MESOSPHERE

DC/OS Upgrade FAQ & Workbook

Guidance for DC/OS Upgrades

July 24, 2018

Arthur Johnson,

Solutions Architect, Mesosphere

[*ajohnson@mesosphere.io*](mailto:ajohnson@mesosphere.io)

Getting Started

Overview of Upgrades

Upgrades are some of the most difficult tasks in running a production environment-- the scope of changes can be tremendous, and frequently, service interruptions are incurred as code changes, fixes and features are introduced into a live ecosystem of software. Upgrading DC/OS and ancillary software components is no different than upgrading underlying Operating Environments and great care and consideration must be taken to ensure that the environment has the best possible outcome.

One very important consideration, which is important enough to mention in the overview, is that currently Mesosphere has a N-2 release support methodology. That is, software which is two versions older than the current shipping product is still within support. Older than that, it is considered end-of-support (EOS) and nearing end-of-life (EOL). While this may not be of utmost importance in some environments; if future long-term stability, growth, functionality and capacity are goals of the organization and environment, upgrades must be conducted.

This document will deal with recent (aka modern) versions of DC/OS starting with the 1.9.x version following through to the current shipping version, 1.11.3.

Additionally, this document is not a replacement for the standard upgrade documentation / procedures, but meant to augment existing best practices. For actual upgrade steps/procedures please reference the Version specific steps found in this document.

Acceptable and Qualified Upgrade Paths

Understanding the Release Version Numbers

DC/OS release Numbers are formatted as:

`<Major>.<Minor>.<Patch>`

As such `1.10.2` is the Version 1 of the Product, Minor release "10", and Patch "2". At this time, Mesosphere does consider `X.<Minor>.Y` releases a "Major" release/increase in functionality. With that understood, `1.10.X -> to -> 1.11.X` is considered a major release. Once DC/OS crosses the `1.x.y to 2.x.y` release stream minor releases are considered "Major". Although this may seem a digression, it is critical to note the X.Y.Z where "Z" is a patch is considered 'not an upgrade' but a patch which can be applied to the cluster in an in-service fashion.

Understanding Supportability of Releases

Currently DC/OS supports a `N-2` set of releases, where `N` is the version and "2" is back two versions. With this stated, if the current shipping version is `1.11.3` we continue to support, fix, and qualify versions `1.11.3`, back to `1.9.3`. Support engineering efforts will continue this supportability perspective with the following effort:

- Fix the bug in the version it was found, and all forward versions, as long as the affected version is currently supported
- Backport customer fixes from master `<ver> to <ver>-2`
  - E.g. if the current master is `1.11.3`, issues will be fixed in `1.11; 1.10; and 1.9`

If an issue needs to get backported to <ver>-3 it can be explicitly asked for and efforts to resolve the issue will be considered in planning

Acceptable and Qualified Steps and Procedures

A complicated layered solution such as a cloud environment cannot be easily decoupled and upgraded without impacting other layers of the environment. Care must be taken to ensure that operations continue to run, or, (preferably) a separate environment is constructed to build a replica of the current solution with the latest (or chosen) versions of code.

Where do you Start?

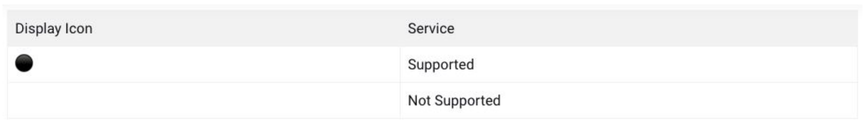
Do we upgrade the operating environment, Docker, or DC/OS first? In most cases it is prudent to consider upgrade of the operating environment first, selecting the version of Docker which is either supported or bundled with that version, then qualifying (up-the-stack) the version of DC/OS which is to be deployed and/or upgraded.

If the core of the solution runs on DC/OS, it is best practice to begin qualification at that level, and reference which subcomponents are tested and qualified to work in a compatible, supported manner.

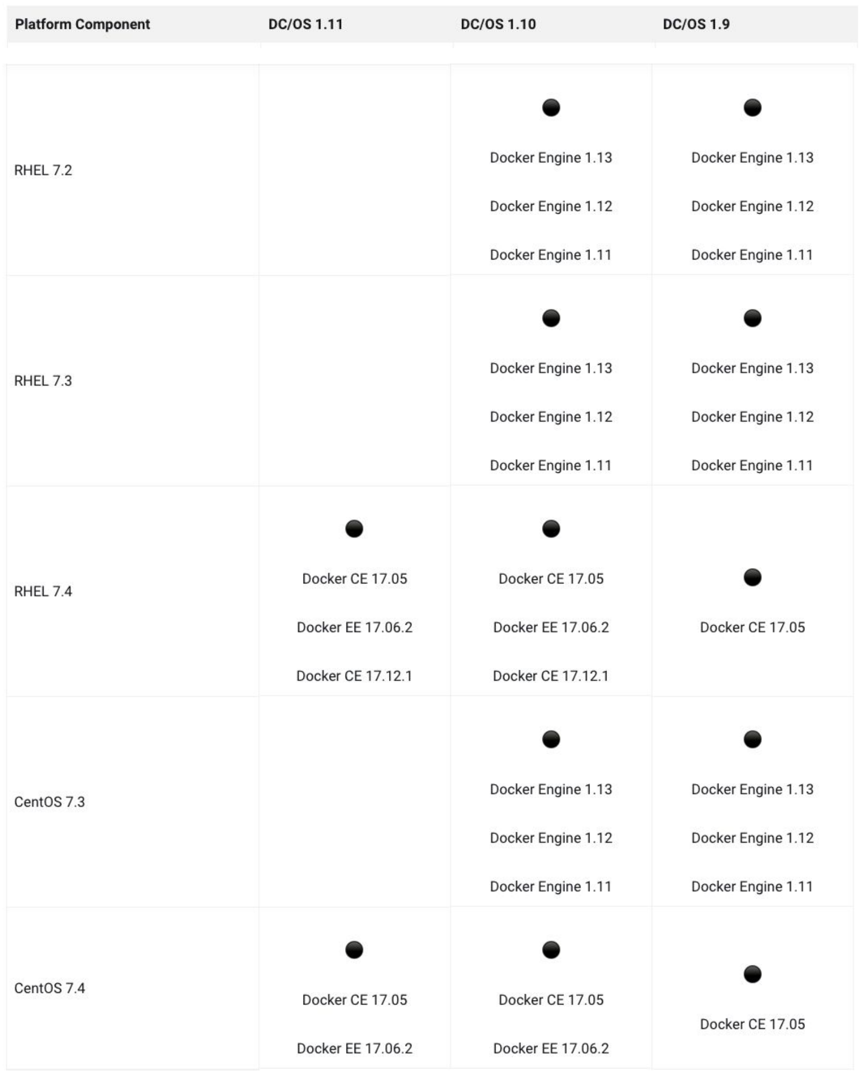
Mesosphere publishes system requirements for DC/OS Deployments which can be found here:
https://docs.mesosphere.com/1.11/installing/ent/custom/system-requirements/

