on a separate node. Each registry connected to a Swift proxy running on the same node. The Swift object servers also ran on the same set of nodes, and Swift was configured to have 3 replicas like the new design. The performance of both systems were compared by increasing the number of clients connecting to the registry, with the new system load balancing using consistent hashing and the conventional using round robin. In each test the clients sent 5 thousand requests. Figure 5.11 shows the results of the experiment.

The new design outperforms the conventional design significantly at all levels. This is due in part because of caching and because of the clients' ability to request the layer with one network hop. Additionally, the bottleneck in the swift setup is hard to identify. Clients must communicate via NGINX, which load balances the requests to the 6 registries. The registries use the Swift proxy to fetch the layer from one of the object nodes, often causing additional hops between nodes.



(a) Latency (b) Throughput

Figure 5.11: Throughput and latency comparison between the new design and the current design, with Swift used as the backend object store.

Chapter 6

Future Work

This chapter focuses on the future work for the project. There are several avenues of further research that can make the project stronger. These range from a deeper analysis of the anonymized traces, changes to the new design, and changes to the trace player.

One of the main points for future research would be including the manifests in the new design. Currently, manifests are handled as they are in the normal design, but a deeper study on them can bring new insights to further improve Docker performance. The first step in incorporating manifests would be to add the ability to create them using the trace player. Creating manifests in a useful way would be tricky however, as the anonymized traces only show how layers relate to manifests that share the same repository, but do not associate specific layers to manifests. Successfully adding manifest support will enable testing on prefetching algorithms such the one discussed in Section 4.5.

Another change to the trace player that can open new research avenues would be to enable support for real layers and images. Implementing this change could map real layers to the anonymized traces either randomly or based on layer size. Pushing and pulling real images would enable the exploration for registry side finer grained deduplication.

Using real images may also open up the opportunity for the registry to scan the layers for malicious or out of date code. Because manifests are signed by users, the registry would likely not be able to automatically update the layers, but may be able to notify users during

the pull that they should update their containers.

Finally, the new design could use the manifest requests to also provide the locations of the layers. This can eliminate the need for clients to keep their own rings. However, because registries can fail at any time, clients will still need a way to locate layers. This could be achieved by configuring registries to reply with redirect requests rather than 404 errors for any layers for which they are not responsible.

The last area of further research that can move forward is with a thorough caching analysis on the anonymized traces. There are many caching replacement policies which can improve the registries' performance even more. Some examples of different caching policies which should be tested would be least frequently used or adaptive replacement policies. Prefetching and caching can also be studied together to find the policy most effective for serving the Docker workload.

Chapter 7

Conclusion

By improving the Docker registry scalability and performance, container distributions are made faster. This allows for more rapid scaling and updating of containerized applications. As time passes, more and more cloud providers will opt for containers over virtual machines, so making sure the container distribution time is minimized is essential.

This thesis presents a new Docker registry design that coalesces all the services of the current design into a single, highly scalable service. The performance improvements shown by the use of caching and the elimination of proxies highlights the need to design distributed services with scalability in mind. By creating simple, scalable solutions instead of layering preexisting services with new services also eases maintainability and the identification of bottlenecks. By using the new design, administrators can simply add more registry nodes as more clients use their system to store and distribute their containers.

Bibliography

[1] Artifactory. https://www.jfrog.com/confluence/display/RTF/Docker+Registry.

[2] Amazon aws. https://aws.amazon.com/.

[3] Microsoft Azure. https://azure.microsoft.com/en-us/.

[4] Bigcache. https://github.com/allegro/bigcache.

[5] BitTorrent. http://www.bittorrent.com/.

[6] Bottle. https://bottlepy.org/docs/dev/.

[7] CoreOS. https://coreos.com/.

[8] What is Docker, . https://www.docker.com/what-docker.

[9] Dockerhub, . https://hub.docker.com.

[10] alibaba Dragonfly. https://github.com/alibaba/Dragonfly.

[11] DXF. https://github.com/davedoesdev/dxf.

[12] Google Cloud Storage. https://cloud.google.com/storage/.

[13] Git.

[14] The Go Programming Language. https://golang.org/.

[15] Google Container Registry. https://cloud.google.com/container-registry/.

[16] HelloBench. https://github.com/Tintri/hello-bench.

[17] Microservices and Docker containers, . goo.gl/UrVPdU.

[18] IBM Cloud Container Registry, . https://console.bluemix.net/docs/services/Registry/index.html.

