DC/OS Day Two Operations

Day Two

MESOSPHERE

Agenda:

Day Two

1. Monitoring and Metrics
2. Failure Handling
3. Production Checklist

Monitoring & Metrics

Gathering Health and Performance Metrics

Data-Driven Reporting

Monitoring and performance metrics help to answer the following questions:

- Are we close to cluster capacity?
- Are resources being used optimally?
- Is the system performing better or worse over time?
- Are there bottlenecks in the system?
- What is the response time of applications?

Metric Sources

Application Metrics

- QPS, latency, response time, hits, active users, errors

Container Metrics

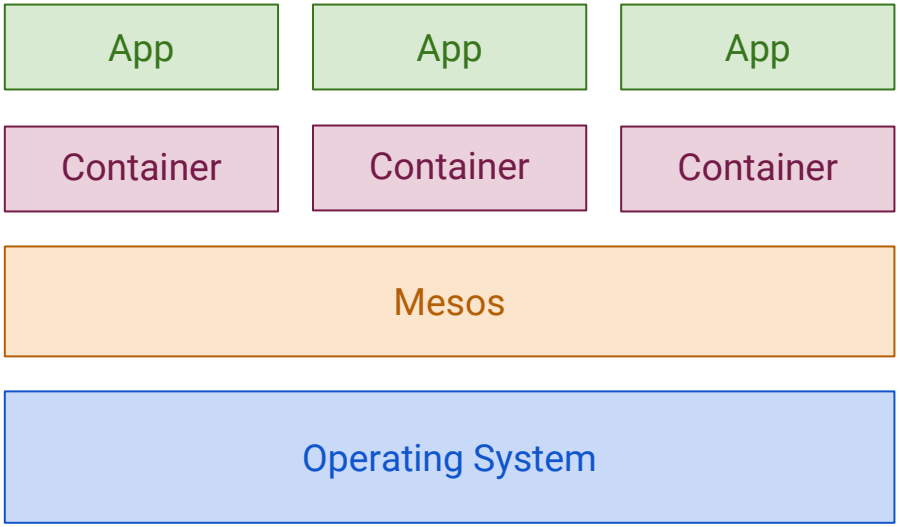
- CPU, memory, disk I/O, network I/O

Mesos Metrics

- Resources, frameworks, masters, agents, tasks, system, events

Host Metrics

- CPU, memory, disk I/O, network I/O



Data Sources

There are 4 components built-in to DC/OS that provide data that can be used for performance metrics and health monitoring:

1. DC/OS Metrics API (`dcos-metrics-master`, `dcos-metrics-agent`) exposes node, container, and application metrics
2. Marathon API (`dcos-marathon`) exposes metrics relating to Marathon's lifecycle, services, and connectivity with Mesos
3. Mesos APIs (`dcos-mesos-master`, `dcos-mesos-agent`) exposes performance and health metrics
4. DC/OS Diagnostics (`dcos-3dt`) aggregates and exposes health of DC/OS software components

Metrics API

- REST API that can retrieve metrics pertaining to nodes, containers, and applications running on the cluster
- Data can be sent to external services for visualization and alerting
- Runs on every node in the cluster
- Access to API is routed through `dcos-adminrouter` running on master node
- Master route:
  - `http(s)://<master_ip>/system/v1/metrics/v0/`
- Agent route:
  - `http(s)://<master_ip>/system/v1/agent/<agent_id>/metrics/v0/`
- Documentation

Metrics API: Benefits

Simplified configuration

- Automated collection of host and container level metrics
- Application level metrics integration via environment variables

Context injection

- Automated source tagging (agent IDs, container IDs, task IDs, etc.)
- Distributed aggregation

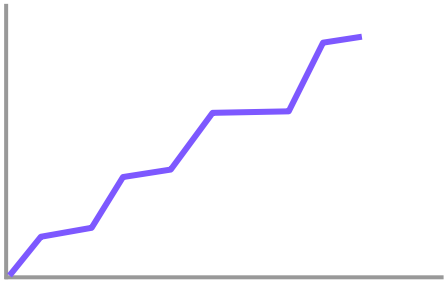
Collector per node

- Decoupled for faster upgrades and reconfigurations

Flexible output

- Kafka, logstash, Prometheus, etc.

