State-of-the-art Application Containers

Andrej Galad, Shivam Maharshi

Introduction

Most of the cloud computing setup nowadays revolves around provisioning of virtual machines, setting up network traffic rules, storage allocation and access filtering (security). All these steps present a large overhead for the user especially in the SOA type of applications. Application Containers (AC) are among alternatives to standard virtualized instances based on the ideas of ultra-fast deployment, flexible portability and superior ease of use. Currently, Docker [1] is the predominant application container with hundreds of companies making use of it and thousands of developers actively participating in its development. However, Docker is no longer the only player on the market. In 2014 CoreOs announced rkt (Rocket) [2], its own implementation of AC, with main focus on the improvement of security (a major issue in the container world) and image distribution, and promotion of container format and runtime standardization.

Motivation

Although there are few tech blogs and news articles stating similarities and differences between Docker and Rocket, these articles tend to neglect evaluation from several important perspectives. Firstly, they do not clearly state a comprehensive list of functionalities supported in these technologies. Secondly, they do not present any quantitative evaluation of performance for both ACs. We feel that a comparative study that provides quantitative and qualitative assessment of supported functionality, performance and container clustering would be beneficial both to the software developers and devops. To the best of our knowledge we are not aware of any such study.

Questions

What are the performance implications on Network, CPU and IO when running Docker versus rkt? How do these ACs behave in distributed big data processing? How do distributed web applications perform when containerized? Which of the two ACs should be preferred for software development, for a given set of requirements? To avoid a vendor lock-in, which of the two ACs must be chosen and why? Performing customizations and implementing additional support is easier in which one of the two ACs? How do these technologies compare in distributed environments? What is the overall support by popular cloud providers? What solutions are available for container clustering and scheduling?

Hypothesis

Given the current state of the two application containers, we believe that Docker is a more logical choice over rkt from the aspect of performance, features, container clustering and community support.

Evaluation Metrics

To perform a fair and scientific evaluation of the two ACs, we decided to create an evaluation metrics shown below. From a list of long metrics that we initially drafted, we have included only a few important and most relevant metric parameters.

#	Criteria	Notes
1.	Ideology & Motivation	Specification standardization and security concerns.
2.	Architecture	Terminology and architecture.
3.	Lifecycle	Startup & Lifecycle of a container.
4.	OS Support	Supported operating systems & package manager support.
5.	Feature Support	Network Types, Volume Mounting, Image Building, Registry Availability, Dynamic Image Creation, Image Distribution, Interactive, Native daemon service. Stop/Restart
6.	Performance Benchmarks	Startup Time, Network Performance, Fedora Benchmark, Wikipedia Benchmark
7.	Clustering	Native cloud platform for running containers on Google Cloud, Amazon, Azure & Kubernetes tool.
8.	Community + Documentation Support	Community support, development and bug resolving pace and documentation support.

Table 1. Evaluation Metrics

Ideology & Motivation

Docker is an application container execution provider and a platform, since it provides numerous utility services compiled into one monolithic binary running primarily as root on a server. Rkt is an implementation of the App Container spec [3], a new set of simple and open specifications for a portable container format, by the development team at CoreOS. There are two major motivational differences behind Docker & rkt. Firstly, rkt follows the Unix philosophy and aims to device the idea of a “standard container” with a simple component, a composable unit, that could be used in a variety of systems. Hence with rkt supports the idea of easy multiple implementations of the App Specification. Docker on the other hand has tightly coupled components without any standard interface thereby making it difficult to customize and have multiple implementations. Secondly, in the Docker process model everything runs through a central daemon with root privileges. Rkt developers believe that this is a fundamental security flaw. Hence to enhance security, every instance of a running container in rkt is given a unique identity, coupled with a lightweight HSM-like service for signing. No existing VM or container environment has a concept like this.

Architecture

From architectural point of view there are a few basic differences between Docker and rkt. Firstly, Docker runs as a centralized daemon with a Client-Server model. Hence all the application containers rely on its continuous support throughout their runtime. Rkt on the other hand follows a standalone model, where all application containers once initiated are self-reliant and do not communicate with any background rkt service.

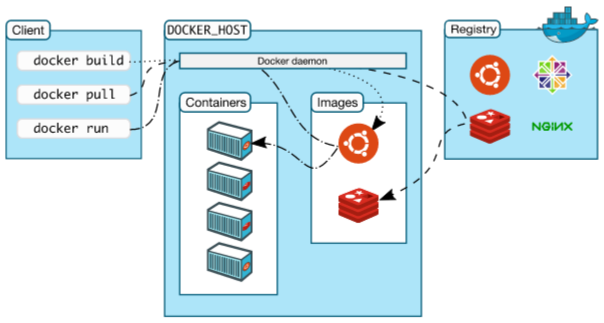


Figure 1. Docker Architecture [4]

Secondly, in Rkt the basic unit of execution are Pods, which are a collection of application containers. All containers in a Pod share the same context in terms of the networking & volume configurations. In Docker the basic unit of execution are Application Containers itself and each container must be initiated by explicitly stating their configurations even if they share their configurations with other containers.

