State-of-the-art App Containers

Andrej Galad, Shivam Maharshi

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 and container clustering support.
- In a long term rkt has a potential to catch up with/succeed Docker.

Evaluation Points

#	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	OS & package manager support
5.	Feature Support	Network Types, Volume Mounting, Image Building, Registry Availability, Dynamic Image Creation, Image Distribution
6.	Performance Benchmarks	Startup Time, CPU Compute, Network Performance, File IO Bandwidth, Distributed Processing Benchmark, Load Benchmark
7.	Clustering	Turn key cloud solutions & Kubernetes
8.	Community + Documentation Support	Community size, development bug resolving pace and documentation

Container Overview

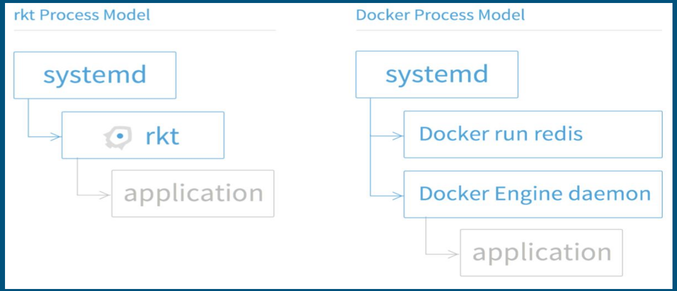
- Ideology
- Lifecycle
- Container Management
- Image Distribution

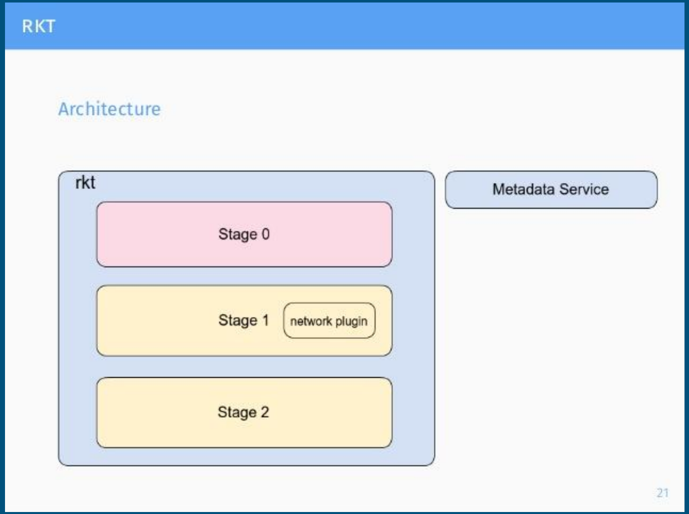
Ideology

- rkt
    - Standard Container - App Container Specification
    - Improved Security - Privilege Separation
    - Easy multiple implementations possible
    - Unix Philosophy - Components Isolation & Clear Integration Points
- Docker
    - Vendor-specific interface
    - Push towards platform - Machine, Compose, Swarm, Cloud...
    - Tightly coupled components - Docker Engine ⇔ Containers

Life Cycle & Startup

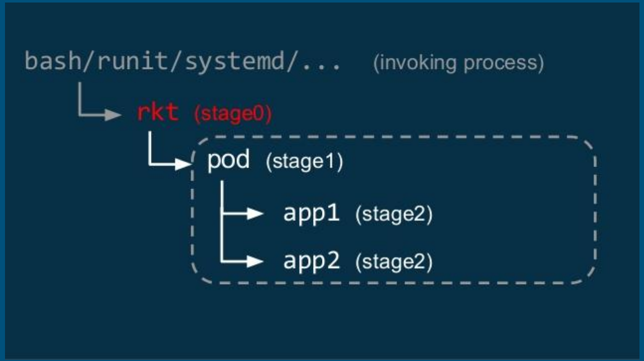
- rkt
  - 4 Phases of container - Prepare, Run, Exited Garbage & Garbage
  - Execution in 3 steps - (Hence the pluggable isolation)
    - Stage0 (stage1 aci, generate manifest, uuid)
    - Stage1 (create cgroups, namespaces pod)
    - Stage2 (runs the fs prepared)
  - Linear execution
- Docker
  - Blackbox - Docker Engine
  - Steps
    - Fetch Image from Docker registry
    - Setup file system with read-write layer
    - Setup networking for docker-host communication
    - IP address is attached to running container
  - Circular execution

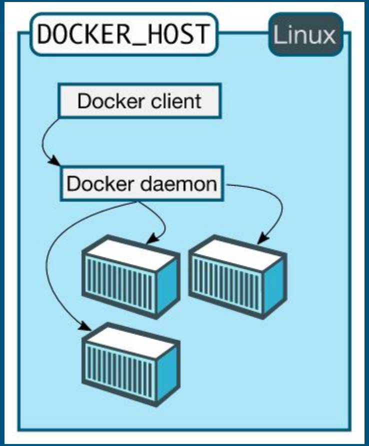




Architecture

- rkt
  - Configures AC namespaces using systemd-nspawn
  - Pods - Basic units of execution (Kubernetes)
  - Apps in a Pod share context
  - Can be updated in-place
- Docker
  - Client-Server model
    - Docker Engine + CTL
  - Containers - orchestrated by daemon
  - Basic unit of execution is an AC





