State of the Union: Containers – Part 2

A Collection of Design & Development Principles and Technical Specifics of Ecosystem Tools

Mike Metral
Product Architect, Rackspace
mike.metral@rackspace.com
April 2015

Rev. 3

Table of Contents

1 Introduction

This report builds upon and is intended to serve as a follow-up to "State of the Union: Containers – Part 1." It aims to highlight the more technically inclined insights, usage and design/development tips in terms of using containers as well as the surrounding ecosystem.

Again, because the container space is evolving at such an incredible pace it is worth restating the facts and findings as well as the opinions expressed in this report are *extremely* time sensitive as a small time-span of merely 6 months can and has rapidly shifted what the community has come to think and consider as a viable option.

Lastly, because there are flood of options not only in the design and usage of containers, but with the ecosystem as well, the interpretation, usage and permutations capable will vary immensely, not to mention the use case and requirements your application(s) needs, so this report will attempt to be as neutral as possible.

2 Overview

It is worth mentioning that because the natural inclination of the technical community is to test the boundaries of a given technology, and some are more vocal than others, as expected, there is an unclear line as to what is an unwritten accepted standard by the community and what is heresy.

As a general rule of thumb, the authors believe that authoritative decisions from a design and development standpoint come directly from Docker Inc. themselves, primarily from CTO Solomon Hykes. Though there are several voices in the container industry, Solomon's opinions & suggestions tend to be well received (business theatrics with CoreOS aside), and are almost relatable to that of Linus Torvalds with the Linux Kernel. However, since Docker Inc. is in fact still a business, one can expect that Solomon has a responsibility to his investors and himself to position all of Docker's offerings as the preferred methodology – as one can imagine in these scenarios, the suggestions should be taken with a grain of salt.

3 Container Design Philosophy

According to Docker's "Best Practices" guide, it is suggested that you "run only one process per container."[1] Though it initially seems as a simple design decision that you could easily overlook, this concept has been met with various opinions,

[1] https://docs.docker.com/articles/dockerfile_best-practices/#run-only-one-process-per-container

discussions and interpretation in the community. Though you can *actually* run more than one process per Docker container, some believe that this diverges away from the simple packaging and ease-of-use containers are supposed to provide. Confusion in this philosophy stems primarily from newbies wanting to find a relatable manner to use containers, and their natural instinct is to treat it as a VM that they would SSH into, hence the necessity for multiple processes. However, once one realizes that there are other methods of working with a container (i.e. enforcing proper logs, using development containers with a TTY for shell interaction etc.) this necessity and/or hurdle of packing many different processes into a container is no longer needed.

When this insight is made, the offering that containers provide really shines through and the instinct to create monolithic packages, such as one would do with a VM, becomes less relevant. Thus, the adoption of a microservice architecture becomes prominent as its relationship to containers not only appears as a better & clearer fit, but a concept that is now synonymous with using containers.

Martin Fowler describes the concept as such:

> The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.[2]
>
> If one is willing to accept this concept when using containers, then it is safe to state that microservices are emerging as the pioneered methodology when it comes to building your application's stack. It evangelizes software to not only become more modular, but it intrinsically advocates for loose coupling of dependencies, which aid in alleviating the pain points of CI/CD, and ultimately, aid in creating better software.
>
> In closing, it is easier from both a mental and technical stand-point that one should manage and use Docker containers as roles such as an app, database, cache layer, load balancer etc. rather than as individual processes (nginx, apache2, sshd etc.) – doing so, will highlight the strengths of a container + microservice architecture for your application's stack.

[2] http://martinfowler.com/articles/microservices.html

4 Docker Best Practices

4.1 Dockerfile

4.1.1 Overview

To build a Docker image, application & dependency specifics aside, there are two primary methods to do so:

- Update a container created from an image and commit the changes to an image or
- Use a Dockerfile

The Update & Commit route requires that one run a previously created Docker image that you've retrieved from either a public or private image registry and then instantiated the container on your host running the Docker daemon, preferably with an activated TTY. Once you have access to the container, you can make the required changes to personalize it. Upon completion, you exit the container and use `` `docker commit` `` to commit a copy of the container to your local or remote registry.