Metric Types



Counters

Discrete events that are monotonically increasing

- # of failed tasks
- # of agent registrations

Gauges

An instantaneous sample of some magnitude

- % of memory allocated in cluster
- # of connected slaves

Metrics API: Master Nodes

- Metrics for a master node are available at the following endpoint:
  - `http(s)://<master_IP>/system/v1/metrics/v0/node`
- Response is returned as a JSON document
- Metrics returned include the following measurements:
  - CPU
  - System load
  - Memory/Swap
  - Process count
  - Uptime
  - Filesystem usage
  - Network I/O

Metrics API: Agent Nodes

- Metrics for an agent node are available at the following endpoint:
  - `http(s)://<master_IP>/system/v1/agent/<agent_id>/metrics/v0/node`
- Response is returned as a JSON document
- Metrics returned include the following measurements:
  - CPU
  - System load
  - Memory/Swap
  - Process count
  - Uptime
  - Filesystem usage
  - Network I/O
  - Metadata

Metrics API: Node Level Measurements

Metric	Description
`cpu.{cores,idle,total)`	Percentage of cores used/idle/total
`load.{1,5,15}min`	Load average for the past 1, 5, and 15 minutes
`memory.{free,total}`	Amount of free/total memory in bytes
`processes`	Number of processes that are running
`swap.{free,used,total}`	Amount of swap free/used/total
`uptime`	Host uptime
`filesystem.{name}.capacity.{free,total,used}`	Amount of free/total/used storage capacity for filesystem
`network.{name}.in.{bytes,dropped,errors,packets}`	Incoming network metrics for a network interface
`network.{name}.out.{bytes,dropped,errors,packets}`	Outgoing network metrics for a network interface

Metrics API: Containers

- Metrics for a container on an agent are available at the following endpoint:
  - `http(s)://<master_IP>/system/v1/agent/<agent_id>/metrics/v0/node/container/<container_id>`
- Response is returned as a JSON document
- Metrics returned include the following measurements:
  - CPU
  - Memory
  - Disk
  - Network
  - Metadata

Metrics API: Container Level Measurements

Metric	Description
`cpus.limit`	Number of allocated CPUs
`cpus_system_time_secs`	Total CPU time spent in kernel space
`cpus_throttled_time_secs`	Total time container has been throttled by the host kernel
`cpus_user_time_secs`	Total CPU time spent in userspace
`disk_limit_bytes`	Hard limit for disk
`disk_used_bytes`	Disk consumption
`mem_limit_bytes`	Hard RAM limit
`mem_total_bytes`	RAM consumption
`net_rx_{bytes,dropped,errors,packets}`	Incoming network metrics
`net_tx_{bytes,dropped,errors,packets}`	Outgoing network metrics

Metrics API: Application Measurements

- Application metrics can be published to the dcos-metrics component through environment variables
  - `STATSD_UDP_HOST`: Initialized to the IP address of the agent where the container is running
  - `STATSD_UDP_PORT`: Initialized to the bind port for the statsd server on the agent
- Metrics are appended with metadata such as:
  - `agent_id`
  - `container_id`
  - `framework_id`
  - `hostname`
  - `labels`
- Only works with UCR containers

Metrics API: `dcos` CLI

- Node and container level metrics can be retrieved through the `dcos` CLI
- Node level metrics:
  - Retrieve the ID of an agent node: `dcos node`
  - Retrieve a summary of metrics for a node: `dcos node metrics summary <ID>`
  - Retrieve a detailed report of each metric for a noce: `dcos node metrics details <ID>`
- Container level metrics:
  - Retrieve the ID of a task: `dcos task`
  - Retrieve a summary of metrics for a task: `dcos task metrics summary <ID>`
  - Retrieve a detailed report of each metric for a task: `dcos node task details <ID>`

Lab 6

Metrics API

Marathon Metrics

- Metrics for Marathon are available at the following endpoints:
  - `http://<master_IP>/marathon/metrics`
  - `http://<master_IP>/marathon/v2/apps`
- Metrics can be automatically exported to Graphite or Datadog via Marathon start up options:
  - `--reporter_graphite <graphite_server>`
  - `--reporter_datadog <datadog_server>`