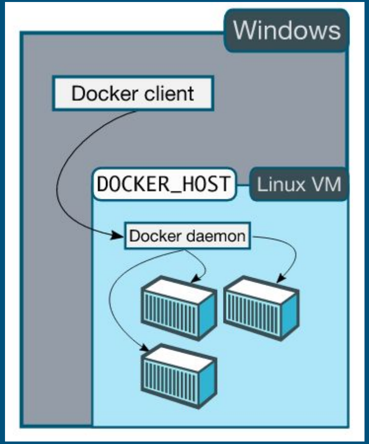
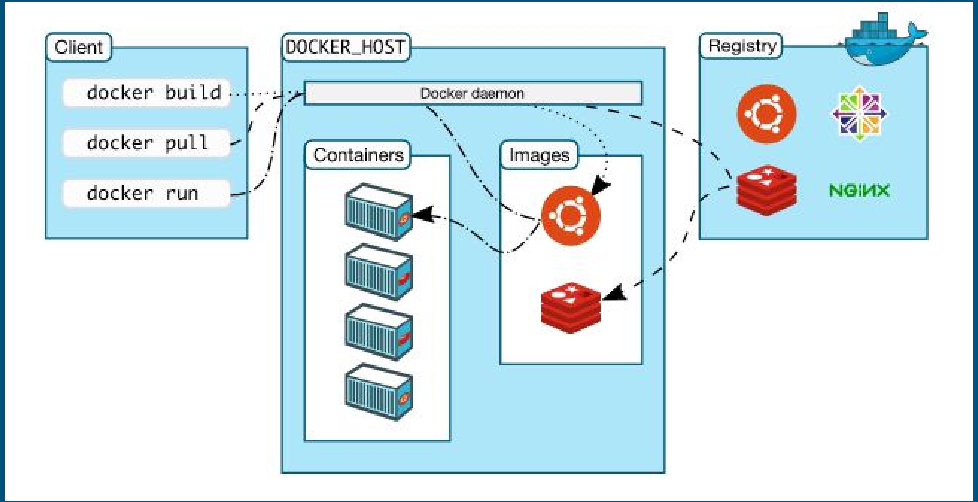


Image Distribution

- Rkt (distributed)
  - Plain tarball images over HTTP
  - DNS discovery of custom namespace & signatures
  - docker2aci
- Docker (centralized)
  - Primary - Docker Hub
  - Secondary - 3rd party registries
    - Cloud providers + private repos



Performance, performance, performance

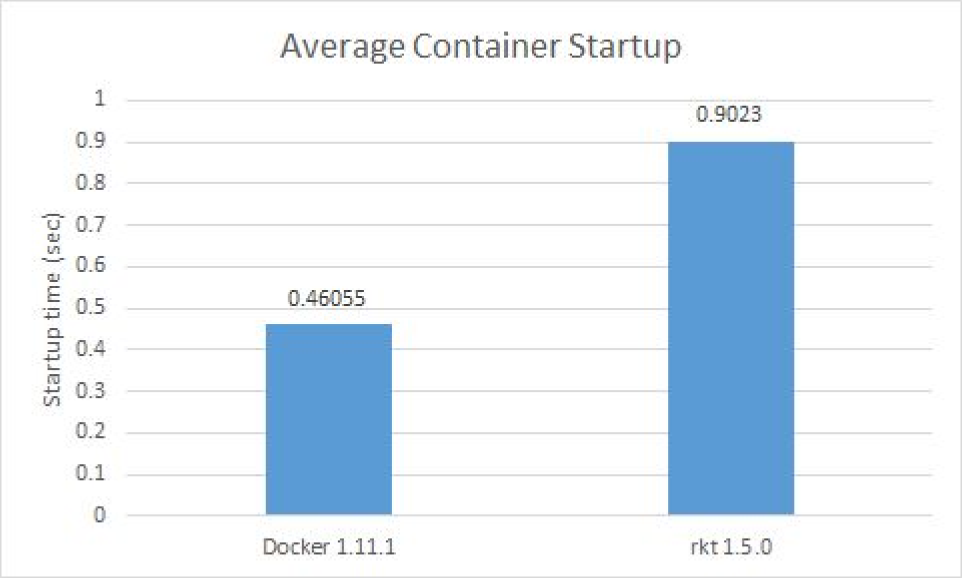
- Startup
- CPU
  - Sysbench, CoreMark
- Network
  - iPerf3
- File I/O
  - fio
- Distributed processing
  - Fedora 4 Benchmark
- Load
  - Wikipedia Benchmark