Using the update & commit methodology is the simplest way of an extending an image, but its not easy to maintain or share and it is definitely not the cleanest. Going the Dockerfile route is preferred as doing so allows you to utilize the `` `docker build` `` command to build images from scratch. In a Dockerfile you can prescribe the base OS for the container, in addendum to the instructions you'd like to execute inside the image from installing packages to setting & configuring options.

Here is a snippet of a simple Dockerfile:

```
# This is a comment

FROM ubuntu:14.04

MAINTAINER Kate Smith ksmith@example.com

RUN apt-get update && apt-get install -y ruby ruby-dev

RUN gem install sinatra
```

Each instruction creates a new layer of the image. At the time of this report, an image cannot have more than 127 layers regardless of the storage driver. This limitation is set globally to encourage optimization of the overall size of images[3].

It is worth nothing that the base OS initially chosen when starting off tends to be the most familiar OS one is accustomed to working with i.e. Ubuntu, CentOS, Fedora etc.

[3] https://docs.docker.com/userguide/dockerimages/

However, it becomes evident that most of the interaction with the actual OS revolves around the filesystem layout and its binaries. For example, if you chose Ubuntu as your base OS, and you aren't doing anything super extravagant in terms of the OS itself, then you could potentially switch to the Debian image as it will most likely have everything you need and can save you at least 100MB+ in size. Again, it is a per case basis, but if you're utilizing a bloated OS base when BusyBox could suffice, then you're not only consuming more space than needed, you're also adding time to Docker builds and Image Repository interaction.

4.1.2 Dockerfile Best Practices

The following list is a collection of unordered recommendations and tips that don't get enough upfront attention when starting off with Docker. One should keep these tips in mind so that you can ultimately improve creation and utilization of Dockerfiles to build your container images.

- Use a .dockerignore file to exclude directories and/or files from the build process
- Keep the number of instructions/layers to a minimum as this ultimately affects build performance and time
    - Consolidation of similar instructions into a single layer is preferred (i.e. in the snippet from the previous example, place the `` `gem install sinatra` `` in the preceding line)
- Build cache: During the process of building an image Docker will step through the instructions in your Dockerfile executing each in the order specified. As each instruction is examined Docker will look for an existing image in its cache that it can reuse, rather than creating a new (duplicate) image
    - In the case of the ADD and COPY instructions, the contents of the file(s) being put into the image are examined. Specifically, a checksum is done of the file(s) and then that checksum is used during the cache lookup. If anything has changed in the file(s), including its metadata, then the cache is invalidated.
    - Aside from the ADD and COPY commands, cache checking will not look at the files in the container to determine a cache match. For example, when processing a RUN apt-get -y update command the files updated in the container will not be examined to determine if a cache hit exists. In that case just the command string itself will be used to find a match.[4]
- When testing an edit to your codebase, write a simple Dockerfile describing your build process (usually in 5 to 10 lines), then build a new container from source with `` `docker build` ``
    - Building is very fast because docker re-uses previous build steps whenever possible (see preceding "Build cache" tip). And by building

[4] https://docs.docker.com/articles/dockerfile_best-practices/#build-cache

every time, you can use containers as reliable artifacts. For example, you can go back and run the container from 4 changes ago to inspect a problem, or run long tests on a version while editing the code.

