**Apache Mesos Documentation**

**(Roll no 137130,137112,137132)**

## A Basic Overview of Apache Mesos

Apache Mesos is an open source cluster manager that simplifies running applications on a scalable cluster of servers, and is the heart of the Mesosphere system.

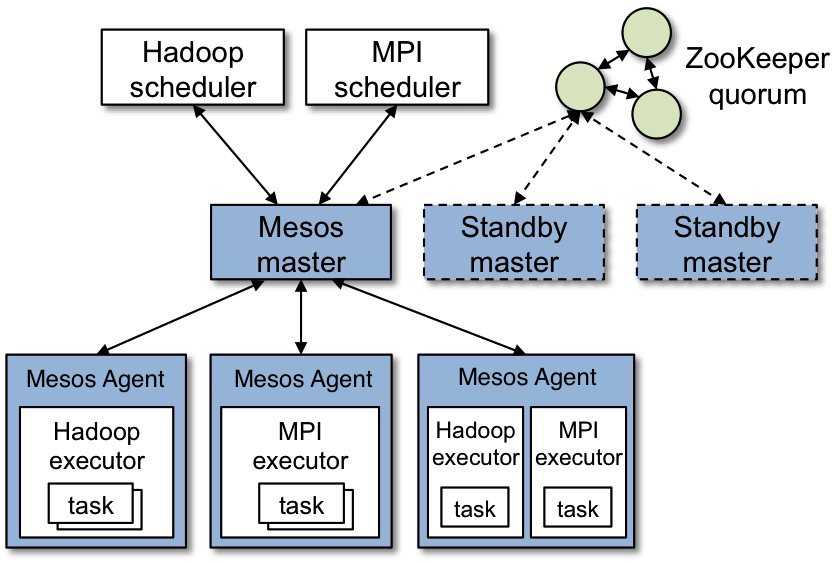
Mesos offers many of the features that you would expect from a cluster manager, such as:

* Scalability to over 10,000 nodes
* Resource isolation for tasks through Linux Containers
* Efficient CPU and memory-aware resource scheduling
* Highly-available master through Apache ZooKeeper
* Web UI for monitoring cluster state

### Mesos Architecture

Mesos has an architecture that is composed of master and slave daemons, and frameworks. Here is a quick breakdown of these components, and some relevant terms:

* **Master daemon**: runs on a master node and manages slave daemons
* **Slave daemon**: runs on a master node and runs tasks that belong to frameworks
* **Framework**: also known as a Mesos application, is composed of a scheduler, which registers with the master to receive resource offers, and one or more executors, which launches tasks on slaves. Examples of Mesos frameworks include Marathon, Chronos, and Hadoop
* **Offer**: a list of a slave node's available CPU and memory resources. All slave nodes send offers to the master, and the master provides offers to registered frameworks
* **Task**: a unit of work that is scheduled by a framework, and is executed on a slave node. A task can be anything from a bash command or script, to an SQL query, to a Hadoop job
* **Apache ZooKeeper**: software that is used to coordinate the master nodes



This architecture allows Mesos to share the cluster's resources amongst applications with a high level of granularity. The amount of resources offered to a particular framework is based on the policy set on the master, and the framework scheduler decides which of the offers to use. Once the framework scheduler decides which offers it wants to use, it tells Mesos which tasks should be executed, and Mesos launches the tasks on the appropriate slaves. After tasks are completed, and the consumed resources are freed, the resource offer cycle repeats so more tasks can be scheduled.

### High Availability

High availability of Mesos masters in a cluster is enabled through the use of Apache ZooKeeper to replicate the masters to form a quorum. ZooKeeper also coordinates master leader election and handles leader detection amongst Mesos components, including slaves and frameworks.

At least three master nodes are required for a highly-available configuration--a three master setup allows quorum to be maintained in the event that a single master fails--but five master nodes are recommended for a resilient production environment, allowing quorum to be maintained with two master nodes offline.

For more about Apache Mesos, visit [its official documentation page](http://mesos.apache.org/documentation/latest/).

## A Basic Overview of Marathon

Marathon is a framework for Mesos that is designed to launch long-running applications, and, in Mesosphere, serves as a replacement for a traditional init system. It has many features that simplify running applications in a clustered environment, such as high-availability, node constraints, application health checks, an API for scriptability and service discovery, and an easy to use web user interface. It adds its scaling and self-healing capabilities to the Mesosphere feature set.