Performance, performance, performance

- Startup
- CPU
  - Sysbench, CoreMark
- Network
  - iPerf3
- File I/O
  - fio
- Distributed processing
  - Fedora 4 Benchmark
- Load
  - Wikipedia Benchmark

Startup Time

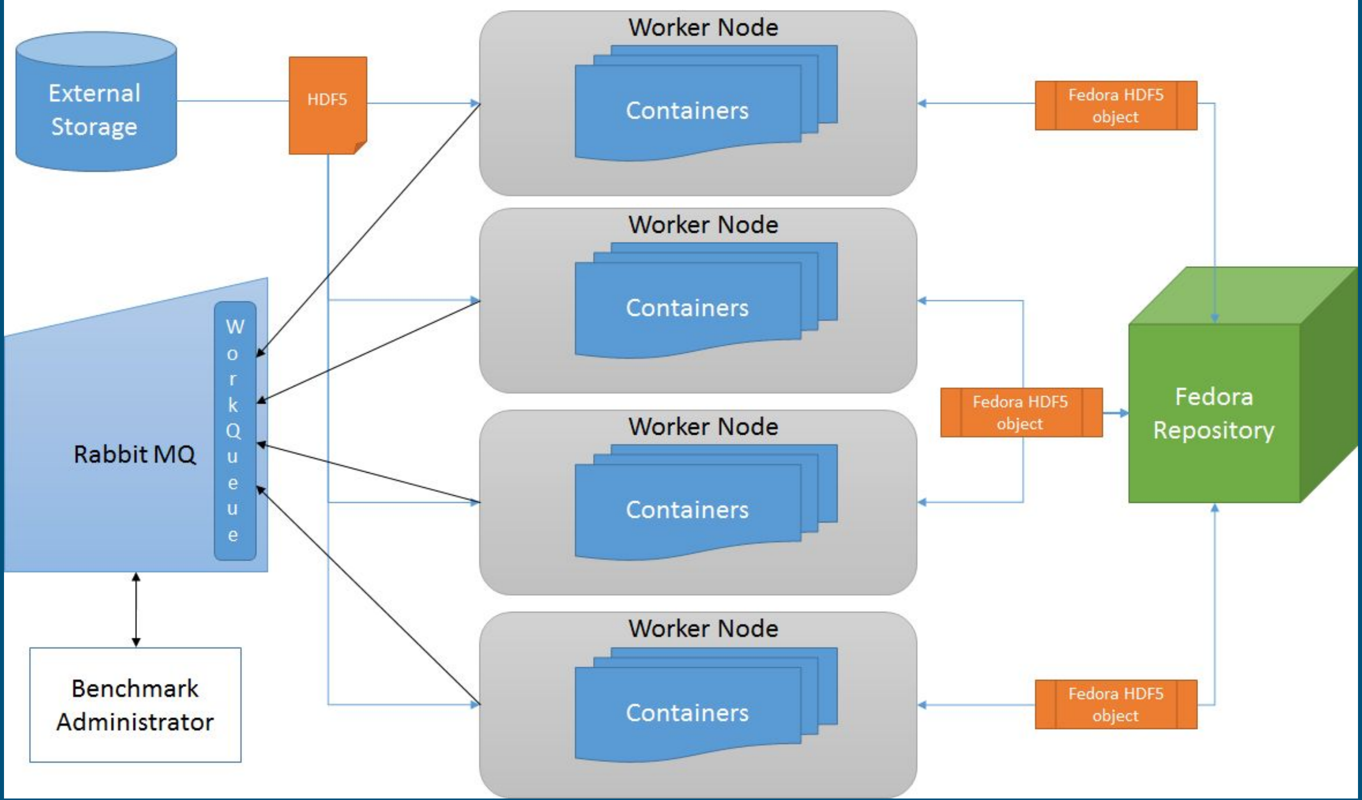
- “Vanilla” machine
  - Docker 1.11.1
  - Rkt 1.3.0 -> 1.5.0
- Image - busybox:latest
- Sequential startup of 20 containers



Distributed Processing Benchmark

- Fedora 4 Repository - http://fedorarepository.org/
  - ModeShape/Infinispan
- Input
  - Goodwin Hall sensor data - HDF5
  - 180 files of approx. 50MB -> 9GB (subset of 2 day data collection)
- Chameleon - https://www.chameleoncloud.org/
  - OpenStack KVM
  - Bare Metal
- Fedora Benchmark - https://github.com/VTUL/VT-Fedora-Benchmark
  - 3 workflows - Ingestion, Fixity checking, FFT