- For a development environment, map your source code on the host to container using a volume so that way you can use your editor of choice on the host and test right away in the container
    - This is done by mounting the current folder as a volume **rather** than using the ADD command in the Dockerfile
- Know the Differences Between CMD and ENTRYPOINT
    - There are many details of how the Dockerfile CMD and ENTRYPOINT instructions work, and expressing exactly what you want is key to ensuring users of your image have the right experience. These two instructions are similar in that they both specify commands that run in an image, but there's an important difference: CMD simply sets a command to run in the image if no arguments are passed to docker run, while ENTRYPOINT is meant to make your image behave like a binary. The rules are essentially:
        - If your Dockerfile uses only CMD, the provided command will be run if no arguments are passed to docker run
        - If your Dockerfile uses only ENTRYPOINT, the arguments passed to docker run will always be passed to the entrypoint; the entrypoint will be run if no arguments are passed to docker run
        - If your Dockerfile declares both ENTRYPOINT and CMD, and no arguments are passed to `` `docker run` ``, then the argument(s) to CMD will be passed to the declared entrypoint
        - Be careful with using ENTRYPOINT; it will make it more difficult to get a shell inside your image. While it may not be an issue if your image is designed to be used as a single command, it can frustrate or confuse users that expect to be able to use the idiom[5]

For further best practices and the full set of Docker's instructions please visit: https://docs.docker.com/articles/dockerfile_best-practices/

4.2 Image Repository

4.2.1 Community Images

After you've constructed your Docker images, one of the main advantages that Docker brings is focused on the fact that the community has created several Docker images for a slew of purposes. Many of the images created by users have been uploaded to the Docker Hub Registry (https://registry.hub.docker.com/) and this allows everyone to use them.

[5] http://www.projectatomic.io/docs/docker-image-author-guidance/

You can retrieve or pull an image by identifying the <user/image_name> and passing it to Docker, i.e: `` `docker pull cpuguy83/openvpn` ``

The Docker Hub Registry also has official images from certain providers ranging from OS companies to software providers i.e. Ubuntu, MySQL, Golang etc. Pulling these images has a shortened call such as `` `docker pull mysql` ``.

4.2.2 Personal Images

Of course, you too can upload your custom images to the Docker Hub Registry, CoreOS' Quay.io or to your own private image registry. Pushing it to the Docker Hub Registry is as simple as issuing `` `docker push <user/image_name>` `` after you've logged in using `` `docker login` ``. You can also tag the image if you wish to give it a label i.e. dev, production, v2 etc. to distinguish amongst various flavors of your image.

The default behavior for `` `docker push` `` is to push to the central Docker Hub Registry, but in the case that you prefer your own private Docker registry there is an image for that too. Once you've got it running, the only step that you are required to change is to login to your private registry as opposed to the central one. That switch is as easy as `` `docker login http(s)://<YOUR_DOMAIN>:<PORT>` ``.

4.3 Data & Stateful Applications

As soon as one gets over the initial hurdle of what containers can provide, the next natural step is to build an actual stack that could benefit from Docker – a common use-case is a 3-tier webapp with a frontend, middle logic layer and backend database. In doing so, the frontend and middle logic layers are perfect fits to be containerized, if they are stateless. However, when you get to thinking about stateful information such as a backend database layer or data in general that needs to be preserved, used and possibly written to logs, containers aren't the best match, yet.

4.3.1 Datastore & Logs

There are strong recommendations that you should never store data or logs in a container as the longevity of a container is short-lived and containers themselves are conceptually ephemeral, more so than VM's. Instead, one should store the data and logs by leveraging Docker's volume mounts to create either a data volume or a data volume container that is used and shared by other containers.

Using volume mounts creates a specially designed directory within one or more containers that bypasses the Union File System and provides a set of features for persistent and shared data:

- Volumes are initialized when a container is created. If the container's base image contains data at the specified mount point, that data is copied into the new volume.

- Data volumes can be shared and reused among containers.
- Changes to a data volume are made directly.
- Changes to a data volume will not be included when you update an image.
- Data volumes persist even if the container itself is deleted.

Data volumes are designed to persist data, independent of the container's life cycle. Docker, therefore, [will] *never* automatically delete volumes when you remove a container, nor will it "garbage collect" volumes that are no longer referenced by a container.[6] Data volumes are created by either using the `-v` flag in `docker run`, or by using the `VOLUME` instruction in your Dockerfile – in either case, both options ultimately do the same task: create a volume in the container that is mapped to a directory on the host itself, but the crux being that **its location is irrelevant to us**. The only varying factor between both options is that with the `-v` flag you can specify to mount a particular file or directory of your choosing from the host, by issuing a command such as `docker run –v /host/src/foobar:/opt/foobar` if you do care about the location of the volume's from the host perspective.