Additionally, Mesosphere publishes a “Version Policy” for the software in the entire stack and has qualified and tested these versions as known to be compatible. The Version Policy can be found here:
https://docs.mesosphere.com/version-policy/

Version Policy



Display Icon	Service
●	Supported
	Not Supported



Platform Component	DC/OS 1.11	DC/OS 1.10	DC/OS 1.9
RHEL 7.2		● Docker Engine 1.13 Docker Engine 1.12 Docker Engine 1.11	● Docker Engine 1.13 Docker Engine 1.12 Docker Engine 1.11
RHEL 7.3		● Docker Engine 1.13 Docker Engine 1.12 Docker Engine 1.11	● Docker Engine 1.13 Docker Engine 1.12 Docker Engine 1.11
RHEL 7.4	● Docker CE 17.05 Docker EE 17.06.2 Docker CE 17.12.1	● Docker CE 17.05 Docker EE 17.06.2 Docker CE 17.12.1	● Docker CE 17.05
CentOS 7.3		● Docker Engine 1.13 Docker Engine 1.12 Docker Engine 1.11	● Docker Engine 1.13 Docker Engine 1.12 Docker Engine 1.11
CentOS 7.4	● Docker CE 17.05 Docker EE 17.06.2	● Docker CE 17.05 Docker EE 17.06.2	● Docker CE 17.05

Upgrade Paths for DC/OS

If an environment was running DC/OS 1.9.4, and decided to upgrade their current environment(s), the minimum upgrade path should be to the latest release of DC/OS 1.9.x. Ideally, an upgrade would be made to the later version families, either 1.10.x or preferably 1.11.x. Below are some potential use-case upgrade path scenarios:

Upgrade Paths

Scenario 1: Customer wants to make sure their current 1.9.4 cluster is at the latest:

Current Version	*Path-to-Current*	*Target-end-Version*
`1.9.4`	`1.9.4->1.9.9`	`1.9.9`

Scenario 2: Customer wants to upgrade their 1.9.4 cluster to 1.10.8 for fixes and functionality, but not to 1.11 because they want to independently test that version prior to running the latest code:

Current Version	*Path-to-Current*	*Target-end-Version*
`1.9.4`	`1.9.4->1.9.9->1.10.7`	`1.10.7`

Scenario 3: Customer wants to completely upgrade their 1.9.4 cluster to the latest shipping version (at the time of this writing), to 1.11.3:

Current Version	*Path-to-Current*	*Target-end-Version*
`1.9.4`	`1.9.4->1.9.9->1.10.7->1.11.3`	`1.11.X(3)`

Things to Know Before Beginning

This is a question and answer section, before going into the details of how to upgrade. In complicated scenarios, where a cluster may be several versions behind, or out of support, corners may look to be cut. ***No. Corners. Can. Be. Cut.*** Asking to circumvent time consuming upgrade paths is actually asking to take much longer time to recover a failed or lost environment. It's not a matter of whether the upgrade will work, or the cluster will crash, *it's a matter of keeping tasks and applications in service* as you upgrade the environment.

FAQ Regarding Upgrades

Question	**Answer**	**Reasoning**
Our cluster is running 1.9.0 and we want to upgrade to 1.11.1. Can we do this in one event, in the current cluster, retaining all tasks?	Absolutely Not.	There are many minute iterative changes between the versions and jumping several versions at once will leave components in a state where they are not reachable or usable.
Our cluster is running 1.9.1 and we want to upgrade the Master nodes to 1.10.1 and leave the agents at 1.9.1. Is it possible?	Yes.	This is possible for a period of time- it is best practice to upgrade the agents to the same DC/OS - Mesos release as the masters. Mesos supports a N-1 (current release, back one version) compatibility between Masters and agents.
We want to upgrade, but intend on doing the agents first, then the masters, is this acceptable?	No.	Mesosphere's best practice is to upgrade Masters first, then agents. Upgrading agents first may end in a scenario where agents are not registered with the masters and then experience various issues supporting running workloads and unable to deploy new workloads.
Our cluster is running 1.10 and we want to get to 1.11.3. Is this a one step upgrade?	No.	Unfortunately, this is not a one step upgrade. We qualify latest patch release of a version upgrading to the next major version. In this scenario you must follow this upgrade path: 1.10.x->1.10.8->1.11.3.
What are the recommended rollback procedures for a cluster upgrade in the event that it does not work?	None.	Currently there is no rollback procedure. Utmost care must be taken to consider the upgrade and then troubleshoot accordingly. To rollback is tantamount to a re-install at the desired version (previous version).
In our path to upgrade our cluster from 1.9.2 to 1.11.3 we want to move the masters iteratively through the	No.	While this would seemingly work, remember that Mesos supports N-1 compatibility. Following the logic in