Marathon can be used to start other Mesos frameworks, and it can also launch any process that can be started in the regular shell. As it is designed for long-running applications, it will ensure that applications it has launched will continue running, even if the slave node(s) they are running on fails.

For more about Marathon, visit [its GitHub page](https://github.com/mesosphere/marathon).

## A Basic Overview of Chronos

Chronos is a framework for Mesos that was originally developed by Airbnb as a replacement for cron. As such, it is a fully-featured, distributed, and fault-tolerant scheduler for Mesos, which eases the orchestration of jobs, which are collections of tasks. It includes an API that allows for scripting of scheduling jobs, and a web UI for ease of use.

In Mesosphere, Chronos compliments Marathon as it provides another way to run applications, according to a schedule or other conditions, such as the completion of another job. It is also capable of scheduling jobs on multiple Mesos slave nodes, and provides statistics about job failures and successes.

For more about Chronos, visit [its GitHub page](https://github.com/mesosphere/chronos).

## A Basic Overview of HAProxy

HAProxy is a popular open source load balancer and reverse proxying solution. It can be used in Mesosphere to route network traffic from known hosts, typically Mesos masters, to the actual services that are running on Mesos slave nodes. The service discovery capabilities of Mesos can be used to dynamically configure HAProxy to route incoming traffic to the proper backend slave nodes.

For more about the general capabilities of HAProxy, check out our [Introduction to HAProxy](https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts).

**Getting Started**

Downloading Mesos

There are different ways you can get Mesos:

1. Download the latest stable release from [Apache](http://mesos.apache.org/downloads/) (***Recommended***)

$ wget http://www.apache.org/dist/mesos/1.2.0/mesos-1.2.0.tar.gz

$ tar -zxf mesos-1.2.0.tar.gz

2. Clone the Mesos git [repository](https://git-wip-us.apache.org/repos/asf/mesos.git) (***Advanced Users Only***)

$ git clone https://git-wip-us.apache.org/repos/asf/mesos.git

*NOTE: If you have problems running the above commands, you may need to first run through the***System Requirements***section below to install the wget, tar, and git utilities for your system.*

System Requirements

Mesos runs on Linux (64 Bit) and Mac OS X (64 Bit). To build Mesos from source, GCC 4.8.1+ or Clang 3.5+ is required.

For full support of process isolation under Linux a recent kernel >=3.10 is required.

The Mesos agent also runs on Windows. To build Mesos from source, follow the instructions in the [Windows](http://mesos.apache.org/documentation/latest/windows/) section.

Make sure your hostname is resolvable via DNS or via /etc/hosts to allow full support of Docker’s host-networking capabilities, needed for some of the Mesos tests. When in doubt, please validate that /etc/hosts contains your hostname.

Ubuntu 14.04

Following are the instructions for stock Ubuntu 14.04. If you are using a different OS, please install the packages accordingly.

# Update the packages.

$ sudo apt-get update

# Install a few utility tools.

$ sudo apt-get install -y tar wget git

# Install the latest OpenJDK.

$ sudo apt-get install -y openjdk-7-jdk

# Install autotools (Only necessary if building from git repository).

$ sudo apt-get install -y autoconf libtool

# Install other Mesos dependencies.

$ sudo apt-get -y install build-essential python-dev python-virtualenv libcurl4-nss-dev libsasl2-dev libsasl2-modules maven libapr1-dev libsvn-dev

Ubuntu 16.04

Following are the instructions for stock Ubuntu 16.04. If you are using a different OS, please install the packages accordingly.

# Update the packages.

$ sudo apt-get update

# Install a few utility tools.

$ sudo apt-get install -y tar wget git

# Install the latest OpenJDK.

$ sudo apt-get install -y openjdk-8-jdk

# Install autotools (Only necessary if building from git repository).

$ sudo apt-get install -y autoconf libtool

# Install other Mesos dependencies.

$ sudo apt-get -y install build-essential python-dev python-virtualenv libcurl4-nss-dev libsasl2-dev libsasl2-modules maven libapr1-dev libsvn-dev zlib1g-dev

**DCOS**

**DC/OS SDK** is a collection of tools, libraries, and documentation for easy integration and automation of stateful services, such as databases, message brokers, and caching services.

DC/OS SDK is currently in alpha stage: it can run services, but APIs change regularly, and features are under active development.