Data volume containers operate in a similar fashion to Data volumes, but are designed to persist data that you want to share between containers, or want to use from non-persistent containers. To use the Data Volume container with other containers, you first start by creating it: `docker create –v /data –name datastore mysql`. This command will start a container and exit immediately, because it is used as a container shell that references a newly created volume (that other containers will use), and not an active process like other application containers. Once the Data volume container exits, you can reference it from other containers by using the `--volumes-from` flag on the subsequent containers such as: `docker run –d --volumes-from datastore –name mywebapp foobar/webapp`. If you remove the datastore container or the containers that reference it, again it wont delete the volume where your data is stored. To delete the volume from disk, you must explicitly call `docker rm –v` against the last container with a reference to the volume[7]. This allows you to upgrade, or effectively migrate data volumes between containers or even the containers using the data volume container with no worries of losing the data itself.

Using volumes with containers is great because when you need to back up your database, logs etc. living in the volumes, which in turn live on the host, you would use your specific tools on the host itself as you've been doing in your organization before containers showed up. After doing so, you can carry out tasks such as upgrading the image itself by shutting the previous container down and starting the updated one with the same volume(s), or one can perform any other routine updates and maintenance. In summary, the general rule of thumb is that if it gets written to, it should be done in a volume.

---

[6] https://docs.docker.com/userguide/dockervolumes/
[7] http://docs.docker.com/userguide/dockervolumes/

Using a Data volume container sounds like a great solution to the data management issue, in theory, however, what happens when you want to scale out where your data resides or when your host machine goes down in say events such as a restart due to maintenance, or possibly failure? Well the short answer is that there is no great answer. The current standard is the same one we've come to know, and that is to effectively relay the data management tasks to self-managed systems such as using dedicated rsyslog servers for logs, or redundant cloud services such as cloud block storage and cloud database as a service to persist your database, but these could introduce topics such as cloud vendor lock-in and performance implications. Another strategy is to integrate the data management back into the stateless applications as we've been accustomed to, but doing so only creates the monolithic mess we're trying to avoid in the first place with containers.

In theory, we should be able to collocate the stateless apps with the data management services they rely on, with all of these services running in containers, but Docker isn't there quite yet and this quickly becomes a whole different ballgame which Docker itself does not touch. ClusterHQ is attempting to tackle this problem for containers with their tool Flocker, by using ZFS as the foundation for data management & replication. However, Flocker is still in the very early stages from being practical, and it assumes you are accepting of a clustered filesystem model and the nuances it can introduce such as relying on a storage pool, and not to mention I/O taking a performance hit. However, Flocker is open source and free to use if you care to kick the tires and ClusterHQ seems to be making serious waves as a leader on this front, so keeping tabs on them is well-advised.

For Docker to truly flourish as the new infrastructure stack that powers the next web, containers need serious work mitigating the data management problem. For now, it is suggested that you stick to the traditional data management solutions that live outside of containers, or strictly use Data volumes as a means to map to a host directory or a file from a container, that you can curate and manage external to Docker.

4.3.2 Block Storage

Given the previous section's description of Docker volumes to manage your data, and the assumption that you are operating on a cloud provider, it is only logical to think that one could use the provider's block storage as a service in conjunction with volume mounts, if you decided to go down that route.

In theory, this sounds great; however, you could imagine the scenario where you have several hundred containers running on a host, all requiring an allocated & attached block device to mount their individual volumes. In such a setting it is only a matter of time before you hit a limit of some sort at the account level, or more restrictively, at the service provider's infrastructure level – both of which are not novel issues you would have run into if you were running containers or not. However, you do have the option to map a Data volume container to a block storage

device and share said container with other containers without having to introduce an additional management layer.