versions to 1.11.3 and then do the agents in one upgrade. Can this be done?		this question would lead to an environment where the agents are out of compatibility/support with the masters and running tasks may be lost, and new tasks would not be deployed.
Is there the concept of an in-service upgrade where we export meta-data from the cluster, upgrade the software, then import the data and be up and running again?	No, not currently.	Currently not possible due to the changing landscape of back-end components and sub-systems currently used and in review within DC/OS.
We want to upgrade our cluster and make configuration changes at the same time to config.yaml, can this be done?	It depends, but no, not supported.	In test, it may be found, that this does work- to change a configuration parameter and upgrade at the same time. Best practice is to either make a configuration change, or to upgrade the cluster. Not both at the same time.
We are going to upgrade the cluster next week and one of the master nodes is done, can we upgrade the cluster with the master missing?	Not recommended.	No. Any and all issues must be rectified prior to beginning the upgrade. Trying to resolve existing issues after an upgrade only increases complexity.
During our upgrade, we want to add additional agent nodes, is this possible?	It depends	Best practice is to either add them before the upgrade, at the current version, or to install them as new nodes at the end of the upgrade on the upgraded version of code.
During our upgrade, we want to replace and/or add masters, is this possible?	No	For a variety of reasons this is a particularly bad idea. Specifically speaking, masters cannot be added to the cluster without making an entire configuration change to *every node in the cluster*. Best practice would be to address any issues before the upgrade.
Upgrading our cluster through all versions from 1.9.1 to 1.11.3 is too many steps, can we create another cluster and move applications over?	Yes.	Yes, this actually is the quickest path to resolution in scenarios where many versions have passed and need to be upgraded <thru> to get current. Application and integration testing must be thorough in this scenario to ensure that all tasks will be successful

		and stable running in the latest version.
We want to upgrade our cluster and at the same time move between permissive to strict mode. Is this possible?	No.	Either upgrade the cluster first, or change the security mode. Do not do both at the same time.
We want to upgrade the core DC/OS platform to version 1.11.3 but have a mix and match of different component versions, is this possible?	It depends	DC/OS comes bundled with versions of Mesos, Marathon, and other sub-components. These versions are carefully selected as part of the build-- while there may be extenuating circumstances which drive use of other versions, best practice is to use the versions bundled. There is limited compatibility of components between DC/OS versions and best practice for this type of deployment is to engage with Mesosphere services/support.
Are upgrades the same as patching?	No, not really.	They follow the same implementation methodology but the scope of changes is drastically different.
Will everything (application wise) I have configured in my cluster running in 1.9.2 work in 1.11.3?	It depends	There were feature enhancements and changes made between these versions in addition to schema changes, command line argument differences, etc. Applications will require requalification and testing between the versions.
We constructed our cluster using publicly available cloudformation templates on AWS, is this environment upgradeable?	No.	No, the cloud formation templates are constructed in a way for use in test or small temporary environments. As such they are not for Enterprise deployments were upgrades will likely need to be supported.
In our cluster, we need to upgrade the following: Operating Environment, Docker, and DC/OS. Can we do all at once, together?	No.	Doing all simultaneously will likely result in many different issues with little possibility of sorting through the pieces to identify what broke and which component is in conflict with

		another. Best practice is to choose one component to upgrade-- upgrade throughout the cluster, let it soak for a period of time, then move to the next layered component.
Well, we did a leap-frog type of upgrade and the process worked upgrading from 1.9.0 to 1.11.1 but none of the apps are working and the universe/catalog is missing. Why?	Ugh. No.	Due to the many differences between the versions (e.g. schema changes, feature/functionality, authentication, etc.) this upgrade path will not work.
During our upgrade, we want to redirect or prevent users from accessing the apps during the upgrade, is this possible?	Yes.	Use ACLs to restrict user access to just their app namespaces. Use ACLs to restrict Marathon-LB service accounts.
We've made substantial modifications to DC/OS at our site and the changes are in /opt/mesosphere--- will these be retained during the upgrade?	No.	Customizations for DC/OS should be made in /var/dcos ; /var/mesos, *not /opt/mesosphere*
Our cluster had a lot of constraints configured for apps in 1.9.8, but now in 1.10.7 they are gone. Why? Shouldn't they have persisted?	No.	You must check the constraints of your cluster environment prior to upgrade. Use contraints.py within the documentation to check for validity of constraints before upgrading…
In order to save a bit of time we decided to run the upgrade installation script with --preflight checks disabled. Now things are broken?	No.	In production environments, *always allow and have complete* preflight checks. They are there to make sure the environment will support the version of code being upgraded / deployed.

Analysis - Pre-Upgrade Activities

Capturing Current Cluster and Application State

Before beginning the actual upgrade activity, time and care must be taken to capture the current state of operations- both from the application perspective, deployments, and overall cluster health. The more diligent this pre-upgrade-activity examination is, the more likely it is that any issues discovered and addressed will assist in assuring that the upgrade is successful.