[19] Ipfs. https://www.ipfs.com/.

[20] Arvados keep. https://dev.arvados.org/projects/arvados/wiki/Keep.

[21] Memcached. https://memcached.org/.

[22] Microservices Architecture, Containers and Docker. goo.gl/jsQ1sL.

[23] Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html.

[24] Alibaba Cloud Object Storage Service. https://www.alibabacloud.com/product/oss.

[25] Quay.io. https://quay.io/.

[26] Redis. https://redis.io/.

[27] Amazon S3. https://aws.amazon.com/s3.

[28] OpenStack Swift. https://docs.openstack.org/swift/.

[29] Trace Player, . https://github.com/chalianwar/docker-performance.

[30] . https://https://dssl.cs.vt.edu/drtp/.

[31] VMWare ESX. https://www.vmware.com/products/esxi-and-esx.html.

[32] Zookeeper. https://zookeeper.apache.org/.

[33] A name-based mapping scheme for rendezvous. Technical report, 1996. URL https://www.eecs.umich.edu/techreports/cse/96/CSE-TR-316-96.pdf.

[34] 451 Research. Application Containers Will Be a $2.7Bn Market by 2020. http://bit.ly/2uryjDI.

[35] Samer Al-Kiswany, Suli Yang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Nice: Network-integrated cluster-efficient storage. In *HPDC '17 Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 29–40. ACM, 2017. ISBN 978-1-4503-4699-3. URL https://dl.acm.org/citation.cfm?id=3078612.

[36] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee1, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, Big Sky MT, 2009. ACM.

[37] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, 2018. USENIX Association. ISBN 978-1-931971-42-3. URL https://www.usenix.org/conference/fast18/presentation/anwar.

[38] Paul B Menage. Adding generic process containers to the linux kernel. 01 2007.

[39] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003. ISBN 1-58113-757-5. URL https://dl.acm.org/citation.cfm?id=945462.

[40] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *ACM SIGMETRICS*, 2005.

[41] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, November 1996. ISSN 0734-2071. doi: 10.1145/235543.235544. URL http://doi.acm.org/10.1145/235543.235544.

[42] 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.

[43] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *USENIX OSDI*, 2004.

[44] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-28-7. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter.

[45] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. FID: A Faster Image Distribution System for Docker Platform. In *IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 191–198, Tucson AZ, 2017. IEEE. ISBN 978-1-5090-6558-5.

[46] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for

relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, El Paso TX, 1997. ACM.

[47] Alexander Kedrowitsch, Danfeng Yao, Gang Wang, and Kirk Cameron. A first look: Using linux containers for deceptive honeypots. In *SafeConfig '17 Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, pages 15–22. ACM, 2017. ISBN 978-1-4503-5203-1. URL https://dl.acm.org/citation.cfm?id=3140371.

[48] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. Tap: Table-based prefetching for storage caches. In *USENIX FAST*, 2008.

[49] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. CoMICon: A Co-Operative Management System for Docker Container Images. In *IEEE IC2E*, pages 116–126, Vancouver Canada, 2017. ISBN 978-1-5090-5817-4.

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

[51] John Pescatore. Nimda worm shows you can't always patch fast enough. https://www.gartner.com/doc/340962.

[52] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *FAST'02*. USENIX, 2002.

[53] Adrian Sanabria. Malware analysis: Environment design and artitecture. Technical report, 2007. URL https://www.sans.org/reading-room/whitepapers/threats/malware-analysis-environment-design-artitecture-1841.

[54] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance

Alternative to Hypervisors. In *ACM EuroSys*, pages 275–287. ACM, 2007. ISBN 978-1-59593-636-3. URL https://dl.acm.org/citation.cfm?id=1273025.

[55] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*, pages 149–160, San Diego CA, 2001. ACM.

[56] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In search of the ideal storage configuration for Docker containers. In *IEEE AMLCS*, Tucson, AZ, 2017. IEEE. ISBN 978-1-5090-6558-5. URL http://ieeexplore.ieee.org/document/8064124/.

[57] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Cakowski, Cezary Dubnicki, and Aniruddha Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *FAST'10*, San Jose, California, 2010. USENIX.

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

[59] Zhe Zhang, Amit Kulkarni, Xiaosong Ma, and Yuanyuan Zhou. Memory resource allocation for file system prefetching: From a supply chain management perspective. In *ACM EuroSys*, 2009.