**Benefits**

* **Simple and Flexible**: The SDK provides the simplicity of a declarative YAML API as well as the flexibility to use the full Java programming language.
* **Automate Maintenance**: Stateful services need to be maintained. With the SDK, you can automate maintenance routines, such as backup and restore, to simplify operations.
* **Production-Proven**: Building reliable services is hard. Uber and Bing platform teams use the SDK for mission-critical databases and message brokers.

**Quick Start**

From a workstation with 8G Memory, [Git](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git), [VirtualBox 5.0.x](https://www.virtualbox.org/wiki/Download_Old_Builds_5_0), and [Vagrant 1.8.4](https://releases.hashicorp.com/vagrant/1.8.4/), run:

1. Download the DC/OS SDK.

git clone https://github.com/mesosphere/dcos-commons.git

1. Create your local development environment.

cd dcos-commons/ && ./get-dcos-docker.sh

* Visit the DC/OS cluster [dashboard](http://172.17.0.2/) to verify your development environment is running.

1. Enter your development environment.

cd tools/vagrant/ && vagrant ssh

1. Build your hello-world example project.

cd /dcos-commons/frameworks/helloworld/ && ./build.sh local

1. Start your hello-world DC/OS service.

dcos package install hello-world

1. Explore your hello-world service.

* Visit the [dashboard](http://172.17.0.2/#/services/%2Fhello-world/) to see your hello-world service running.
* Click through to one of your tasks (e.g. world-server-1-<uuid>), select the **Files** tab, select **world-container-path**, and finally select the **output** file.

**Understanding the Hello World Service Specification**

The service specification declaratively defines the helloworld service:

name: "helloworld"

pods:

helloworld:

count: {{COUNT}}

tasks:

server:

goal: RUNNING

cmd: "echo 'Hello World!' >> helloworld-container-volume/output && sleep 10"

cpus: {{SERVER\_CPU}}

memory: 32

volume:

path: "helloworld-container-volume"

type: ROOT

size: 64

In above yaml file, we have:

* Defined a service with the name helloworld
* Configured the service to use ZooKeeper at master.mesos:2181 for storing framework state and configuration.
* Configured the API port using api-port: 8080. By default, each service comes with a default set of useful APIs to enable operationalization.
* Defined a pod specification for our helloworld pod using:

pods:

helloworld:

count: {{COUNT}}

tasks:

...

* Declared that we need atleast {{COUNT}} instances of the helloworld pod running at all times, where COUNT is the environment variable that is injected into the scheduler process at launch time via Marathon. It defaults to 1 for this example.
* Defined a task specification for our server task using:

tasks:

server:

goal: RUNNING

cmd: "echo 'Hello World!' >> helloworld-container-volume/output && sleep 10"

cpus: {{SERVER\_CPU}}

memory: 32

We have configured it to use {{SERVER\_CPU}} CPUs (which defaults to 0.5 for this example) and 32 MB of memory.

* And finally, configured a 64 MB persistent volume for our server task where the task data can be persisted using:

volume:

path: "helloworld-container-volume"

type: ROOT

size: 64

Deploying Hello Marathon: An Inline Shell Script

Let’s start with a simple example: an service that prints Hello Marathon to stdout and then sleeps for 5 sec, in an endless loop.  
Use the following JSON application definition to describe the application. Create a file with the name of your choice.

{

"id": "basic-0",

"cmd": "while [ true ] ; do echo 'Hello Marathon' ; sleep 5 ; done",

"cpus": 0.1,

"mem": 10.0,

"instances": 1

}

cmd in the above example is the command that gets executed. Its value is wrapped by the underlying Mesos executor via /bin/sh -c ${cmd}.

Then, add the service to DC/OS

dcos marathon app add <your-service-name>.json

When you define and launch a service, Marathon hands over execution to Mesos. Mesos creates a sandbox directory for each task. The sandbox directory is a directory on each agent node that acts as an execution environment and contains relevant log files. The stderr and stdout streams are also written to the sandbox directory.

Declaring Resources in Applications

To run any non-trivial application, you typically depend on a collection of resources: files or archives of files. To manage resource allocation, Marathon has the concept of URIs (uniform resource identifiers). URIs use the [Mesos fetcher](http://mesos.apache.org/documentation/latest/fetcher/) to do the legwork in terms of downloading (and potentially) extracting resources.