Cluster State

Activity	*Result*
Capture number of tasks	Per node, ascertain number of containers
Number of nodes (public/private)	
Health of Marathon(s)	
Health of Service Marathon (MoM)	
Health of Exhibitor	

Commands:

Task	*Command*
Number of tasks	`For i in hosts; do docker ps |wc -l; done; curl leader.mesos:8080/v2/apps|jq .apps[].id|wc -l`

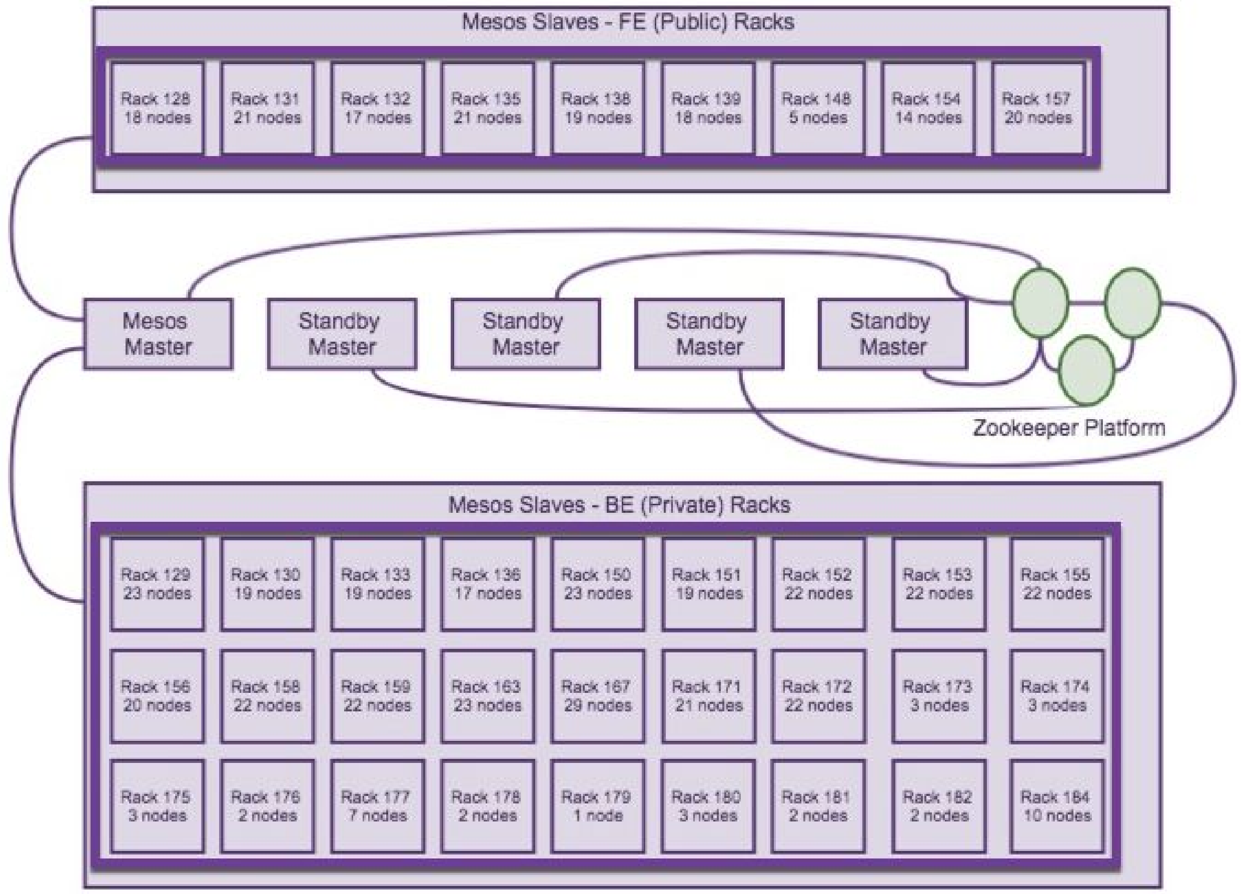
Number of nodes	`curl leader.mesos:5050/slaves|jq '.slaves[] | "\(.hostname) \(.active)"' | grep true|wc -l`
Nodes not connected to the Cluster	`curl leader.mesos:5050/slaves|jq '.slaves[] | "\(.hostname) \(.active)"' | grep false|wc -l`
Marathon health	`curl leader.mesos:8080/v2/leader; curl leader.mesos:8080/v2/info | jq .`
Apps deployed on Marathon	`curl leader.mesos:8080/v2/apps|jq .apps[].id|wc -l`
Exhibitor health	`curl leader.mesos:8181/exhibitor/v1/cluster/status | jq.`

Gathering Data

Activity	*Current Result*	*Target Version*
Current version of DC/OS		
Current version of Marathon		
Current version of Operating Environment		
Current version of Docker		
Existing support cases and JIRAs		
DC/OS log bundle		
Journalctl & systemctl output		
Application data		
Identify what is running, where...	<check constraints and pinned tasks>	
Validating constraints	<constraints.py>	

Cluster Architecture Diagram

--Current diagram represents a sample cluster--



Node-Rack Configuration Worksheet

Private Slave Nodes:

Rack Number	Number of nodes	Pinned / Unique Tasks
###	XX	TBD
###	XX	TBD
Total	### (tbd)	-

Public Slave Nodes:

Rack Number	Number of nodes	Pinned / Unique Tasks
All		TBD
###	XX	TBD
###	YY	TBD
Total	### (tbd)	-

Actionable Items before Beginning an Upgrade

Things to Fix Before an Upgrade

1. Zookeeper state directory

Starting Location	*Target Location*
`/var/lib/zookeeper`	`/var/lib/dcos/exhibitor/zookeeper`

2. Config.yaml

Starting Configuration	*Target Configuration*
TBD	TBD

3. Marathon

Current Marathon Version	*Target Version*
TBD	TBD
Current Service Marathon	Target Service Marathon
TBD	TBD

Marathon Monitoring Before Upgrade (Start this)

** In the pre-upgrade data collection phase, and during the upgrade log levels should be set to informational so there will be plenty of log data in the event RCA needs to be conducted during the entire process.