Life Cycle

There are a few differences from Life Cycle point of view between rkt and Docker. Firstly, rkt has linear execution with 4 phases of life cycle [5] - Prepared, Run, Exited Garbage and Garbage. This means that once a container goes to Garbage it can’t be restarted. Docker has a cyclic life cycle which means that a stopped container can be restarted. Secondly, a container in rkt can’t be explicitly stopped but a Docker container can be. This is a conscious decision by the Core OS team, since they believe that containers are a small lightweight execution unit which should be instantiated everytime before use. Thirdly, Docker can perform automatic garbage collection, which means that onces a container is exited, it will be automatically be garbage collected and disk space will be freed. However to free disk space taken by a rkt container, we must explicitly issue a gc command.

Feature Support

A comparison of the features supported by the two ACs can help us make a wiser choice since the tasks to be performed on application containers must be supported by them. The table below shows a few important features supported by these ACs.

#	Feature	Docker	rkt
1.	Runs Docker images	Yes	Yes
2.	Runs App Container images	No	Yes
3.	Architecture	Client-Server	Standalone
4.	Image Distribution via	Centralized Docker Hub	Decentralized over HTTP
5.	Supporting Operating Systems	Linux (with package manager), Mac (via Docker Machine),	Linux (without package manager)

		Windows (only 2016 Server)	
6.	Default enforced Image Signing	No	Yes
7.	Pluggable Isolation	Not supported	Supports pluggable isolation due to 3 staged startup. Fly is a Stage 1 ACI example.
8.	Image Building	Using Dockerfile	With manifest files using actool, goaci, acibuild and docker2aci (for docker images)
9.	In Place Updates	No. Updates requires containers restart.	Yes. Even with Containers

Table 2. Docker v/s rkt feature comparison

Performance Benchmarks

Startup

Our first benchmark models simple container startup speed as ACs proud themselves with their speed of deployment (when compared to VMs). To eliminate external influences on the benchmark we chose one common image for both container technologies - busybox:latest. Our benchmark then consists of sequential startup of 20 containers running busybox that print a message to stdout (echo) exiting immediately. The results are shown below.

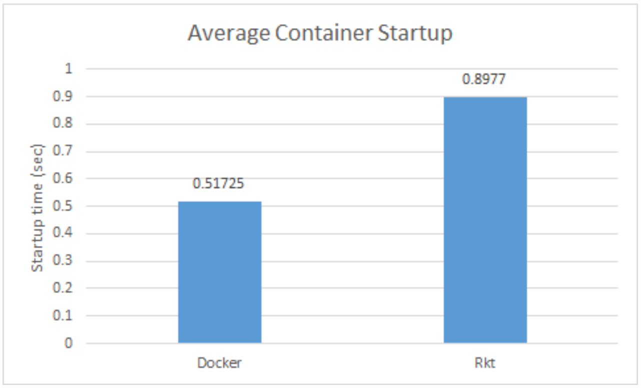


Figure 2. Startup Time Comparison

We measured overall time it took to start all 20 containers using Linux time utility and averaged the result by the number of containers. The results suggest that an instance of Docker can be spinned up almost twice as fast as Rocket. Moreover, further observations and tests suggest that this ratio increases with the increasing complexity of container image.

Network Benchmark

The main aim of running network benchmarks is to evaluate any networking overheads imposed by running applications within containers. We used IPerf3 [6], an industry standard networking benchmark to evaluate network traffic throughput. Our setup included containers running IPerf servers on one host machine connected and IPerf clients running on another host machine. These two machines were connected in a private network via a switch. We generated traffic between clients and servers to capture the network throughput. To generate a real world scenario we had multiple containers running IPerf server. The graph below shows network bandwidth per client as we increased the number of containers serving the requested traffic. Both Docker and Rocket performed close to each.