4.4 Container Linking

Docker has a concept known as “linking” that allows you to connect containers via a socket or through a hostname using a sender and recipient model. Links can also be used to leverage service discovery between containers. The way links function is by having the client container use the private networking interface provided by Dockers to access an exposed port in another server container.

For example: to create a database backend server that will be linked, you start it just like any other container:

```
`docker run -d --name mysql_server mysql`
```

On the client container, you create the container as follows using an alias for the serving container’s name:

```
`docker run --rm -t -i --name webapp --link mysql_server:db mywebpp:v1
/bin/bash`
```

With linking established, the client is now allowed to access the information on the serving container. This access is facilitated by links in the form of a secure tunnel that is created between containers and the connectivity information is exposed to the client container in two ways: environmental variables, and via the /etc/hosts file.

In the environmental variable route, the client will see information such as the following:

- DB_NAME=/webapp/db
- DB_PORT=tcp://172.17.0.5:3306
- DB_PORT_3306_TCP=tcp://172.17.0.5: 3306
- DB_PORT_3306_TCP_PROTO=tcp
- DB_PORT_3306_TCP_PORT=3306
- DB_PORT_3306_TCP_ADDR=172.17.0.5

In addition to the environment variables, Docker adds a host entry for the source container to the/etc/hosts file.

4.4.1 Ambassador Pattern

In Docker linking, a pattern emerged that allows the proxying of connections between a server and client container. This separate proxy container transparently redirects connections based on parameters.

However, because ambassador containers themselves depend on links, they too are exposed to the issues linking has as described in the proceeding section, primarily, that it is only beneficial to use it as long as it does not fail and/or crash.

In essence, the usage of the ambassador pattern has dwindled down as far as the community is concerned as it's not necessarily adding any benefit that links don't already provide, and it further complicates the architecture without solving the underlying issues at hand.

4.4.2 Issues with Linking

As provocative as linking seems, it seems to be met with a couple of different nuances and issues when you begin to think of linking containers across different hosts as well as how much of an ephemeral lifecycle a particular container can hold.

Some problems and needs not addressed by links include:

- Service discovery suffers from the static nature of linking
- Links are volatile. IP addresses, port mappings, link names can change as the result of manipulating a link, and other containers are not notified of these changes nor can they trivially deal with these issues.[8]

With such issues, it becomes hard to use containers with links when certain impositions exist. Links functionality can only work for containers on a single / same node or by using the ambassador model, which itself isn't being adopted much.

Additionally, the exposure of the environmental variables in linking is done in a very odd manner (reference the previous section's example): one must know know the intended interface and the port of the service set in the environment beforehand, which itself is supposed to be handling the task of supplying you with what the interface and port *are* – a redundant and quite useless feature. For example, if one cares to know what protocol is being used in the connection such as described in the previous section's example of the environmental variables, which is TCP, one must know that the environmental variable is named DB_PORT_3306_TCP_PROTO.

Also, once you discover the environmental variables, you'll have to parse the various strings for each connection to compose the full connection information, so you may as well just have a single environment variable containing all the host/port/interface information, such as the following:

- DB_NAME=/web2/db

[8] https://github.com/docker/docker/issues/7467

- DB_PORTS=tcp://172.17.0.5:3306,udp://172.17.0.5: 3306

In summary, links are not at the state to easily aid the developer in terms of connecting server and client containers without much premonition of the connection itself, and its stability and benefits just aren't ready to be integrated into mission-critical/production-grade stacks.

An alterative, that is becoming an industry standard is to use a service registration & discovery tool.

5 Ecosystem Overview & Evaluation

This section is intended to be a collection of topics centered on the technologies that categorically are creating an ecosystem on Docker. Specifically, the topics discussed will either have a perspective of what community adoption seems to look like around the technology, or the author's opinion of where the technology stands and how it relates to similar offerings in the market.