Collect details from end-points

1. Implement monitoring loop for gathering Marathon metrics

```
while sleep 1; do echo; date; curl
mon-marathon-service.containerip.dcos.thisdcos.directory:8080/metrics; done >
metrics.log
```

```
while sleep 1; do echo; date; curl mon-marathon-service.mesos:22223/metrics;
done > metrics2.log
```

```
while sleep 5; do
> echo "";
> date;
> curl leader.mesos:8080/metrics;
> echo "--------------------new line---------------------";
> done >metrics-leader-mesos.log
```

2. Implement task on DC/OS cluster to monitor Marathon state:

```
{
  "id": "/mon-marathon-service-metrics-curl-history",
  "cmd": "while sleep 5; do echo; date; curl
mon-marathon-service.mesos:22223/metrics | jq .; done",
  "cpus": 0.1,
  "mem": 32,
  "disk": 0,
  "instances": 1,
  "container": {
    "type": "MESOS",
    "volumes": []
  },
  "env": {
    "CONTAINER_LOGGER_MAX_STDOUT_SIZE": "100mb",
    "CONTAINER_LOGGER_LOGROTATE_STDOUT_OPTIONS": "rotate 9\ncompress"
  },
  "portDefinitions": []
}
```

4. Constraints

Use the following Python script to validate constraints before upgrading:
https://github.com/mesosphere/public-support-tools/blob/master/check-constraints.py

Timing of Events

a. Verify that all systems are health, logs are being monitored, and inventory is done
b. Verify that the code required for the upgrade is local and available to the cluster
c. Upgrade the masters
    i. Test/Soak
d. Upgrade the private agents
    i. Test/Soak
e. Upgrade the public agents
    i. Test/Soak
f. Verify the stability and acceptance of the upgrade

What Happens During an Upgrade

DCOS-Upgrade-Script

The DC/OS Upgrade script is generated as part of the preparation to upgrade or patch the cluster. It's important to understand what this script is going to do to the nodes in the cluster before beginning the upgrade. The script does check status of the upgrade and cluster both pre and post upgrade process. Here's a general idea of what it does at a high level:

Upgrade Script:

Step	Action
a)	Sources /opt/mesosphere/environment.export -- contains details about the current cluster installation and configuration, including version and binaries for DC/OS
b)	Checks version of the DC/OS cluster by verifying the /opt/mesosphere/etc/dcos-version.json file
c)	If $SKIP_CHECKS is set to true (e.g. don't do the checks) no checks are performed. If $SKIP_CHECKS is set to default, false, then the upgrade script checks the environment out, at the node level. It does this specifically with the command: /opt/mesosphere/bin/dcos-diagnostics check node-prestart -- if the checks fail, the upgrade stops and leaves the node/cluster undisturbed *Note: Don't skip the checks. Do the checks.
d)	Verifies the role of the node in the cluster by checking: /etc/mesosphere/roles -- {slave|slave_public|master}
e)	Depending on the role discovered in the previous step the script knows which binaries or flags to run for pkgpanda in the next step
f)	Using pkgpanda, the upgrade script fetches (downloads) the target version DC/OS binaries
g)	Using pkgpanda, the upgrade script activates (switches old, to new) the DC/OS binaries
h)	After pkgpanda had applied the updated packages, dcos-diagnostic checks are run again and with the value of node-poststart to verify how successful the upgrade was, or if it failed

Checking Status During an Upgrade

Sometimes, upgrades seem to take a long time. Maybe longer than expected. A couple of things to keep in mind-- these can all impact the duration of the actual run-time of the upgrade script:

- Size of the cluster environment and number of nodes being upgraded simultaneously
- Bandwidth between the node and the bootstrap node which is being communicated to for packages during the upgrade
- How busy the node being upgraded is
- Are there checks that it is running which are causing it to pause/hang?

During the upgrade it is prudent to monitor the following

- Health of cluster leader
- Health of Marathon
- Losing tasks
- Output of journalctl to verify that there are no issues occurring live during the upgrade

Recovering from Failures During an Upgrade

There are no “back-out” or “rollback” procedures for an upgrade. The simplest thing to do is figure out what caused the upgrade to fail and re-try. This takes a combination of looking at current state, logs, and potential bread crumbs left as output from the installation/upgrade script.

The most important point about failures during an upgrade is the following: **Do not proceed, especially if you are upgrading the masters**. However, if you are upgrading the agents, and have an issue with 1 out of 500, it might be reasonable to set the troubled node aside and proceed with the upgrade. If, the error happens again, then it must be investigated. In some cases, nodes may have unique reasons for failing -- e.g. missing binaries, operating environment issues, etc. In these cases the nodes may be set aside and rectified later-- either as a fresh new installation of the version or as a fix to the previously attempted upgrade.

Execute an Upgrade

Beginning an upgrade is something which must be taken with great consideration. Before running the first upgrade event, verifying the following, fix / adjust as necessary:

- All data has been gathered and analyzed
- The cluster is healthy and in an operational state
- You have verified both the existing / starting configuration and the target configuration ensuring that there will be no surprises in config changes which could lead to a catastrophic failure
- You've backed up ZK state

If the above conditions have been met, proceed with the upgrade. In the event that the upgrade script automated, either via Ansible, Puppet, or some other means, you must inspect the upgrade automation to make sure it follows Mesosphere's steps to perform the actual upgrade.