Before we dive into this topic, let’s have a look at an example:

{

"id": "basic-1",

"cmd": "`chmod u+x cool-script.sh && ./cool-script.sh`",

"cpus": 0.1,

"mem": 10.0,

"instances": 1,

"uris": [

"https://example.com/app/cool-script.sh"

]

}

The example above executes the cmd, downloads the resource https://example.com/app/cool-script.sh (via Mesos), and makes it available in the service instance’s Mesos sandbox. You can verify that it has been downloaded by visiting the DC/OS web interface and clicking on an instance of basic-1, then on the **Files** tab. You should find cool-script.sh there.

**Note:** As of Mesos v0.22 and above, the fetcher code does not make downloaded files executable by default. In the example above, cmd first makes the file executable.

As already mentioned above, Marathon also [knows how to handle](https://github.com/mesosphere/marathon/blob/master/src/main/scala/mesosphere/mesos/TaskBuilder.scala) application resources that reside in archives. Currently, Marathon will (via Mesos and before executing the cmd) first attempt to unpack/extract resources with the following file extensions:

* .tgz
* .tar.gz
* .tbz2
* .tar.bz2
* .txz
* .tar.xz
* .zip

The following example shows you how this looks in practice. Assume you have an application executable in a zip file at https://example.com/app.zip. This zip file contains the script cool-script.sh, which you want to execute. Here’s how:

{

"id": "basic-2",

"cmd": "app/cool-script.sh",

"cpus": 0.1,

"mem": 10.0,

"instances": 1,

"uris": [

"https://example.com/app.zip"

]

}

In contrast to the example basic-1 we now have a cmd with the value app/cool-script.sh. When the zip file gets downloaded and extracted, a directory app according to the file name app.zip is created and the content of the zip file is extracted into it.You can specify more than one resource. For example, you could provide a Git repository and some resources from a CDN as follows:

{

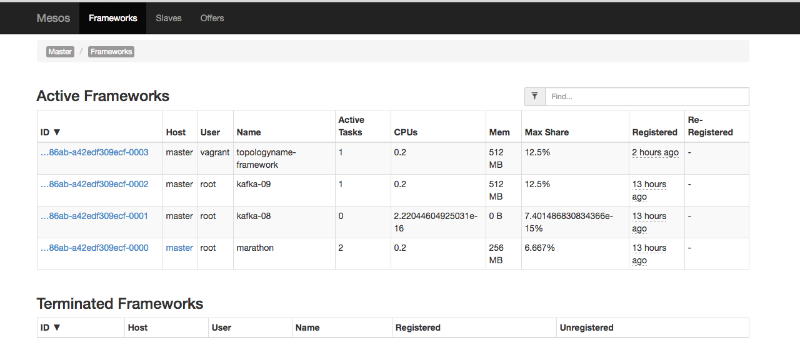
"uris": [

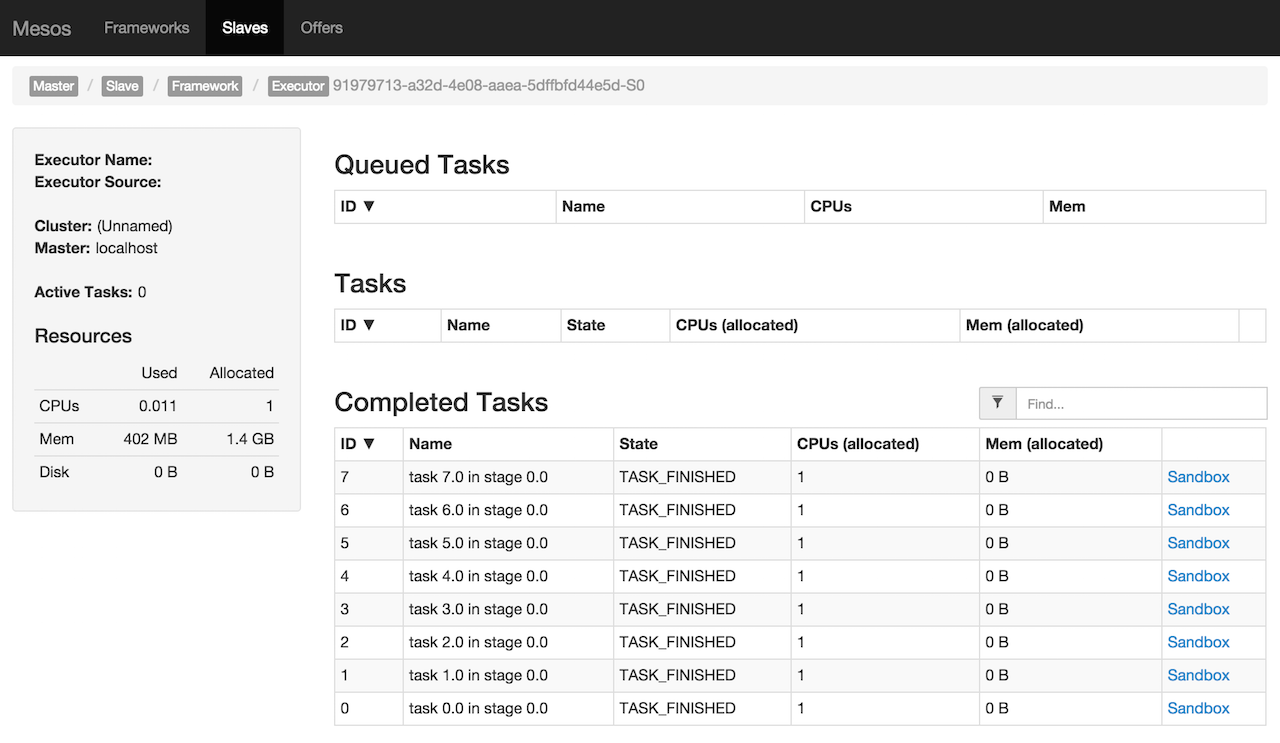
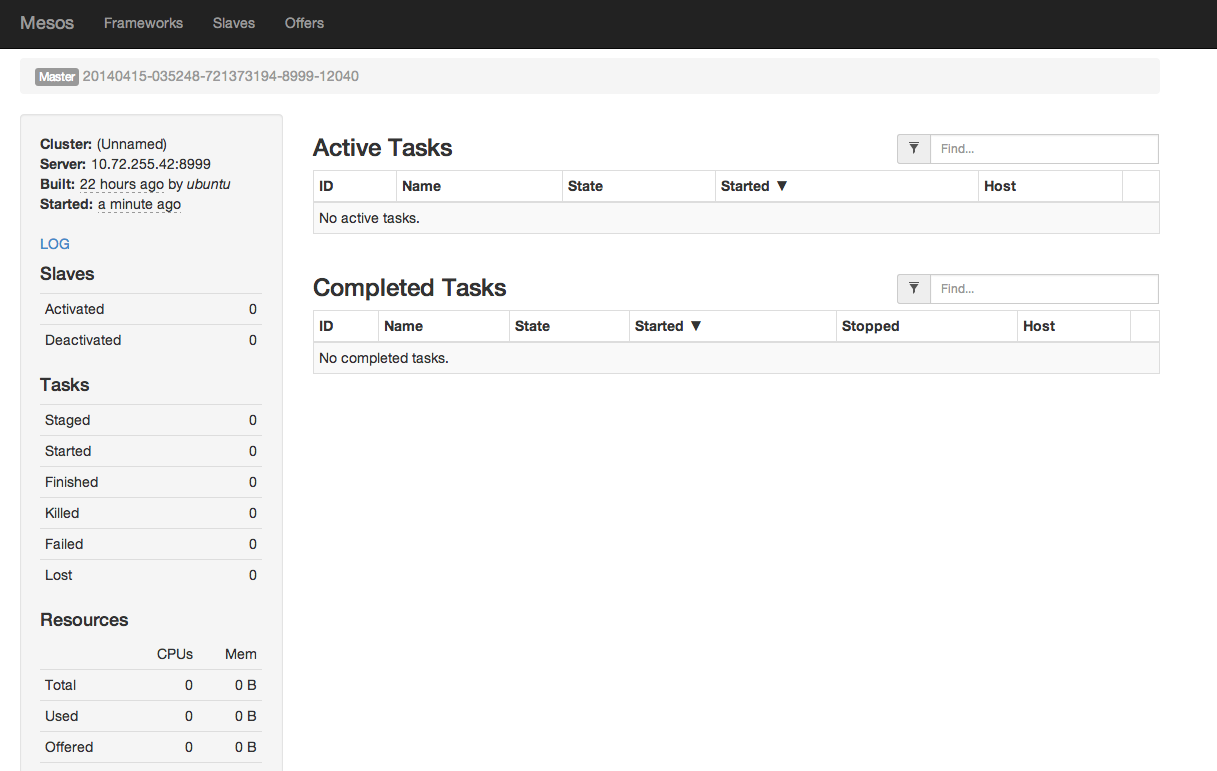