5.1 Kubernetes

5.1.1 Community Status

As detailed in "State of the Union: Containers – Part 1," Kubernetes is still the front-runner for how to manage & orchestrate containers in your stack. Though it is still in beta and not production ready yet, the industry has not strayed away from betting on Kubernetes' future and success, solely based on the fact that Google is backing it and the quality and types of contributors working on it are impressive.

Aside from the positive blog posts and community chatter influencing its adoption, the community is showing its vested interested because as of April 2015, Kubernetes is averaging around 400-500 commits per week and has almost 300 contributors – a very substantial following.

5.1.2 Operating Model

There is lots of material covering what Kubernetes' technical capabilities are and what its purpose is, however, we want to take the opportunity to take a step back and give a synopsis of how Kubernetes is intended to be used and where the real purpose on including it in your stack lies.

Kubernetes' added benefit is that it defined a collection of primitives to aid in establishing and maintaining a cluster of containers – in short, Kubernetes is really just an opinionated model for application containers, their dependencies with regards to other resources, and its lifecycle. Without going into too much fine-grained detail, *pods* "are the atom of scheduling, and are a group of containers that

are scheduled onto the same host…[that] facilitate data sharing and communication [by way of] shared mount point's, [and] network namespace [to create] microservices."[9]

The rest of the concepts/units Kubernetes outlines such as *Services*, *Labels* and *Replication Controllers* are ways to enhance *Pods*, as well as serve as a method for the user to declare the intended state their containers should hold and have Kubernetes enforce. Therefore, it is safe to say that Kubernetes does not even know what an application or microservice actually is, just how you wish to collect and manage the containers, including the adherence of requirements such as resource allocation for containers, affinity, replication, load balancing etc. – anything more than that is beyond the scope of Kubernetes.

5.1.3 Kubernetes Specific Functionality

Kubernetes does not directly intend to reinvent core Docker functionality, but it does establish its own slightly divergent semantics for concepts such as volumes and container communication etc., an in some cases, it does have its own implementation of the concept.

5.1.3.1 Volumes

With regards to Docker volumes, we've learned in this report about Data volumes and Data volume containers. With Data volumes, you can either let Docker choose a random, unspecified location for you to create the volume on the host or you can specify a host directory/file to mount on the container. In Kubernetes, new abstractions such as volume and volume mounts exist for usage with pods with volume types known as EmptyDir and HostDir. However, behind the scenes, these types essentially map to the way one can use a Docker Data volume, with additional overhead for Kubernetes to maintain the user's configuration while it interfaces directly with Docker to actually enable the sharing of volumes.

Where Kubernetes does differ from Docker with regards to volumes is in Data volume containers. Docker, as well as various blog posts & articles prescribe this mechanism as the preferred way to share data amongst containers. As we've seen in the previous section on Docker volumes, various intrinsic issues exist when you begin to scale out your architecture, particularly when using and depending on Data volume containers. The folks behind Kubernetes have recognized this methodology as a potential for failure and have chosen to not support Data volume containers as a type in their volumes. The reasoning being that Data volume containers are ultimately passive containers that cannot only be very unintuitive to understand from a user perspective, but could create corner cases and potentially be problematic for management systems.

[9] http://www.slideshare.net/timothysc/apache-coneu

5.1.3.2 Discovery

For container service discovery, specifically in a pod, Kubernetes does not directly use Docker links as they don't do well outside of a single host and managing their lifecycle can prove to be difficult given its current capabilities. Instead, Kubernetes offers two modes of discovery for the concept of *Services* that resemble linking: environmental variables and DNS.

If a Kubernetes *Service* exists, then Kubernetes has a backward compatible mannerism to create Docker-style link environment variables in the container as described in the section on "Container Linking," but remember that we believe they're implicit and hard to work with. It can also create simplified environmental variables with the pattern {SVCNAME}_SERVICE_HOST and {SVCNAME}_SERVICE_PORT.