Fedora Benchmark Overview

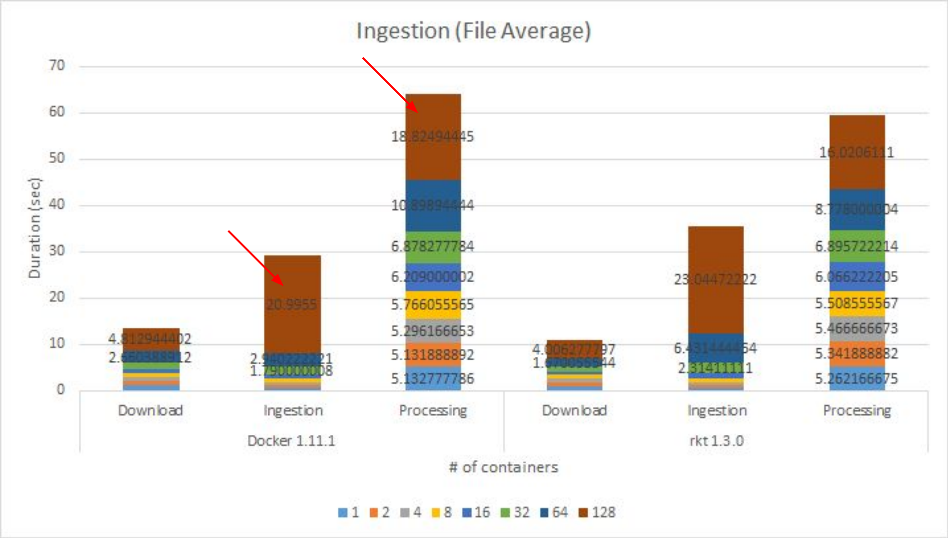


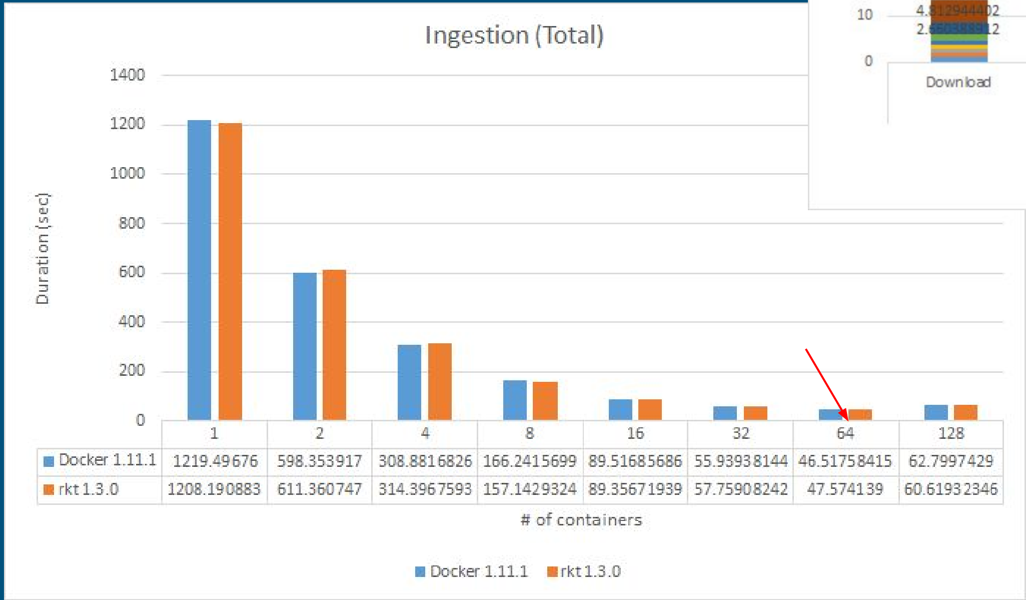
Environment Specs + Setup

- Chameleon OpenStack KVM (Ubuntu Server 14.04 LTS)
  - RabbitMQ
    - m1.medium, 2 VCPUs, 4GB RAM
  - Fedora 4
    - m1.large, 4 VCPUs, 8GB RAM, 80GB drive
- Chameleon Bare Metal (Ubuntu 14.04)
  - 4 Worker Nodes
    - 48 VCPUs, 128GB RAM, 230 GB drive

Phase	# of hosts/# of containers	Total
1	1 host with 1 worker	1
2	2 hosts with 1 worker	2
3	2 hosts with 2 workers	4
4	4 hosts with 2 workers	8
5	4 hosts with 4 workers	16
6	4 hosts with 8 workers	32
7	4 hosts with 16 workers	64
8	4 hosts with 32 workers	128