A Summary of Upgrade Steps

1. Timing of events - Upgrade events should be conducted in the following order
    a. Data collection and analysis
    b. State backup
    c. Configuration backup
        i. Qualification of items such as config.yaml and ip-detect
    d. Upgrade master nodes
        i. Test
    e. Upgrade Service Marathon as required
        i. Test to make sure Apps re-register
    f. Upgrade private nodes
        i. Test
    g. Upgrade public nodes
        i. Test
    h. Validate upgrade success
    i. Apply site-specific upgrade modifications as needed

Pre-Upgrade Checklist

- Document and account for any application dependencies (including pinned or constrained tasks)
- MoM (service Marathon) version - availability of the target MoM image in the datacenter container registry
- Availability of the toolbox (dcos-debug) image in the datacenter container registry
- Availability of the Mesos executor workaround tarball in the datacenter management server
- Check on NTP synchronization for all cluster nodes
- Availability of SSH connection into all cluster nodes
- Ansible reachability into all cluster nodes (ansible ping and automation security keys)
- Docker daemon health
- HAProxy health on agent nodes In this environment it's called HAProxy-Bridge
- Backup ZooKeeper State (can be done prior to the beginning of the upgrade)
    - Make consistent backup with guano - inconsistent if they cannot stop services
- Tar mesos.master:/var/lib to a safe location

    sudo tar -cf exhibitor-server1.tar /var/lib/dcos/exhibitor
    sudo tar -cvf old-zk-server1.tar /var/lib/zookeeper
    Leading master last

Backup ZK State and Other State

Capture current cluster state

Step	**Command**
2.1	`Curl leader.mesos state.json`
2.2	`Output:`
2.3	`Curl leader.mesos state.json --`

- Snapshot current cluster state (state.json)
    - Masters
    - Agents
    - Services
    - Tasks
    - Marathon (root + service)

Capture JSON for all the apps (v2/apps)

Step	**Command**
2.4	`curl marathon.mesos:8080/v2/apps |jq .`
2.5	`Output:`

Execute ZooKeeper backup

Use Guano to backup ZooKeeper (preferred method). However, Guano only allows ZK state to be backed up if files are closed. Since this is likely not possible in 99% of clusters, you can also simply tar the ZK directory and save the tarball to a safe place:

- `tar cvfo /var/tmp/zookeeper.tar /var/lib/zookeeper`
- (Or modern DC/OS /var/dcos/exhibitor/zookeeper)

Steps:

- Copy the toolbox script from https://github.com/mesosphere/docker-containers/blob/183ede6c16d18edac5bdbc73568dca5389a05da3/dcos-debug/toolbox
- Ensure that the `mesosphere/dcos-debug:latest` image is accessible through the Internet. Alternatively, download the Docker image and copy to a local image repository. Modify the toolbox script to reflect the local repository to pull the image from.
- Make toolbox executable using `chmod +x toolbox`
- Execute toolbox using `./toolbox` to enter the interactive shell in the container
- `guano -s localhost:2181 -d <znodes to dump> -o <output directory>`; The command assumes that guano is run from toolbox running on a master node
- It is recommended that the Marathon instance being backed up is suspended (quiesced) before the backup is executed in order to get a consistent backup of all znodes
- If Marathon cannot be suspended, a tarball of the ZooKeeper working directory will serve as an inconsistent backup
- You must *backup both the platform and MoM instances before the upgrade*.
- Consistency of the backup can be verified by counting the number of znodes in the backup and comparing to the output of znode information in Exhibitor

Step	Command
2.6	Copy toolbox script: • https://github.com/mesosphere/docker-containers/blob/183ede6c16d18edac5bdbc73568dca5389a05da3/dcos-debug/toolbox
2.7	`chmod +x toolbox`
2.8	Run the toolbox: ./toolbox (enter a container (toolbox))
2.9	Backup ZK State, on a master: `guano -s localhost:2181 -d <znodes to dump> -o <output directory>`
2.10	Backup zookeeper location with tar: Master node: `tar cvf /var/tmp/zookeeper.tar /var/lib/zookeeper`
2.11	Verify both zookeeper guano backup and tar backup (tar tf)

Upgrade Cluster Nodes

Master nodes first, followed by agent nodes.

Considerations:

- Masters should be done 1 at a time, starting with a non-leading master and concluding with the leader so that election only takes place once
- Query cluster state and make sure that masters re-join the cluster as they are upgraded. If they *do not rejoin* STOP
- Once masters are complete, let the environment settle for a period of time. Verify health of the cluster
- Identify and classify any agent nodes as critical and non-critical
- Run upgrades in a canary fashion beginning with the non-critical agent nodes
- Test deployments on the subset of nodes before proceeding with upgrades
- Example upgrade strategy (modify this strategy based on existing SLOs and user expectations)
- Initiate upgrades on a small subset of common nodes
- Increase upgrades linearly (or at a suitable progression) with considerations on web server or artifact registry hosting binaries for the install
- Ensure nodes are being assigned the right Mesos role (slave, slave_public)

While the Upgrade is in Progress. . . .