Another recommended way to perform service discovery is by using a DNS server. The DNS server watches the Kubernetes API for new *Services* and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all *Pods* should be able to do name resolution of *Services* automatically.[10] The takeaway from this is that you don't need to explicitly create links between communicating pods as done natively in Docker because Kubernetes does the heavy lifting, so long as you oblige by the networking mechanisms described in the proceeding section.

Lastly, lets not forget that 3rd party tools such as etcd, Zookeeper and Consul are also viable options.

5.1.3.3 Networking

Kubernetes deviates from the default Docker networking model. The goal of Kubernetes networking model is to allow each pod to have an IP in a flat networking space in which it can communicate with hosts and containers across the cluster. In doing so, pods can be thought of as any other node in the network with regards to "port management, networking, naming, service discovery, load balancing, application configuration, and migration" and can create a NAT-free address space – this concept is known as the "IP-per-pod" model.[11]

Because Kubernetes applies IP addresses at the *Pod* scope - containers within a Pod share their network namespaces - including their IP address. This means that

---

10 https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/services.md#how-do-they-work

11 https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/design/networking.md

containers within a pod can all reach each other's ports on *localhost*. This does imply that containers within a Pod must coordinate port usage, but this is no different that processes in a VM.[12]

We can achieve the IP-per-pod model via the prescribed network requirements imposed by Kubernetes by allocating each host (minion) with its own subnet in an overlay network that can enable containers to communicate with the host and any other networks available in the enviromnet. A common network split is to allocate the overlay a cluster-wide /16 and then divvy up a /24 for each minion. Once you layout your network space, you can implement the overlay and configure a new bridge for the Docker host to use within it. Some tools that are great for this particular purpose, especially in a cloud environment, are container-intended networking technologies such as Flannel, Weave, SocketPlane and even Open vSwitch.

This is different from the standard Docker model. In that mode, each container gets an IP in a host-private networking space defined in RFC1918 (such as in the 172-dot space) managed by the Docker host bridge. The effect of this is that containers can only communicate with other containers on the same host, as opposed to also being able to communicate with other machines in the network as well. Furthermore, it may even be the case that conflicts and confusion arise due to different Docker hosts using the same network space and configuration.

5.2 Mesos vs. OpenStack

With Docker's rise in fame, the ecosystem built on top of it as well as the one that empowers it has introduced technologies, both new and existing, to be considered as options in the stack. Mesos is one of the technologies that have gotten traction in the ecosystem, especially with its recent support of Docker containers and the development of Mesosphere that orchestrates containers on top of Mesos. However, there is confusion about whether one should be integrating Mesos in to their stack or be using OpenStack, or even some other technology to empower the PaaS offering from the infrastructure layer.

In short, people tend to think of Mesos as a competitor to OpenStack, or vice versa; however, it's quite comparable to the apples vs. oranges debate. In fact, one can run Mesos on top of OpenStack, and this tends to be a very common operating model, but if you choose to, you can also just run Mesos directly on bare metal.

The differentiating factor about Mesos and OpenStack is that OpenStack splits up your cluster across several VM's for your applications to run on, and Mesos

---

12 https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/networking.md

combines all of your resources, VM or bare metal, and presents them as a single entity or machine; therefore, one could think of Mesos as one very large machine to run your applications.

The promoters of Mesos argue that VM's were meant to solve the issue of consolidating several, differentiating workloads on far fewer machines than they once needed, ultimately saving money, management of resources and moving away from traditional virtualization which was plagued with horrible turnarounds and bottlenecks – this much is true. In contrast, Mesos to a degree reverses this pattern at the cost of removing full isolation for your application due to the nature of how Linux containers operate versus VM's.

In particular, Mesos recent attention in the Docker ecosystem centers around the concept that greenfield projects are looking to create more of their applications in containers rather than VM's, so designing your infrastructure to run and consume OpenStack seems like overkill and an added complexity when you can achieve the same efficiency from Mesos.