Ingestion



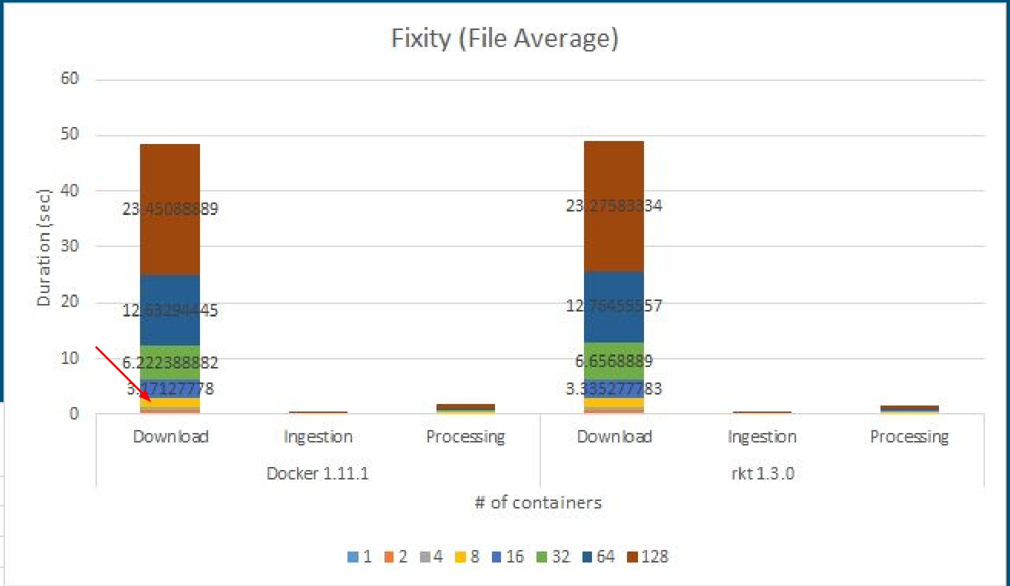


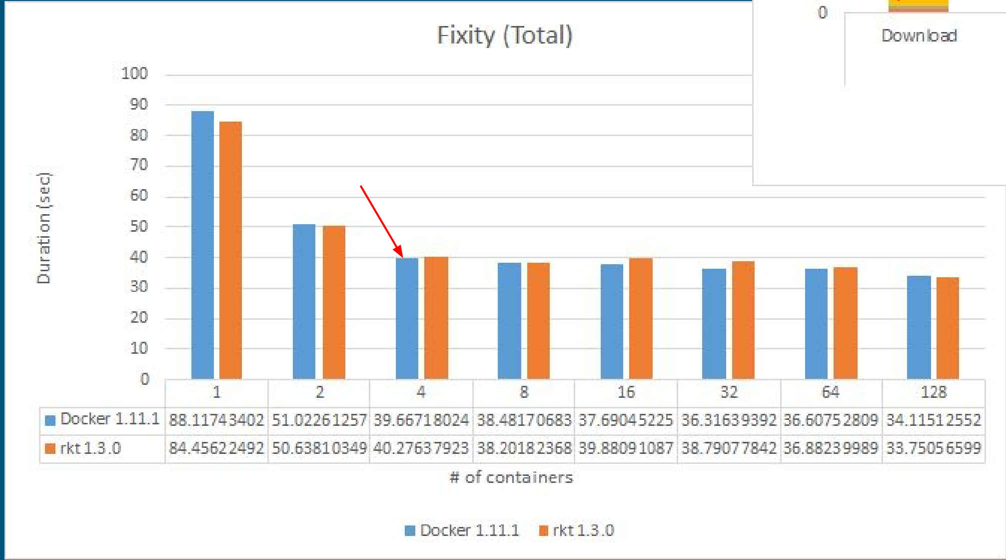
	1	2	4	8	16	32	64	128
Docker 1.11.1	1219.49676	598.353917	308.8816826	166.2415699	89.51685686	55.93938144	46.51758415	62.7997429
rkt 1.3.0	1208.190883	611.360747	314.3967593	157.1429324	89.35671939	57.75908242	47.574139	60.61932346

Steps:

1. Download HDF5 file from external storage - Google Drive
2. Extract metadata/headers using FITS
3. Create Fedora object and ingest data