Marathon Service Performance (Step 1 of 2)

```
curl http://<master_IP>/marathon/v2/apps/hello-world
```


```
{
  "appId": "/hello-world",
  "healthCheckResults": [],
  "host": "10.0.0.99",
  "id": "hello-world.57911772-47d6-11e8-bc94-821c2bfac18f",
  "ipAddresses": [
    {
      "ipAddress": "10.0.0.99",
      "protocol": "IPv4"
    }
  ],
  "ports": [
    3969
  ],
  "servicePorts": [],
  "slaveId": "729d2884-b78f-4aee-8ad4-3cec73d05a4b-S2",
  "state": "TASK_RUNNING",
  "stagedAt": "2018-04-24T15:44:15.251Z",
  "startedAt": "2018-04-24T15:44:16.458Z",
  "version": "2018-04-24T15:44:15.192Z",
  "localVolumes": []
}
```

Marathon Service Performance (Step 2 of 2)

```
curl http://<agent_IP>:5051/monitor/statistics
```


```
{
      "executor_id": "hello-world.57907b30",
      "executor_name": "Command Executor (Task: hello-world.57907b30) (Command: sh -c 'while [ true...')",
      "framework_id": "729d2884-b78f-4aee-8ad4-3cec73d05a4b-0001",
      "source": "hello-world.57907b30-47d6-11e8-bc94-821c2bfac18f",
      "cpus_limit": 0.2,
      "cpus_system_time_secs": 41.39,
      "cpus_throttled_time_secs": 0.386315377,
      "cpus_user_time_secs": 23.84,
      "mem_limit_bytes": 44040192,
      "mem_rss_bytes": 8445952,
      "mem_swap_bytes": 0,
      "mem_total_bytes": 8445952,
      "mem_unevictable_bytes": 0,
      "timestamp": 1524687903.8652
      ...
}
```

Alert Metrics for Mesos

Metric	Inference
`master/uptime_secs` is low	The master has restarted
`master/uptime_secs` < 60 for sustained periods of time	The cluster has a flapping master node
`master/tasks_lost` is increasing rapidly	Tasks in the cluster are disappearing. Possible causes include hardware failures, bugs in one of the frameworks, or bugs in Mesos
`master/slaves_active` is low	Slaves are having trouble connecting to the master
`master/cpus_percent` > 0.9 for sustained periods of time	Cluster CPU allocation is close to capacity
`master/mem_percent` > 0.9 for sustained periods of time	Cluster RAM allocation is close to capacity
`master/disk_percent` > 0.9 for sustained periods of time	Cluster disk allocation is close to capacity
`master/elected` is 0 for sustained period of time	No master is currently elected

Lab 7

Lab until :30

Marathon & Mesos Metrics

DC/OS Components Health

- DC/OS is packaged with a component called DC/OS Distributed Diagnostics Tool (3DT) which collects and aggregates the health of the DC/OS software components running on your cluster nodes
- 3DT runs on every node in the cluster, monitoring the state of `systemd` units
- Workflow:
  - 3DT running on leading master retrieves a list of all masters and agents
  - 3DT running on leading master queries 3DT on all other masters and agents and collects their respective states
  - 3DT aggregates the data and is exposed by dcos-adminrouter at the `/system/health/v1/` route
- Can also be queried via the DC/OS CLI to generate and download a diagnostics bundle that can be sent to the Mesosphere support team

DC/OS Components Health: GUI

Components

52 Components

Filter | All 52 | Healthy 52 | Unhealthy 0 | Download Snapshot

Name	Health
Admin Router Agent	Healthy
Admin Router Master	Healthy
CockroachDB	Healthy
CockroachDB Disable Diagnostics	Healthy
CockroachDB Set Configuration Timer	Healthy
DC/OS Backup Master	Healthy
DC/OS Backup Master Socket	Healthy
DC/OS Certificate Authority	Healthy
DC/OS Checks Timer	Healthy
DC/OS Cluster Linker Service	Healthy
DC/OS Cluster Linker Socket	Healthy

DC/OS Components Health: GUI

Components > Admin Router Agent (Healthy)

3 Health Checks

Filter

All Health Checks

Health	Node	Role
Healthy	10.0.0.11	Agent_public
Healthy	10.0.0.242	Agent
Healthy	10.0.0.99	Agent