Verify health of Exhibitor/ZooKeeper:

Step	Command
1	● Check FDs (`` `ls /proc/`pidof java`/fd` ``) ○ # of Exhibitor's open FD's should be < number of cluster nodes
2	● Check that the cluster is full and there's a leader `curl leader.mesos:8181/exhibitor/v1/cluster/status | jq.`
-	`Cluster state:`

Continued Checking State

Step	Command

3	On a master, run shell code:
	
	`#!/bin/bash`
	``for master in `flm|grep TYPE=CONTROL|awk '{print $2}'` ``
	`do`
	`echo mesos-master-$master`
	`curl $master:8181/exhibitor/v1/cluster/status|jq '.[]| "\(.isLeader)`
	`\(.hostname)"'`
	`done`

4	`Cluster status and Leader State`
-	TBD

Mesos Master Health Check - Replog Status Equal to 1

Step	Command

5	On a master, run shell code:
	
	`#!/bin/bash`
	`for master in $(host master.mesos | cut -d " " -f 4); do ssh $master -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -SsL "curl $master:5050/metrics/snapshot | jq . | grep registrar/log/recovered";done`

-	Output:
-	TBD

Deploy a test app

Step	Command

6	From within DC/OS, deploy a test app to be used during the upgrade for test/validation:
	`{`
	`"id": "/arthur/tmp-app",`
	`"cmd": "while [ true ] ; do sleep 5 ; done",`
	`"cpus": 0.1,`
	`"mem": 10,`
	`"disk": 0,`
	`"instances": 1,`
	`"constraints": [`
	`[`
	`"hostname",`
	`"LIKE",`
	`"<IP-ADDR>"`
	`]`
	`],`
	`"portDefinitions": [`
	`{`
	`"port": 10068,`
	`"protocol": "tcp",`
	`"labels": {}`

	```
	}
	],
	"uris": [
	"file:///docker.tar.gz"
	]
	}
	```
-	Running App:

Version Specific Upgrade Steps

DC/OS 1.9.X

Release Notes:
https://docs.mesosphere.com/1.9/release-notes/1.9.8/

Patching:
https://docs.mesosphere.com/1.9/installing/ent/patching/

Upgrading:
https://docs.mesosphere.com/1.9/installing/ent/upgrading/

1.9.x FAQ:
https://docs.mesosphere.com/1.9/installing/ent/faq/

DC/OS 1.10.X

Release Notes:
https://docs.mesosphere.com/1.10/release-notes/1.10.6/

Patching:
https://docs.mesosphere.com/1.10/installing/ent/patching/

Upgrading:
https://docs.mesosphere.com/1.10/installing/ent/upgrading/

1.10.x FAQ:
https://docs.mesosphere.com/1.10/installing/ent/faq/

DC/OS 1.11.X

Release Notes:
https://docs.mesosphere.com/1.11/release-notes/1.11.1/

Patching:
https://docs.mesosphere.com/1.11/installing/ent/patching/

Upgrading:
https://docs.mesosphere.com/1.11/installing/ent/upgrading/

1.11.x FAQ:
https://docs.mesosphere.com/1.11/installing/ent/faq/

Validating an Upgrade

Post upgrade checks should validate overall health of the cluster and should cover any edge cases that were not in scope during the upgrade phase validations.

- Verify health of DC/OS from the UI/API
- Re-run inventory collection to compare state from pre-upgrade
- Deploy applications to verify service availability

Additional Simple Checks

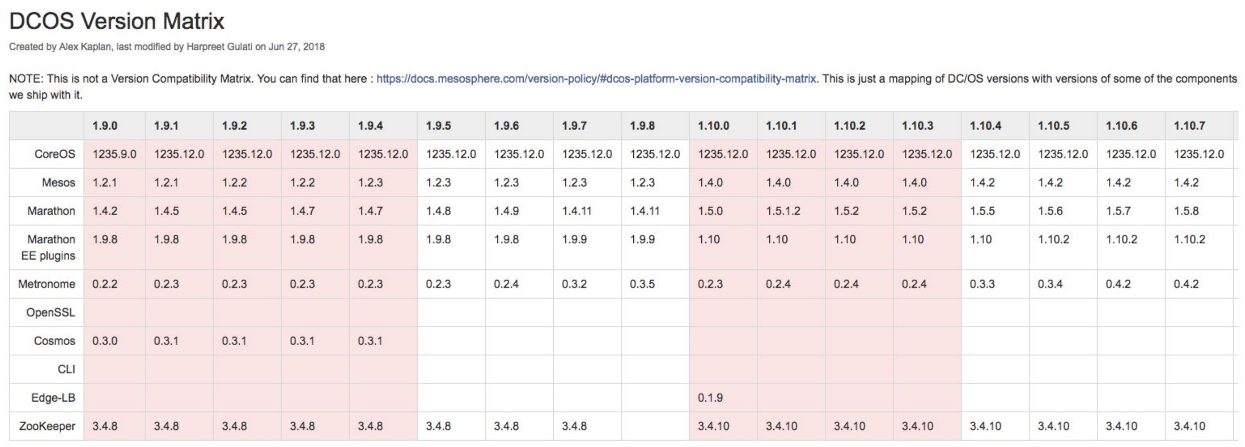
Check	*Action*	*Output*
Exit status of the upgrade command itself--	`<upgrade-script.bash>; echo $?`	`0 = success`
Verify version of the cluster from nodes	`cat /opt/mesosphere/etc/dcos-version.json`	`"version": "1.11.3"`
Verify version of Mesos - masters	`/opt/mesosphere/bin/mesos-master --version`	`mesos 1.5.1`
Verify version of Mesos - slaves	`/opt/mesosphere/bin/mesos-slave --version`	`mesos 1.5.1`

Appendix I: Resources - References

References

Item	URL
DC/OS Version Policy	`https://docs.mesosphere.com/version-policy/#dcos-platform-version-compatibility-matrix`

DC/OS Version Matrix



DCOS Version Matrix

Created by Alex Kaplan, last modified by Harpreet Gulati on Jun 27, 2018

NOTE: This is not a Version Compatibility Matrix. You can find that here : https://docs.mesosphere.com/version-policy/#dcos-platform-version-compatibility-matrix. This is just a mapping of DC/OS versions with versions of some of the components we ship with it.