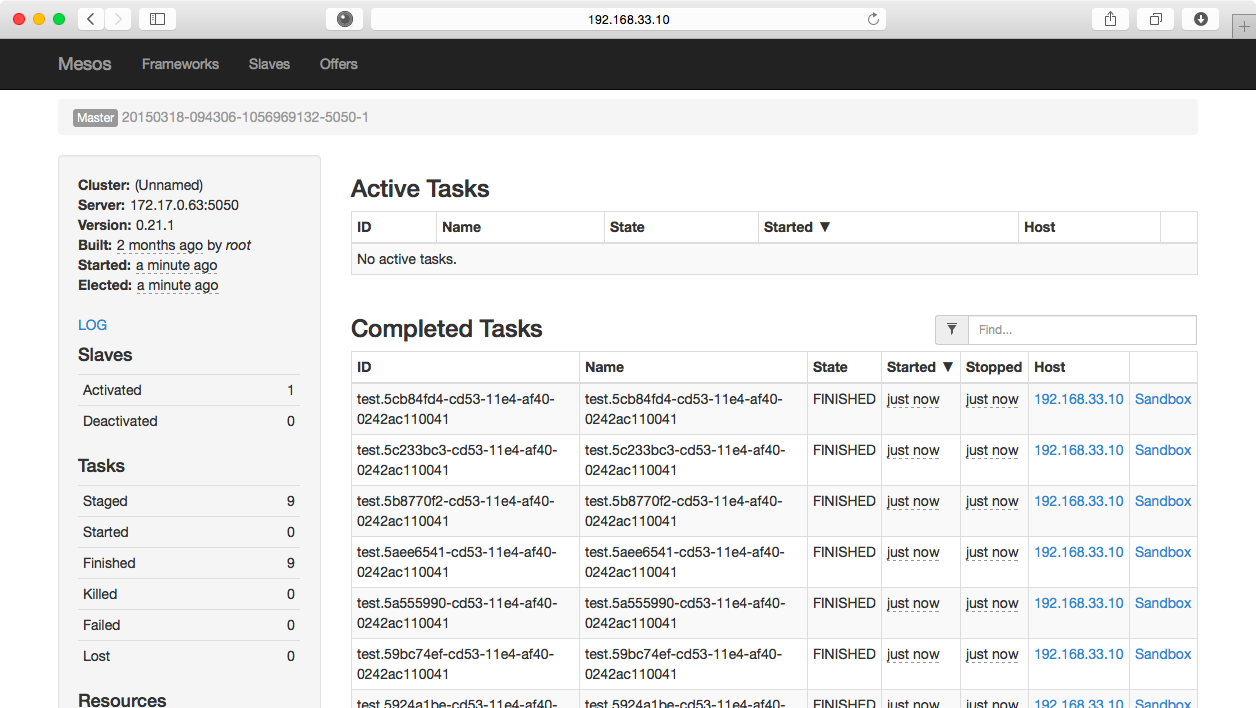
"https://git.example.com/repo-app.zip", "https://cdn.example.net/my-file.jpg", "https://cdn.example.net/my-other-file.css"