Fixity Checking



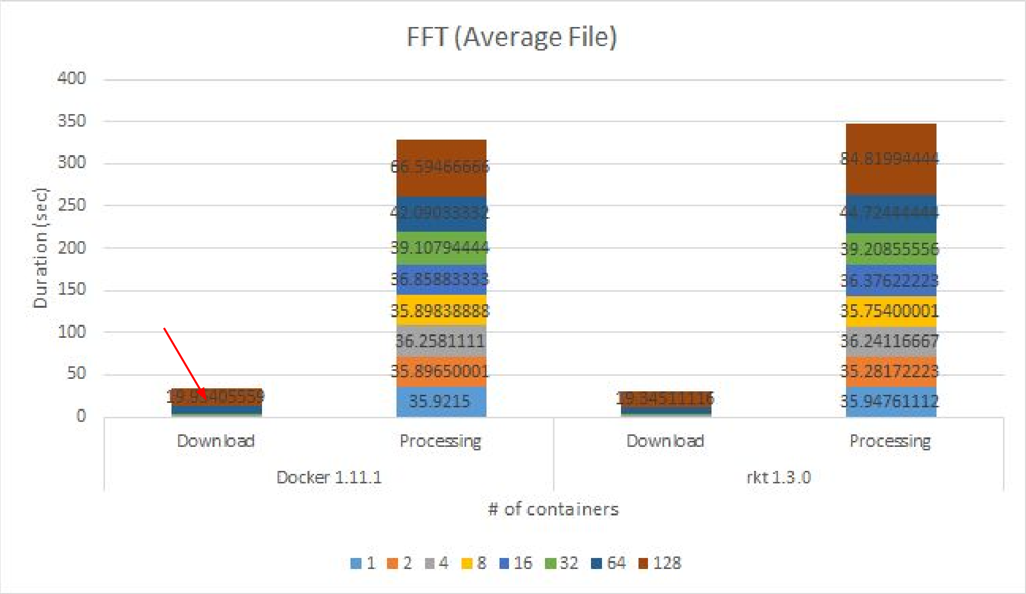


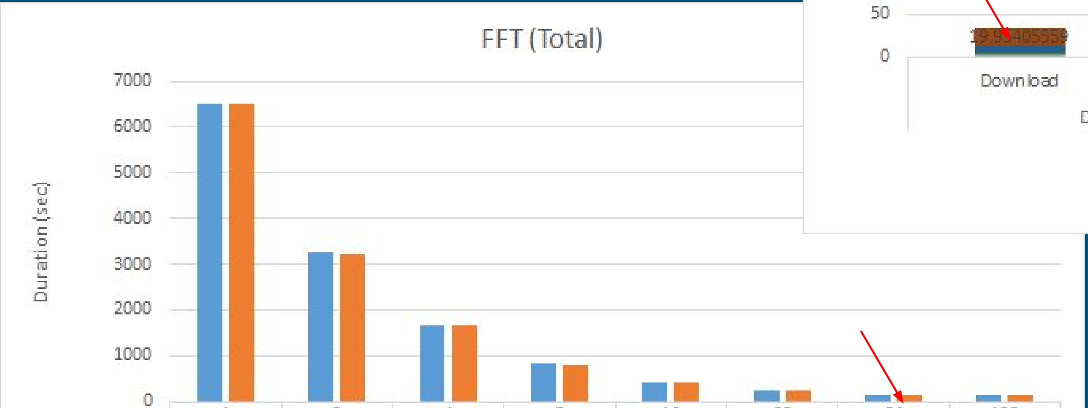
	1	2	4	8	16	32	64	128
Docker 1.11.1	88.11743402	51.02261257	39.66718024	38.48170683	37.69045225	36.31639392	36.60752809	34.11512552
rkt 1.3.0	84.45622492	50.63810349	40.27637923	38.20182368	39.88091087	38.79077842	36.88239989	33.75056599

Steps:

1. Fetch HDF5 checksum from Fedora
2. Reconstitute Fedora object and generate checksum
3. Compare and store result as object metadata

Fast Fourier Transform





	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

Steps:
1. Reconstitute Fedora object
2. Compare FFT (numpy.fft - n*log(n))

2nd run - Setup

- Bring everything together…
- Chameleon Bare Metal
  - 48 VCPUs, 128GB RAM, 230 GB drive
- Fedora 4 + Benchmark + RabbitMQ
  - No network isolation

Phase	# of containers
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

Distributed Load Benchmark - Server Side

- Two configurations
  - Single Node Multiple Containers
  - Multiple Node Two Containers
- Wikipedia Server
  - Realistic Application
  - Realistic Workload
- Yahoo Cloud Service Benchmark
  - Custom Web Module
  - Link: https://github.com/shivam-maharshi/YCSB4WebServices

Single Node Multiple Containers

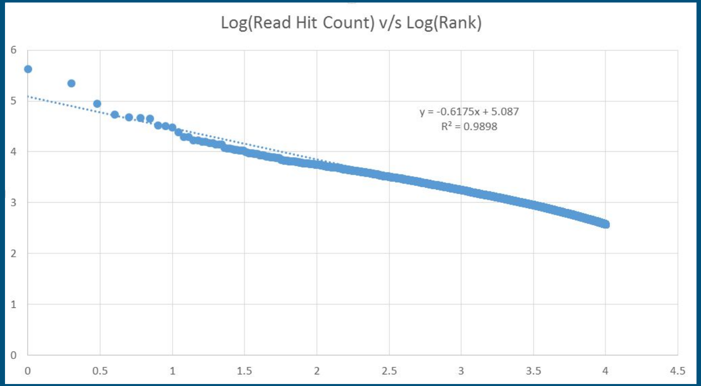
- YCSB
  - OS X El Capitan
  - 4 x 2.66 GHz Intel Core i5
  - 4 GB 1.06 GHz DDR3
- Nginx Load Balancer
  - OS X El Capitan
  - 8 x 3.1 GHz Intel Core i5
  - 32 GB 1.3 GHz DDR3
- Node
  - Ubuntu 14.04
  - 8 x 2 GHz Intel Core i5
  - 16 GB
- Version
  - Apache 2.2, MediaWiki 1.26.2, PHP 5.5, MySQL 5.x