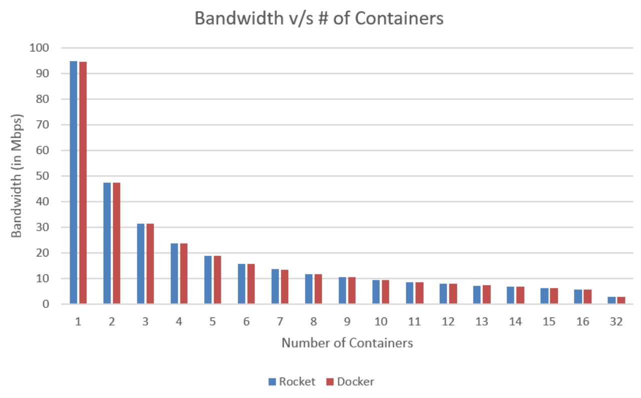


Figure 3. Networking Benchmarking Results

Distributed Processing (Wikipedia Benchmark)

To compare the performance between Docker and rkt over a distributed environment for realistic usage, we performed Wikipedia benchmarking. This involved setting up a Wikipedia Containerized Mirror (Apache Web Server [7], MediaWiki [8] & MySQL [9]) and generating many concurrent HTTP requests to evaluate its performance with metrics like latency, throughput and number of errors received. This had two benefits. Firstly, Wikipedia is a realistic application that is used by millions of people every day. Hence the benchmarking results are actually applicable for real life usage. Secondly, the load distribution of Wikipedia could be prepared accurately because of the huge availability of Wikipedia statistics. Our benchmarking setup involved two configurations - Single Node Four Containers & Four Node Two Containers, as shown below.

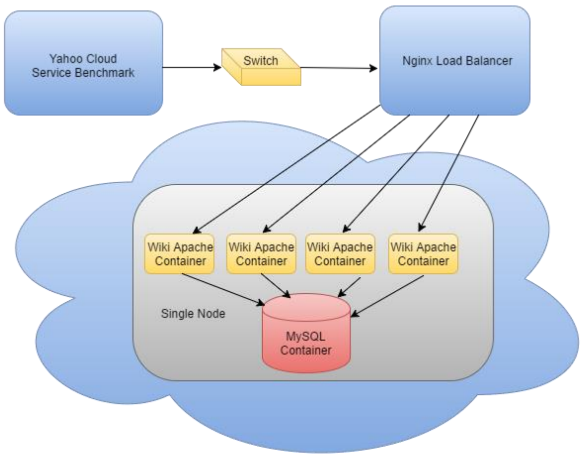


Figure 4. Single Node Four Containers

In the first configuration we had 4 containerized Wikipedia Servers deployed on a single machine, backed by one common MySQL database container. In the second configuration we had 4 containerized Wikipedia Servers deployed over 4 different machines each having one MySQL container for the Wikipedia data. In both the setups, the load generator sent HTTP request to an NGinx load balancing server [10] which further distributed these requests to the Wikipedia containers.

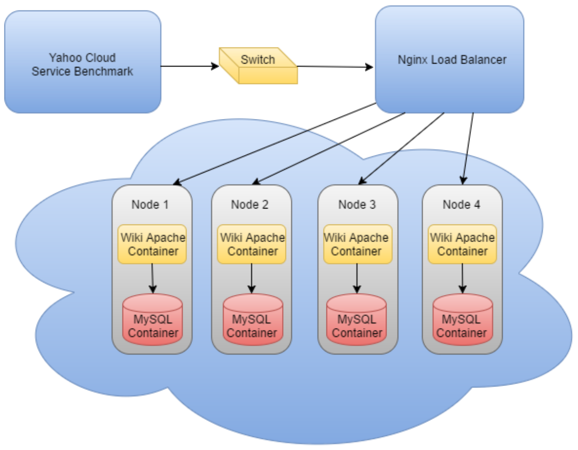


Figure 5. Four Nodes Two Containers

We used Yahoo Cloud Service Benchmark 4 Webservices [11] to generate HTTP workload, which was prepared by analysing the Wikipedia statistics. Both reads and writes had a Zipf distribution with a Zipf's constant value of 0.6175 & 0.6391 respectively. To perform a fair benchmarking, we increased the workload gradually by 25 Requests per Second (RPS), starting from 25 RPS upto 300 RPS, with 60k operations on each step. We did 3 iterations of these benchmarks for both configurations, with each iteration running upto 3 hours. CPU and RAM were monitored throughout the process. The results for average latency, throughput and number of errors (5xx HTTP code) were captured for performance comparison. The graphs for Average Latency versus RPS are shown below for both the configurations.