Lab 8

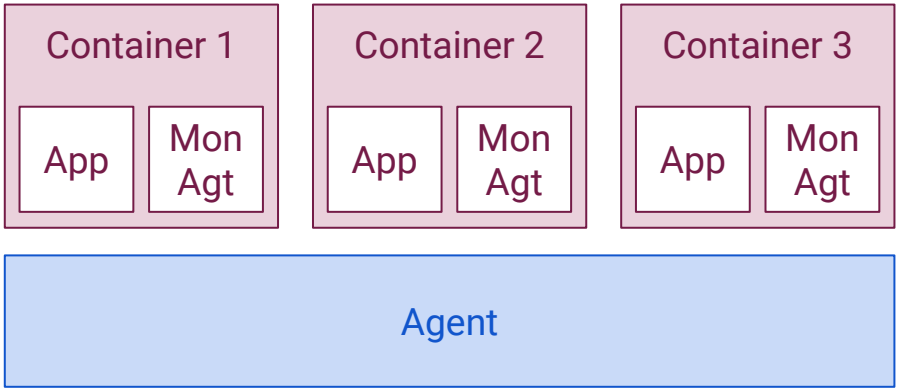
DC/OS Health Monitoring

```
Lab until :00
```

Container Level Metrics

Monitoring agent per container?

- Not scalable
- Increased footprint
- Higher likelihood of user error



High Level Workflow

DC/OS clusters subscribe to an event-driven monitoring model

Metric Collection	Metric Routing	Metric Storage	Metric Visualized
Host, container, and application level metrics are collected from the systems	Metrics are fed to a central routing system which contains rules indicating where metrics are destined to go	Metrics are persisted in a storage system	Metrics are plotted out in a graphing tool to make consumption of information more human-friendly

Step 1: Metric Collection

- A metric collector is a process that runs on a node as either a OS level daemon or as a container
- Responsible for collecting metrics related to:
  - The host itself
  - All the containers running on the host
  - The applications running within the containers
- Upon capturing metrics, the agent sends the data points to an event router
- Examples:
  - `collectd`
  - `cAdvisor`
  - `statsd`

Container Advisor (cAdvisor)

- Open source and created by Google
- Process that runs on a host that collects, aggregates, processes, and exports information about running containers
- Collects metrics such as isolation parameters, resource usage
- Supports both Docker and UCR containers
- Data is available through a REST API
- Data can automatically be exported to an external storage system such as InfluxDB, Elasticsearch, Kafka
- Can be installed through the DC/OS Package Catalog
- Documentation

Step 2: Event Router

- Receives metrics from the metric collector
- Central dispatcher which makes metrics consumable by other systems
- Examples:
  - fluentd
  - Kafka
  - Logstash
  - cAdvisor