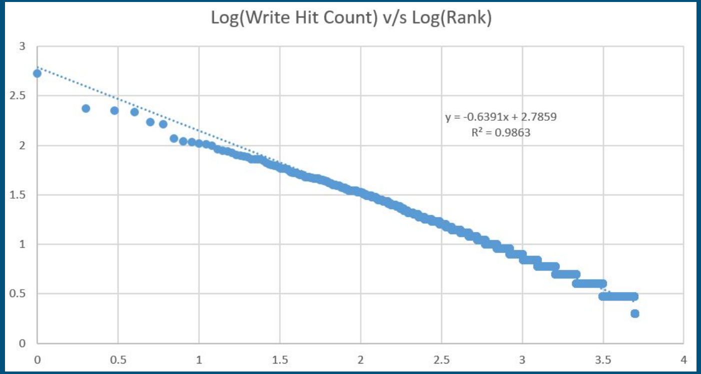
Multiple Node Two Containers

- YCSB
  - OS X El Capitan
  - 4 x 2.66 GHz Intel Core i5
  - 4 GB 1.06 GHz DDR3
- Nginx Load Balancer
  - OS X El Capitan
  - 8 x 3.1 GHz Intel Core i5
  - 32 GB 1.3 GHz DDR3
- Node #
  - Ubuntu 14.04
  - 2 x 2 GHz Intel Core i5
  - 4 GB
- Version
  - Apache 2.2, MediaWiki 1.26.2, PHP 5.5, MySQL 5.x

Wikipedia Workload

- Trace Generation
  - Greek - 400k pages - 4GBs
  - 1 Month Trace - Jan 2016
  - Read = 2.8 Billion
  - Write = 60 K
  - Sort URLs in Descending Hits
  - Read & Write - Zipf's Distribution
  - Read Zipf's Constant = 0.6175
  - Top 10k pages, R2=0.9898
  - Write Zipf's Constant = 0.6391
  - Top 5k pages, R2=0.9772