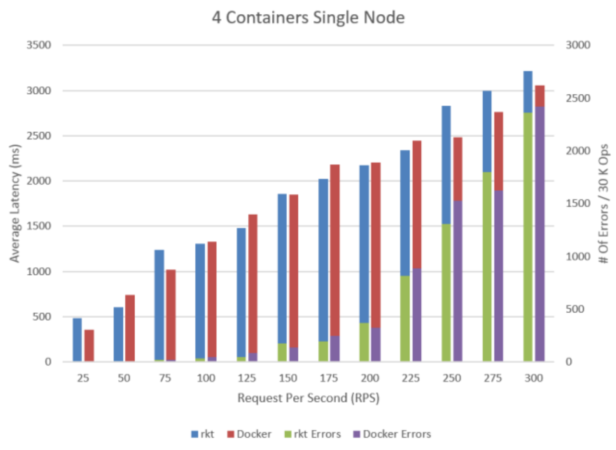


Figure 6. Four Containers Single Node Benchmarking Result

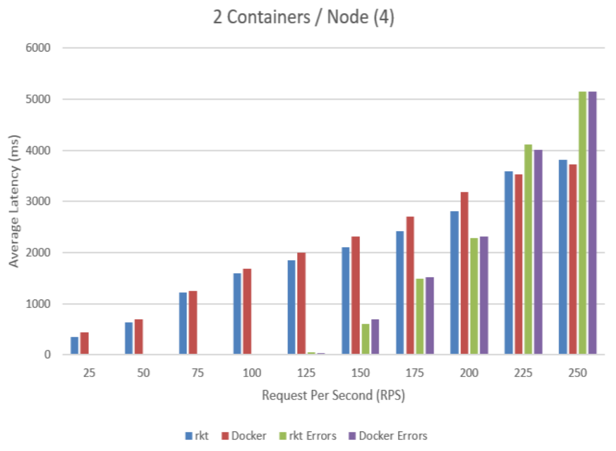


Figure 7. Two Containers Four Nodes Benchmarking Result

The performance of Docker and rkt were pretty comparable in terms of Average Latency, throughput, CPU and RAM usage. Both the containers supported

200 RPS for the first, and 150 RPS in the second configuration. After this point, the number of request exceeding 10 seconds of execution rise rapidly, since the process was CPU bound. We also ran both these configurations with 2 instead of 4 containers, and the results were still comparable. From these numbers we believe that there is no clear winner when it comes to handling the load in realistic distributed server side applications between rkt or Docker. The graphs for CPU & RAM usage, Zipf's constant calculation, benchmarking results with 2 containers are all shown in the Appendix.

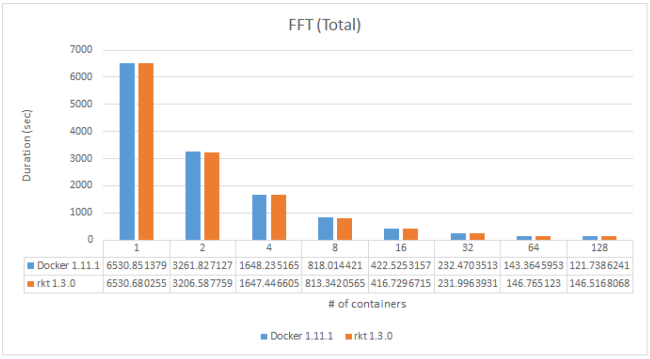
Distributed Processing (Fedora Benchmark)

First of our benchmarks for distributed container setup Fedora Benchmark's main purpose is to evaluate different strategies of work partitioning for various distributed processing workflows. All the benchmark experiments have common denominator - Fedora Repository - popular open-source digital repository platform built on top of Red Hat ModeShape project. The benchmark itself features 3 distinct scenarios modeling 3 different, real-world use cases. Additionally, the benchmark does not generate synthetic data but rather relies on collected sensor data obtained from sensor- and accelerometer-outfitted Goodwin Hall. This raw data is stored in 180 HDF5 files of roughly equal size - 50MB.

We ran Fedora benchmark on Chameleon Cloud service - an experimental environment for large-scale cloud research [12]. Chameleon allows researchers to either provision VMs using its hosted OpenStack KVM or to lease a bare metal high-compute node. For the purposes of the benchmark we leveraged both options - provisioning separate OpenStack VM to run RabbitMQ server, separate VM to run Fedora machine and leasing 4 bare metal machines to host our container workers. For detailed overview of resource setup as well as environment specifications please consult Appendix Figure 1.

Although the full benchmark constitutes of 3 separate workflows emulating 3 different types of data processing - Ingestion (bulk load of HDF5 files into Fedora), Fixity checking (SHA1 checksum comparison) and FFT (highly parallelizable heavy computation) - we decided to only include our results from the last experiment (FFT) as we find it to be the most representative scenario of distributed Mapreduce-like processing. FFT leverages the fact that our stored Fedora objects contain sensor (i.e. signal) data. FFT experiment thus applies fast Fourier forward algorithm (O(nlogn)) to convert said signal data from its temporal domain to its frequency domain. The execution itself is single-threaded promising predictable linear scaling.