]

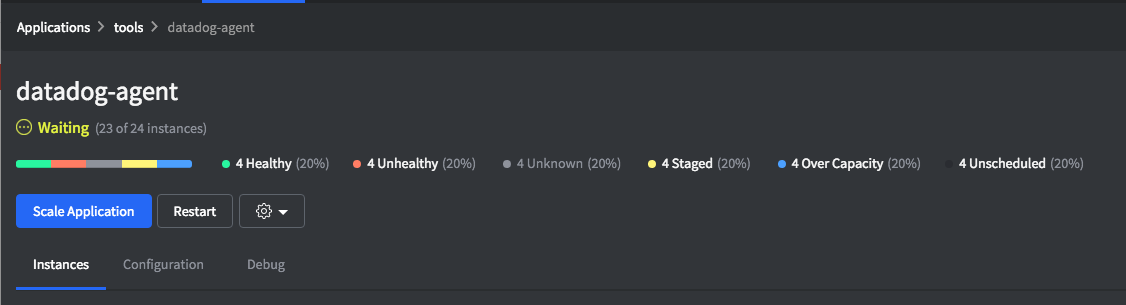
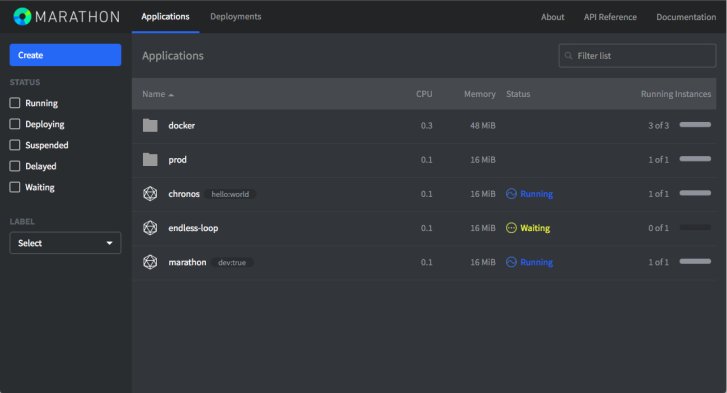
}