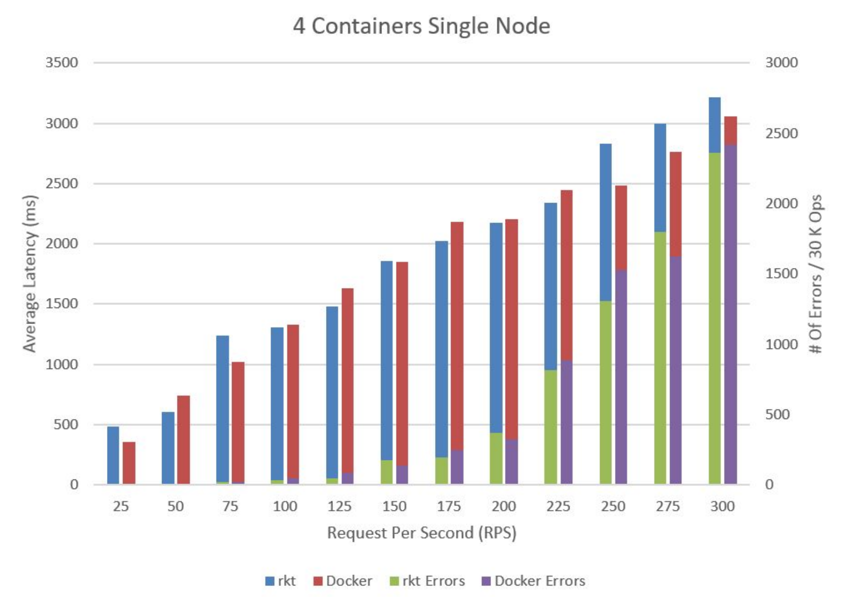


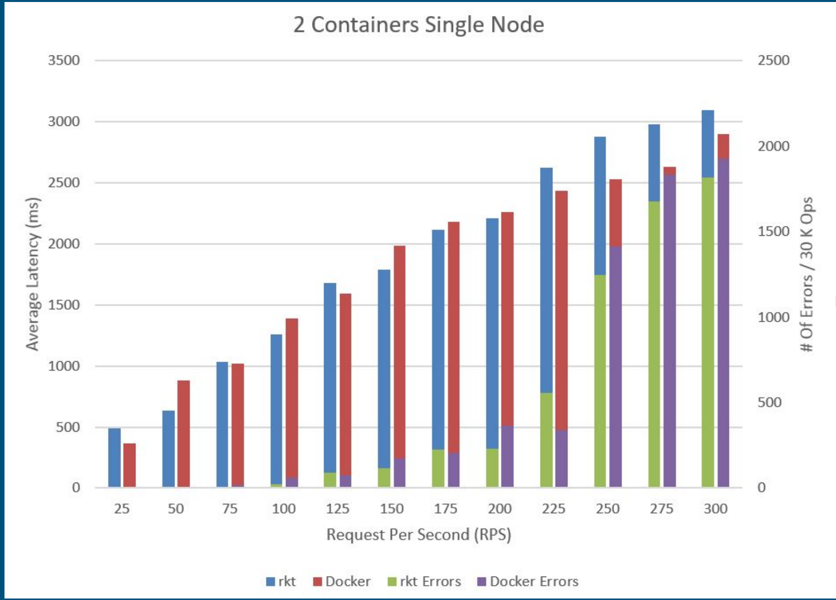


Benchmarking

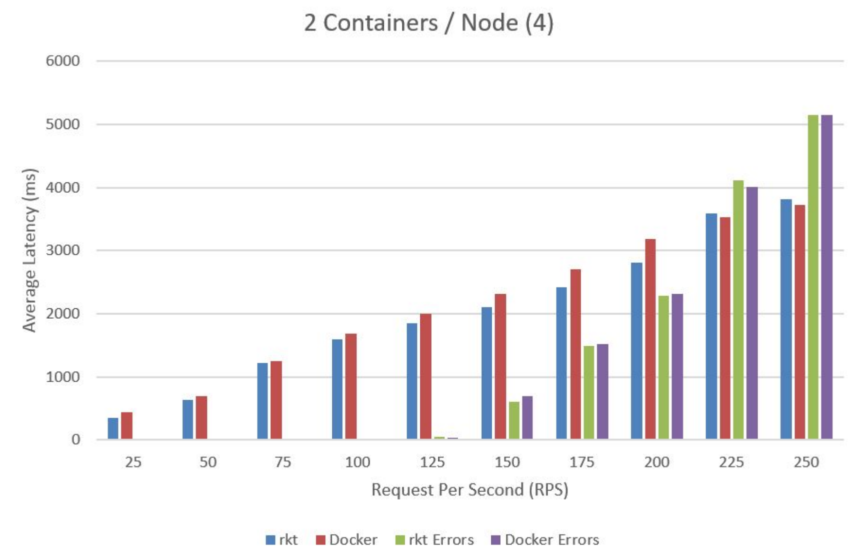
- Yahoo Cloud Services Benchmark
    - Read & Write Trace
    - Parameters - Zipf’s K, R/W Ratio..
    - Execution Timeout = 10 secs
- Step Benchmarking
    - Incremental Workload - In steps of 25 RPS
    - Sufficient Duration - 2 Hours / Conf
    - Resource Monitoring - CPU, RAM & Network
- Metrics
    - Latency
    - Throughput
- Bottleneck
    - CPU Bound

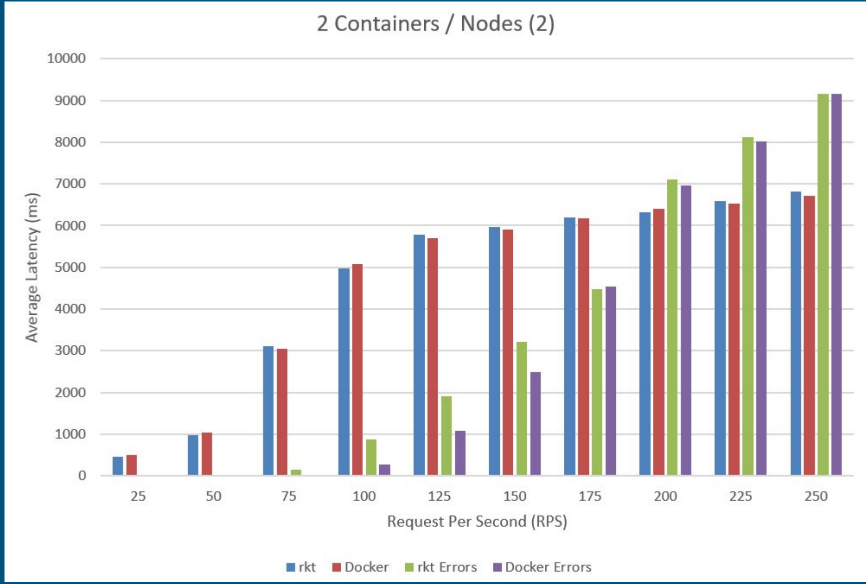
Single Node Multiple Containers





Multiple Node Two Containers





Cluster it up...

- Turn-key Cloud Solutions - rkt though luck :(
  - EC2 Container Service - custom solution
  - Azure Container Service - Swarm (NO LONGER PREVIEW :D)
  - Google Container Engine - Kubernetes (gcloud, kubectl)
- Kubernetes
  - Docker - main runtime (even for CoreOs)
  - Rkt - can be configured as runtime but...
    - Only nodes
    - No automation
  - http://146.148.35.100:8080/
- Rktnetes - rkt 1.7.0

Conclusion

- Docker - still no.1 container on the market
  - Deployment/provisioning, OS availability, startup time, features, 3rd party support, platform…
  - Both developers and ops
- Rkt’s main potential - simplicity and modularity
  - Emphasis on customizability
  - Adherence to a common standard
  - “Made for Kubernetes”