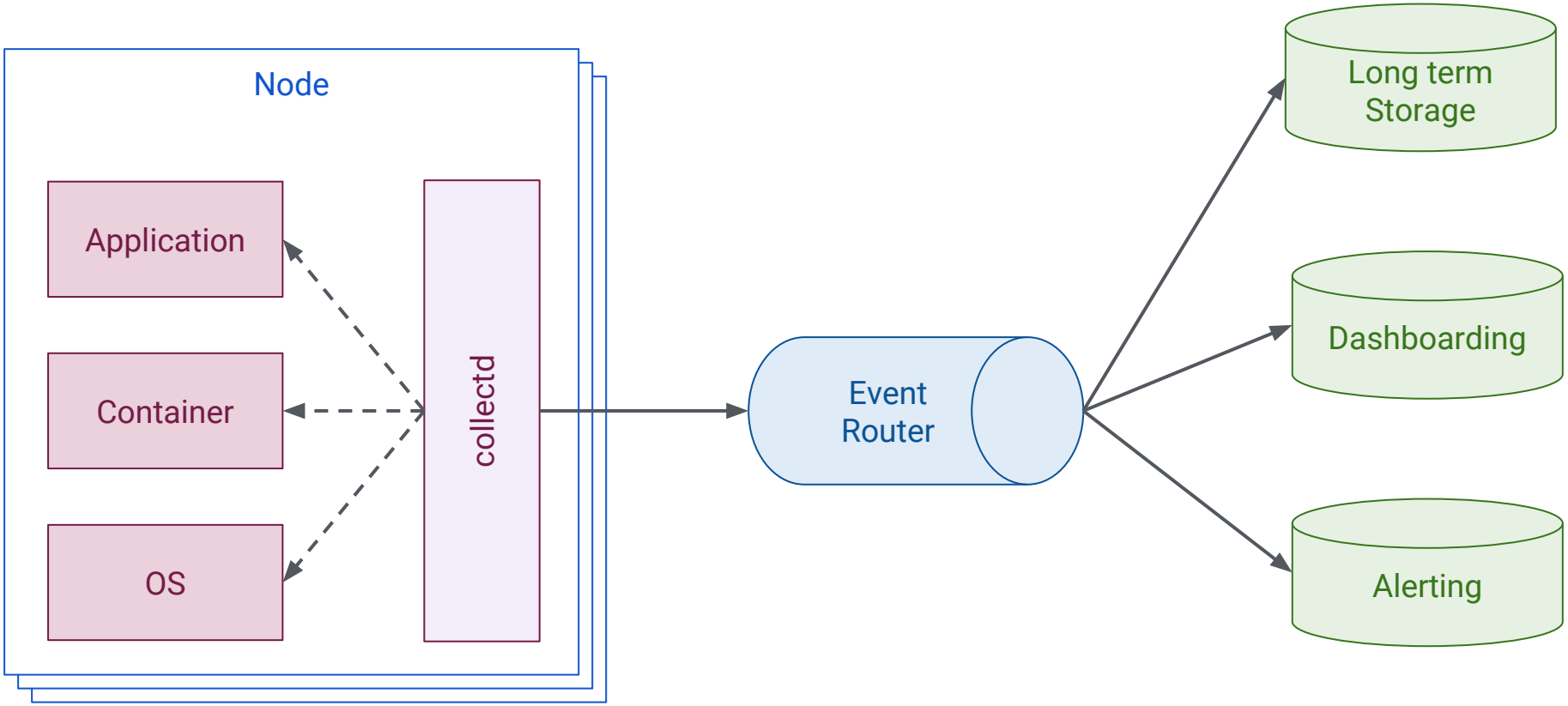
Step 3: Metric Storage

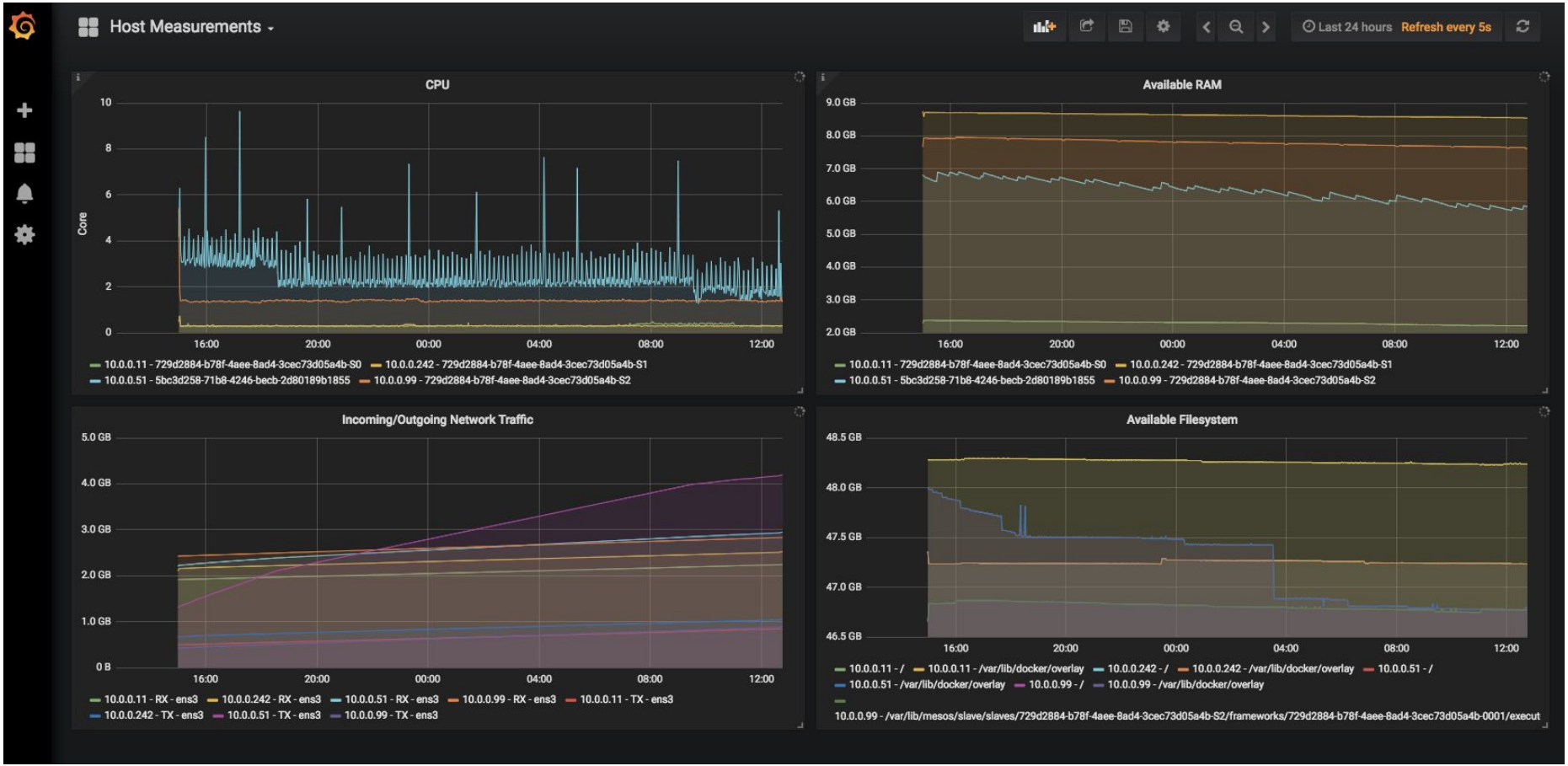
- Provides storage system to persist collected metrics
- Can have three forms from short-term/low-latency systems to long-term/persistent systems:
  - Alerting: If a metric has a certain unhealthy trend
  - Dashboarding: Plot the last hour in high resolution and a summary of the past 24hrs
  - Long Term: For legal and auditing obligations
- Examples:
  - Elasticsearch
  - Graphite
  - InfluxDB
  - Prometheus
  - Cassandra
  - Filesystem (local, CephFS, HDFS, etc.)