At the end of the day, it really comes down to necessity and planning for the future infrastructure that your applications will consume. Whether that is Mesos, OpenStack, or a mix of the two, will highly depend on how decoupled and/or separate you want to create your datacenter, stack and application to accommodate the nature of varying workloads from different user bases whether it be in VM's, containers, or both.

5.3 Docker Swarm's Impact & Reception

A quick Google search on Docker Swarm primarily reveals sites discussing its announcement, but blogs, articles, and community forums centered on reviews and/or experience with it are almost non-existent compared to other container orchestrating tools. Docker founder Solomon Hykes stated, "Fig [now known as Docker Compose], Kubernetes, Mesos etc. are competing orchestration tools. Docker will give [developers] a standard interface to all three…[and Swarm, formerly known as libswarm]…is an ingredient of that [standard] interface, [specifically, acting as] the glue between Docker and orchestration backends"[13]

If you're taken back or confused by what Swarm's added benefit is, you're not alone. Swarm's attempt is to offer entry-level orchestration for managing distributed containers, but whenever you're ready to adopt another orchestrator that aligns with your goals for scale, you can easily swap out Swarm's engine for the one of your choice.

[13] https://twitter.com/solomonstre/status/492111054839615488

It is the author's opinion that the strategy behind Swarm is that Docker did not want to get into the orchestration battle that is taking place between frontrunners Kubernetes and Mesosphere (which runs on top of Mesos), but rather, they wanted to extend their command line-interface, to an extent, but also be able to interact with the other orchestrating engines as well. This tactic seems to be one that keeps Docker relevant in terms of managing containers, local or distributed, and possibly a vector that Docker plans on productizing and monetizing from in some form in the future, so long as the user doesn't switch directly to the orchestrating backend.

However, as stated previously, the community hasn't really shown much interest in Swarm, and is choosing to consume, learn, interact and design their stacks for their orchestrator of choice directly. What the future holds for Swarm is still to be determined, but the gist of it today does not seem to be offering anything worth adopting.

5.4 OpenShift

RedHat's OpenShift has always been associated with managing your applications at the PaaS layer. With their three offerings: OpenShift Online, OpenShift Enterprise and OpenShift Origin, they aim to claim a stake in various markets. Explicitly, Online competes with Heroku & Google App Engine; Enterprise with CloudFoundry; and Origin is their open source attempt at contributing back to the community while benefiting from the enhancements that open source brings to the table.

With Kubernetes really paving the way as the standard for container orchestration in the Docker world, RedHat has taken the stance with the next release of their products, version 3, to really focus, adopt and build on Kubernetes. In doing so, RedHat is not only betting on Kubernetes as the future of container orchestration, along with the rest of the community, but they are allowing themselves to leverage their Project Atomic, which is their preferred hosting platform for Docker containers that competes with CoreOS.

RedHat has stated that OpenShift is capable of integrating with both OpenStack and Project Atomic, but given the nature of RedHat's previous business models, the OpenStack integration is most likely inline with the intent of OpenShift Origin: to appease the open source community. It is quite the possibility that considering OpenShift Enterprise as a viable option for managing containers in a mission-critical IT shop will require a full top-to-bottom adoption of RedHat's container and virtualization products, if both types of workloads are aimed to be collocated. How OpenShift's adoption of Kubernetes and its intent to be a wrapper for it play out in the long run of the "PaaS war" is sure to be an interesting one.

6 Conclusion

Docker has spurned a slew of technical opportunities to revamp and reconstitute what datacenters and application stacks should look and operate like. There are varying degrees of features and capabilities that span and even spill over from one technology to the other, and noticing these subtleties is a very complex and tedious task. Figuring out which set of technologies will help you establish your future roadmap comes down to the types of users you're trying to enable, as well as classes of workloads you wish to manage across your resources.

It is the author's ambition that this document has provided not just beneficial information with regards to using Docker correctly and efficiently, but that a light has been shined on how these varying tools function, how they're being received in the community, and how they compare with one another.

7 Credits

The following name(s) have reviewed and contributed edits to this document:

- James Thorne – Rackspace