We ran Fedora benchmark on both AC runtimes while incrementing the number of containers (and machines) participating in the FFT experiment. Above results correspond to the following allocation of compute resources: 1 machine/1 container, 2 machines/1 container each, 2 machines/2 containers each, 4 machine/2 containers each, 4 machines/4 containers each, 4 machines/8 containers each, 4 machines/16 containers each and 4 machines/16 containers each.



	1	2	4	8	16	32	64	128
Docker 1.11.1	6530.851379	3261.827127	1648.235165	818.014421	422.5253157	232.4703513	143.3645953	121.7386241
rkt 1.3.0	6530.680255	3206.587759	1647.446605	813.3420565	416.7296715	231.9963931	146.765123	146.5168068

Figure 8. FFT Benchmarking Results

Our observed results indeed imply linear performance scaling with doubling of resources between consecutive runs with breaking point between 32 and 64 containers. Both Docker and rkt appear to have comparable performance. To address the breaking point we make use of our per-file average results (figure below).

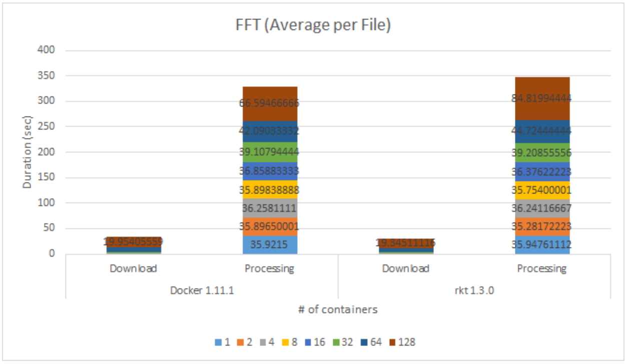


Figure 9. FFT Benchmarking Results

Our average-per-file result graph plots average duration spent by 1 container while processing 1

HDF5 record. Download value corresponds to object extraction while processing refers to actual FFT execution. Average times don't appear to be varying until the limit of 64 containers is reached (performance deteriorates), which, based on our resource monitoring, is caused by excessive CPU context switching (worker nodes only possess 48 cores). Judging from the results, performance-wise rkt seems to be more susceptible to this phenomenon.

Container Clustering

Native Solutions

Without any major surprise Docker takes the lead when it comes to the of-the shelf containerization solutions. Compared to its less popular counterpart Docker as a company puts a lot emphasis on advertising its solution as a full-fledged platform (rather than mere runtime) - essentially striving to become an "AWS of application container world". Thus, Docker offers its custom, out-of-the-box solutions for simple container provisioning (*Docker Machine* [13]), automatic multi-container environment deployment (*Docker Compose* [14]), container cluster orchestration and management (*Docker Swarm* [15]) and even unified web interface (hosted service) for provisioning and management of resources across several popular cloud service providers (*Docker Cloud* [16]).

Rkt on the other hand seems to be focusing its attention elsewhere. In fact at the time of this report CoreOs as a company shadowing the project doesn't envision rkt as anything more than a simple, customizable runtime. As such there exist no native rkt counterparts to above-mentioned Docker solutions.

Turn-key Cloud Providers

Much like in case of native solutions it is only Docker that benefits from 3rd-party cloud integrations. Unsurprisingly, 3 largest and most popular AC clustering solutions - ECS, Docker Swarm, Kubernetes - are offered respectively by 3 largest and most popular cloud providers - Amazon AWS, Microsoft Azure and Google Cloud Platform.

AWS EC2 Container Service is an Amazon service that handles container deployment, management, scheduling, and cluster communication. Behind the scenes ECS really only simplifies provisioning and management of EC2 instances on which Docker containers run - combining several other AWS services such as *EC2*, *CloudFormation*, *EBS*, *Auto Scaling*, *ELB* and *CloudWatch* under one hood - one UI. As ECS really is just an abstraction effectively wrapping other AWS services users only pay for the actual underlying resources (typically only EC2).

Azure Container Service, Microsoft Azure primary clustering platform, is the youngest of all 3 services (until recently only preview release version). Unlike AWS, Azure didn't implement its own clustering system but rather integrated native clustering platform from Docker - *Docker Swarm*. No longer a preview, ACS now provides full support for automatic setup of Docker Swarm (as well as Apache Mesos) via Azure Resource Manager UI, no longer requiring configuration templates [17].

Google Container Engine, much like its Amazon counterpart, features its own container clustering solution managing application containers on Google Compute Engine VMs. It comes as no surprise that GCE is powered by *Kubernetes* - an open source container cluster manager developed by Google. GCE provides uniform environment both for image storage (private image registry) and cluster provisioning.

Kubernetes

Kubernetes is an open-source system for automating deployment, operations, and scaling of containerized applications. We believe that Kubernetes is currently the most mature container clustering platform on the market with largest company adoption and developer/user community. Additionally, the platform features numerous integrations with both cloud providers and custom on-premise infrastructures. Moreover,based on our research Kubernetes is currently the only clustering technology that supports rkt as an alternative container runtime.