	1.9.0	1.9.1	1.9.2	1.9.3	1.9.4	1.9.5	1.9.6	1.9.7	1.9.8	1.10.0	1.10.1	1.10.2	1.10.3	1.10.4	1.10.5	1.10.6	1.10.7
CoreOS	1235.9.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0	1235.12.0
Mesos	1.2.1	1.2.1	1.2.2	1.2.2	1.2.3	1.2.3	1.2.3	1.2.3	1.2.3	1.4.0	1.4.0	1.4.0	1.4.0	1.4.2	1.4.2	1.4.2	1.4.2
Marathon	1.4.2	1.4.5	1.4.5	1.4.7	1.4.7	1.4.8	1.4.9	1.4.11	1.4.11	1.5.0	1.5.1.2	1.5.2	1.5.2	1.5.5	1.5.6	1.5.7	1.5.8
Marathon EE plugins	1.9.8	1.9.8	1.9.8	1.9.8	1.9.8	1.9.8	1.9.8	1.9.9	1.9.9	1.10	1.10	1.10	1.10	1.10	1.10.2	1.10.2	1.10.2
Metronome	0.2.2	0.2.3	0.2.3	0.2.3	0.2.3	0.2.3	0.2.4	0.3.2	0.3.5	0.2.3	0.2.4	0.2.4	0.2.4	0.3.3	0.3.4	0.4.2	0.4.2
OpenSSL																	
Cosmos	0.3.0	0.3.1	0.3.1	0.3.1	0.3.1												
CLI																	
Edge-LB										0.1.9							
ZooKeeper	3.4.8	3.4.8	3.4.8	3.4.8	3.4.8	3.4.8	3.4.8	3.4.8		3.4.10	3.4.10	3.4.10	3.4.10	3.4.10	3.4.10	3.4.10	3.4.10

References

Information	*Link*
DC/OS Metrics	https://github.com/dcos/dcos-metrics
DC/OS Commons SDK	https://github.com/mesosphere/dcos-commons
DC/OS Security	https://docs.mesosphere.com/1.9/administration/id-and-access-mgt/ https://docs.mesosphere.com/1.9/administration/secrets/ https://docs.mesosphere.com/1.9/administration/tls-ssl/
DC/OS Upgrades	https://docs.mesosphere.com/1.9/administration/upgrading/

Appendix II: Marathon - Mesos Metrics

Continuing, Metrics Names and Required Endpoints:

https://<masterPublicIP>/marathon/metrics →

Check values in output:

Marathon Metrics Name	***Value***
service.mesosphere.marathon.uptime	5221988 (~87 min) (value/60000=Nmin)
service.mesosphere.marathon.leaderDuration	5220641
service.mesosphere.marathon.app.count	4
service.mesosphere.marathon.group.count	1
service.mesosphere.marathon.task.running.count	4
service.mesosphere.marathon.task.staged.count	0
service.mesosphere.marathon.core.task.update.impl.ThrottlingTaskStatusUpdateProcessor.queued	
service.mesosphere.marathon.core.task.update.impl.ThrottlingTaskStatusUpdateProcessor.processing	
service.mesosphere.marathon.core.task.update.impl.TaskStatusUpdateProcessorImpl.publishFuture	"count": **57**,"max": **0.033368508000000005**,"mean": **0.006840424377251375**,"min": **0.002051402**,"p50": **0.006769763000000001**,"p75": **0.006925651000000005**,"p95": **0.008879717**,"p98": **0.008879717**,"p99": **0.008879717**,"p999": **0.008879717**,"stddev":
service.mesosphere.marathon.core.launcher.impl.O	"count": 38,"m15_rate": 0.007533606193102653,"m1_rate": 0.016531615665440882,"m5_rate":

fferProcessorImpl.incomingOffers	
service.mesosphere.marathon.MarathonScheduler.resourceOffers	"count": 34,"max": 0.00484972,"mean": 0.00013228407690996707,"min": 0.00008738700000000001,"p50": 0.000114722,"p75": 0.000150264,"p95": 0.000150264,"p98": 0.000150264,"p99": 0.000150264,"p999":
service.mesosphere.marathon.MarathonScheduler.statusUpdate	"count": 57,"max": 0.00190416,"mean": 0.00008020949416247715,"min": 0.000058959,"p50": 0.000075134,"p75": 0.000077107,"p95": 0.000135357,"p98": 0.000135357,"p99": 0.000135357,"p999": 0.000135357,
Mesos Metrics Name	***Value***
files/debug	curl http://10.0.4.177:5050/files/debug |jq
/var/lib/dcos/mesos/log/mesos-master.log	Native on host (master)
/flags	curl http://10.0.4.177:5050/flags |jq .
/frameworks	curl http://10.0.4.177:5050/frameworks |jq
/health	<masterPrivIP>:5050/health
/machine/down	
machine/up	
/roles	curl http://10.0.4.177:5050/roles |jq
/slaves	curl http://10.0.4.177:5050/slaves |jq
/state	curl http://10.0.4.177:5050/state |jq
/state-summary	curl http://10.0.4.177:5050/state-summary
/tasks	curl http://10.0.4.177:5050/tasks.json |jq
/metrics/snapshot	curl http://10.0.4.177:5050/metrics/snapshot |jq
/system/stats.json	curl http://10.0.4.177:5050/system/stats.json |jq

Document Versioning

Document revisions, version, date, details:

Version	Date	Details
1	07/19/2018	Initial draft - ajohnson@mesosphere.com
2	07/24/2018	Review ready draft - ajohnson@mesosphere.com