A typical pattern in the development and deployment cycle is to have your automated build system place the app binary in a location that’s downloadable via an URI. Marathon can download resources from a number of sources.

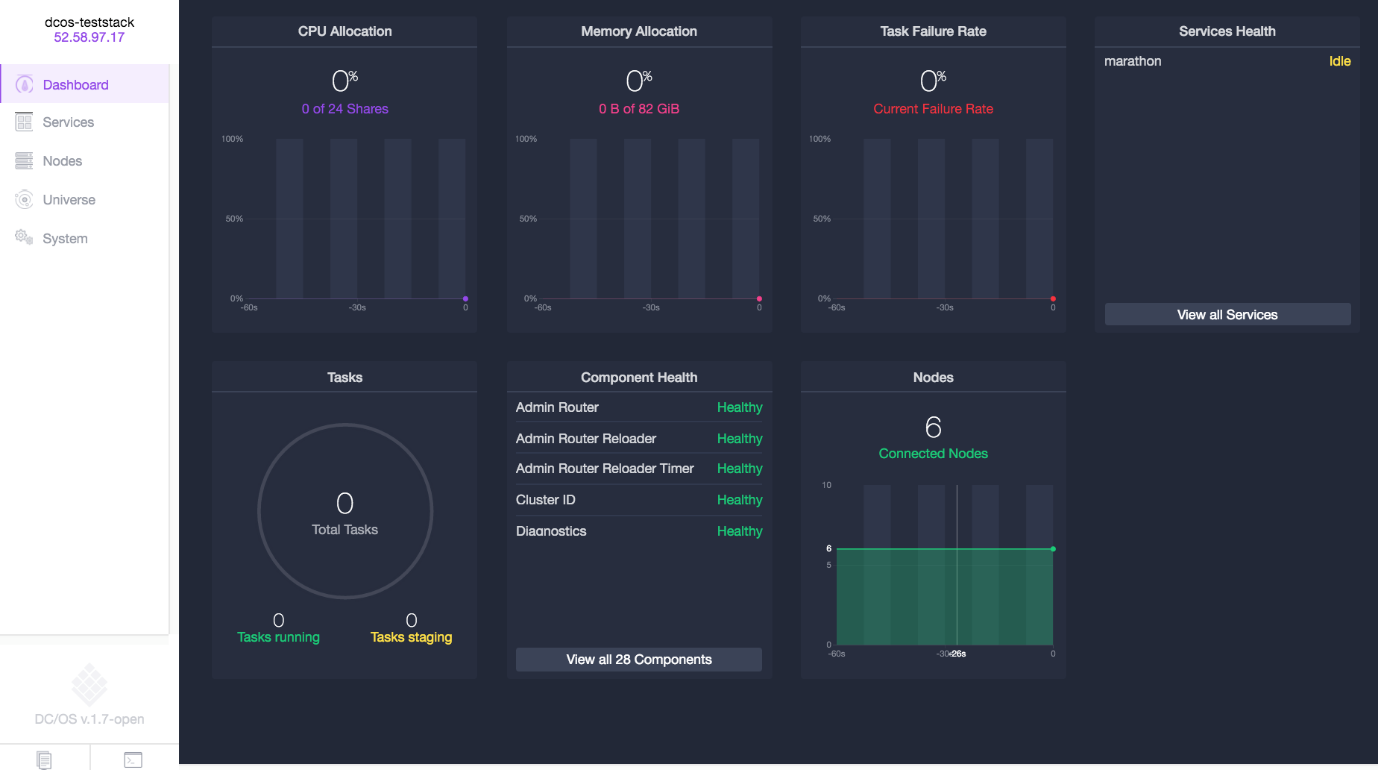
**Web Interfeces of MESOS **

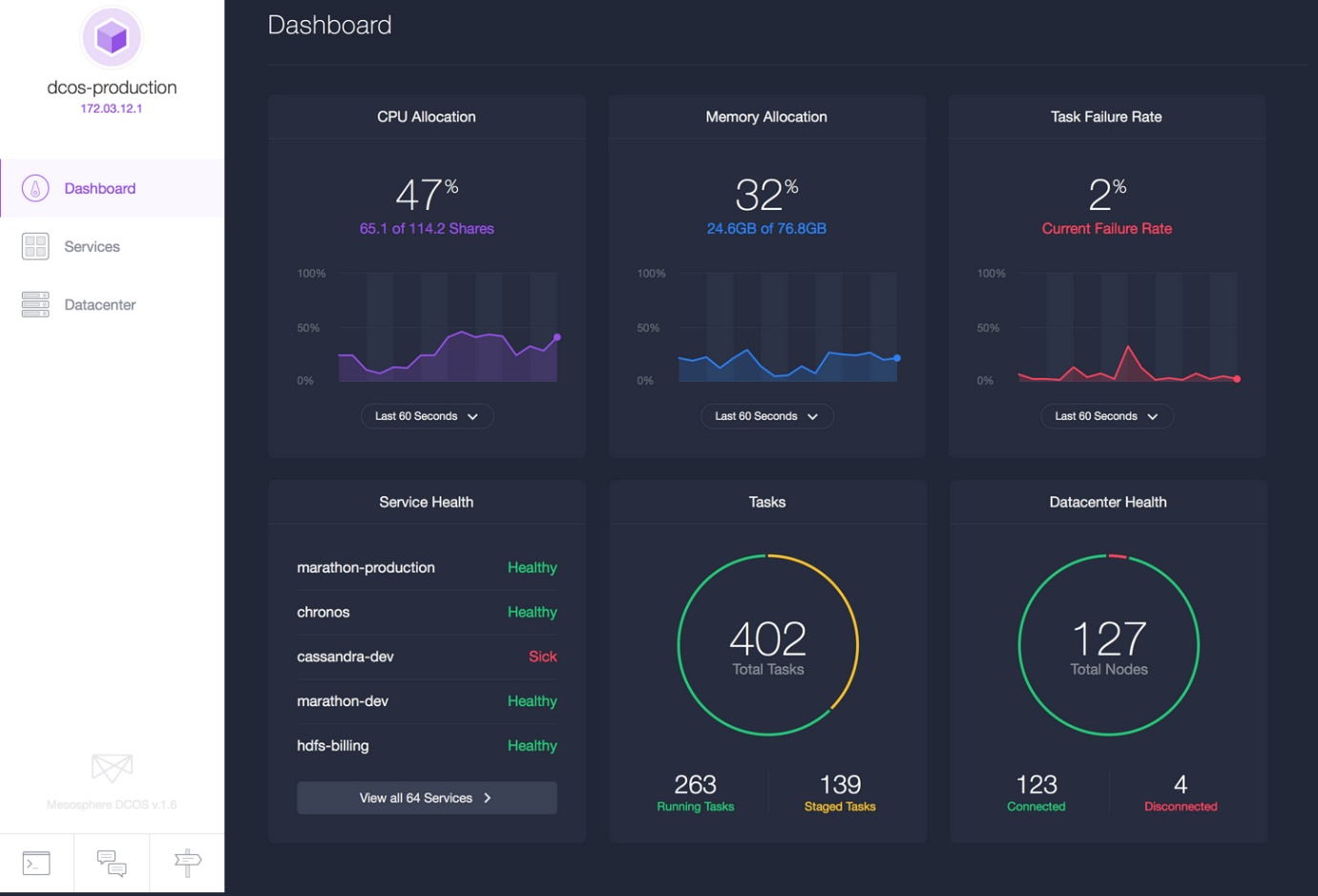
****

**Web Interface of MARATHON:**

****

**Web Interface of DCOS:**

****

****

****

**Web Interface of Application:**