Kubernetes comes as a set of scripts that leverage cloud-specific command-line tools to provision resources and register services. By default cluster *kube-up* script points to Google Cloud Engine but can easily be customized via environmental variables to use AWS, Azure, Rackspace or others instead. After the cluster gets spawned the majority of orchestration happens through *kubectl* - command line tool responsible for deployment creation, pod management, scaling+load balancing and monitoring.

With the introduction of Docker to the market Google adapted their clustering platform to use it as a default runtime. This decision came at the cost of additional level of abstraction bridging gap between Kubernetes CLI and discovering services and Docker daemon. However even without direct access through Docker CLI or container definition via Docker Compose (Kubernetes pods require different configuration YAML templates) Kubernetes still manages to make things as simple as possible (essentially 3 commands to create a cluster and provision containerized service). Rkt seemed like a reasonable runtime alternative especially because of its simplicity and essentially the same type of resource isolation (Kubernetes pods = rkt pods) however current state of things still leaves a lot to be desired. For one, Kubernetes master node mainly responsible for service hasn't been made compatible with rkt yet (only worker nodes are). Additionally, explicit changes to *flannel* (common networking service) config on every files are required, which obviously negates automatic setup. Finally, at the time of this report rkt can be only used to run Docker images (via docker2aci tool), which has interesting conversion implications [18].

Ultimately, rkt setup turned out to be more complicated than we'd foreseen. Strong dependency on CoreOS operating system prevented us from creating the cluster both on GCE (resources were provisioned but cluster setup failed) and AWS (CoreOs was declared as unsupported OS).
Interestingly, even CoreOs' custom Kubernetes solution, which was the only reliable way of getting the cluster running, currently hard-codes Docker as its primary runtime. It would appear that at the time of this review rkt integration with Kubernetes isn't the primary focus for either technology (and documentation is simply outdated) [19]. With that being said, rkt promises full integration with Kubernetes by the end of May [20].

Discussion

Over the course of last three months we got a decent idea about Docker and rkt. This section is dedicated to our experience of using these containers. Firstly, when it comes to the developers community support, Docker wins over rkt. Docker has 1366 number of contributors [21] whereas rkt has 130 [22]. This is due to the fact that Docker has been in the market for over 2 years now, whereas rkt only had its first release in Feb 2016. Secondly, since rkt is fairly new in the market, it is comparably less stable than Docker. In last three months we have faced a few major bugs with the latest rkt version. However Docker is fairly stable and no such bugs surfaced in our use case. Thirdly, from documentation support, Docker is fairly strong with both, detailed official [23][24] and numerous unofficial documents (blogs, articles, guides & tutorials). Hence analysing & resolving issues in Docker takes much less time than it does with rkt.

Conclusion

Based on our experience with both ACs, evaluation of their features, benchmarking results and 3rd party support Docker presents itself as a more mature, stable and efficient solution out of both technologies - both for individual developers interested in quick environment setup, and devops focusing on resilient clustering infrastructure. We believe that the major benefit of Docker is its user friendliness. The company keeps investing a lot of effort into gradual elimination of learning curve by maintaining vast and concise documentation, automating setups and configuration and abstracting as much of the underlying logic as possible. Based on our understanding Docker essentially wishes to make containerization technology easily available to anyone (regardless of knowledge) and anywhere (by building a dedicated platform around the product + integrating with external cloud providers). At the time of this report rkt is still a relatively young product (its first version was released in February 2016) yet we think that it is bound to find wider application with the upcoming releases. Although less straightforward, rkt's design promotes simplicity and clear separation of concerns, and also offers a higher degree of customization and control over execution stages. Additionally, long-term, its architecture (pod) makes rkt an ideal candidate to replace Docker as a main execution environment and greatly simplify Kubernetes. All-in-all, rkt definitely deserves its attention from AC user community.

References

1. Docker: https://www.docker.com/what-docker
2. Rocket: https://coreos.com/blog/rocket/
3. https://github.com/appc/spec/tree/master/spec
4. https://docs.docker.com/engine/understanding-docker/
5. https://coreos.com/rkt/docs/latest/devel/pod-lifecycle.html

6. iPerf: https://sourceforge.net/projects/iperf/
7. Apache Web Server: https://httpd.apache.org/
8. https://www.mediawiki.org/wiki/MediaWiki
9. https://www.mysql.com/
10. https://www.nginx.com/resources/admin-guide/load-balancer/
11. https://github.com/shivam-maharshi/YCSB4WebServices
12. https://www.chameleoncloud.org/
13. https://docs.docker.com/machine/overview/
14. https://docs.docker.com/compose/overview/
15. https://docs.docker.com/swarm/overview/
16. https://docs.docker.com/docker-cloud/getting-started/intro_cloud/
17. https://github.com/Azure/azure-quickstart-templates/tree/master/101-acs-swarm
18. http://kubernetes.io/docs/getting-started-guides/rkt/notes/
19. https://groups.google.com/forum/#!topic/coreos-user/g2hKE921BJw
20. https://github.com/coreos/rkt/blob/master/ROADMAP.md
21. https://github.com/docker/docker/graphs/contributors
22. https://github.com/coreos/rkt/graphs/contributors
23. https://docs.docker.com/
24. https://coreos.com/rkt/docs/latest/

Appendix

Throughout the semester we were documenting and sharing our findings & progress on a Google doc. Please find our full report at this link.

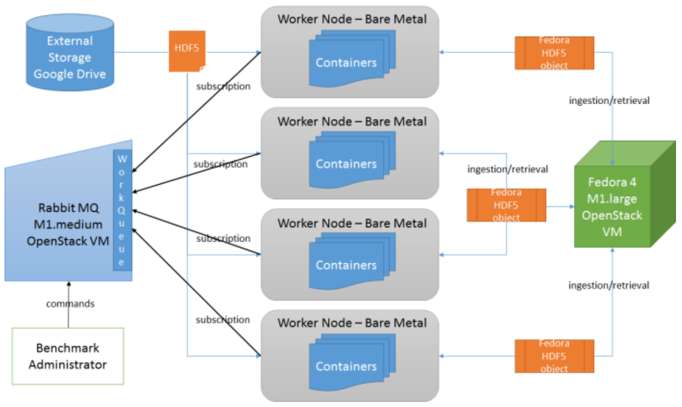


Figure 1. - *Fedora Benchmark Environment Overview*

Figure 2. - *Fedora Benchmark Environment Specs*

Instance Type	m1.large
CPU	Count - 4 Clock Rate - 2299.996 MHz Cache Size - 4096 KB Vendor Id - GenuineIntel Model Name - Intel Xeon E312xx (Sandy Bridge)
Memory	All - 7985 MB
HDD	82.5 GB, 82536112 B
OS	Ubuntu Server 14.04 LTS
Software	OpenJDK 8, Tomcat 7, Fedora 4.5.1

Table 1. Fedora OpenStack VM

Instance Type	m1.medium
CPU	Count - 2 Clock Rate - 2299.996 MHz Cache Size - 4096 KB Vendor Id - GenuineIntel Model Name - Intel Xeon E312xx (Sandy Bridge)
Memory	All - 3953 MB
HDD	41.3 GB, 41251136 B
OS	Ubuntu Server 14.04 LTS
Software	RabbitMQ Server

Table 2. RabbitMQ OpenStack VM

CPU	Count - 48 Clock Rate - 2299.996 MHz Cache Size - 30720 KB Vendor Id - GenuineIntel Model Name - Intel Xeon E5-2670 v3 (Sandy Bridge)
Memory	All - 128705 MB Used - 9751 MB

	Available - 118954 MB
HDD	228 GB, 228992808 B
OS	Ubuntu 14.04
Software	Python 2.7 Docker 1.11.1 Rkt 1.3.0

Table 3. Worker Nodes - Bare Metal

CPU	3.1 GHz Intel Core i5
Memory	16 GB 1333 MHz DDR3
HDD	228 GB, 228992808 B
OS	Ubuntu 15.10
Software	IPerf 3.0
NIC	82540EM Gigabit EC
TCP Window Size	256K
Maximum Transmission Unit	1460

Table 4. Network Benchmarking Setup Specifications

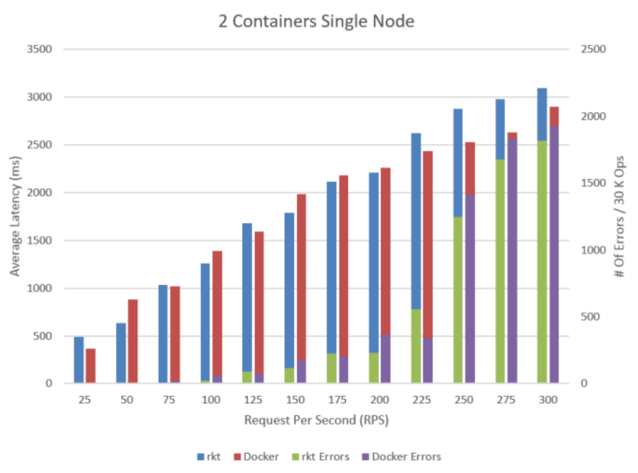


Figure 3. Two Containers Single Node Benchmarking Results

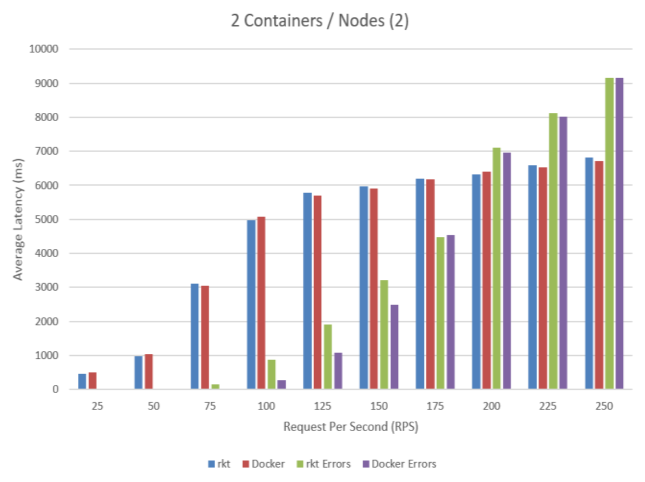


Figure 4. Two Containers on Two Nodes Benchmarking Results

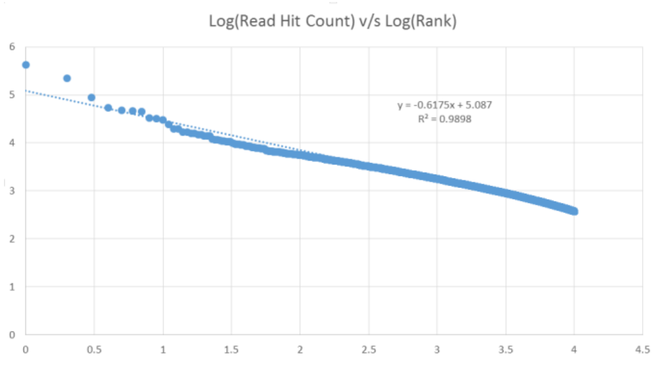


Figure 5. Zipf’s Constant for Read request distribution

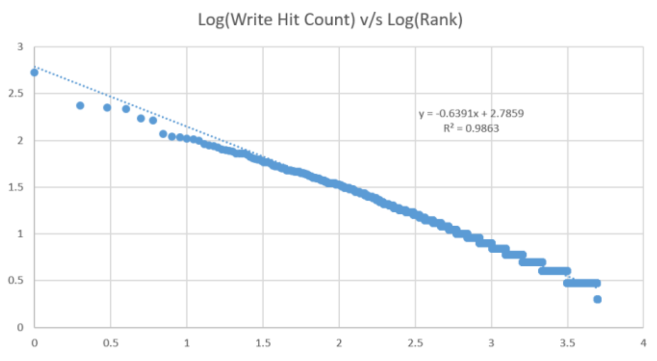


Figure 6. Zipf’s Constant for Write request distribution

YCSB Client Machine	OS X El Capitan 4 x 2.66 GHz Intel Core i5 4 GB 1.06 GHz DDR3
Nginx Load Balancer Server	OS X El Capitan 8 x 3.1 GHz Intel Core i5

	32 GB 1.3 GHz DDR3
Node Server (First configuration)	Ubuntu 14.04 8 x 2 GHz Intel Core i5 16 GB
Node Server (Second configuration)	Ubuntu 14.04 2 x 2 GHz Intel Core i5 4 GB
Software	Apache 2.2, MediaWiki 1.26.2, PHP 5.5, MySQL 5.x

Table 5. Wikipedia Benchmarking Setup Specifications

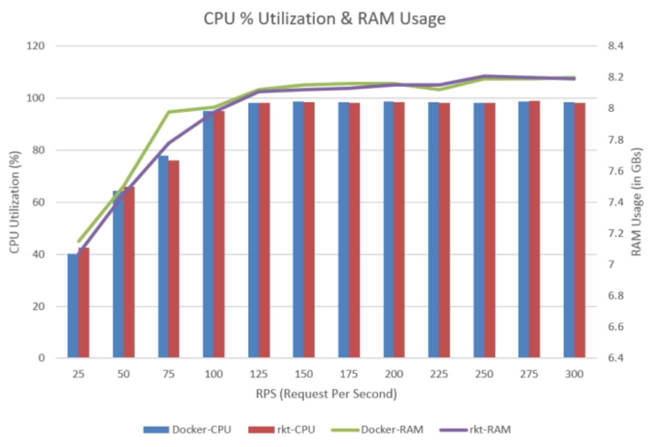


Figure 7. RAM & CPU Utilization for Four Containers on a Single Node