Step 4: Metric Visualization

- Provides customizable dashboarding
- Simplifies digestion of collected metrics
- Examples:
  - Grafana
  - D3
  - SignalFX

Putting it All Together




Host Measurements
Last 24 hours
Refresh every 5s
CPU
Core
10
8
6
4
2
0
16:00
20:00
00:00
04:00
08:00
12:00
10.0.0.11 - 729d2884-b78f-4aee-8ad4-3cec73d05a4b-S0
10.0.0.242 - 729d2884-b78f-4aee-8ad4-3cec73d05a4b-S1
10.0.0.51 - 5bc3d258-71b8-4246-becb-2d80189b1855
10.0.0.99 - 729d2884-b78f-4aee-8ad4-3cec73d05a4b-S2
Available RAM
9.0 GB
8.0 GB
7.0 GB
6.0 GB
5.0 GB
4.0 GB
3.0 GB
2.0 GB
Incoming/Outgoing Network Traffic
5.0 GB
4.0 GB
3.0 GB
2.0 GB
1.0 GB
0 B
10.0.0.11 - RX - ens3
10.0.0.242 - RX - ens3
10.0.0.51 - RX - ens3
10.0.0.99 - RX - ens3
10.0.0.11 - TX - ens3
10.0.0.242 - TX - ens3
10.0.0.51 - TX - ens3
10.0.0.99 - TX - ens3
Available Filesystem
48.5 GB
48.0 GB
47.5 GB
47.0 GB
46.5 GB
10.0.0.11 - /
10.0.0.11 - /var/lib/docker/overlay
10.0.0.242 - /
10.0.0.242 - /var/lib/docker/overlay
10.0.0.51 - /
10.0.0.51 - /var/lib/docker/overlay
10.0.0.99 - /
10.0.0.99 - /var/lib/docker/overlay
10.0.0.99 - /var/lib/mesos/slave/slaves/729d2884-b78f-4aee-8ad4-3cec73d05a4b-S2/frameworks/729d2884-b78f-4aee-8ad4-3cec73d05a4b-0001/execut

Lab 9

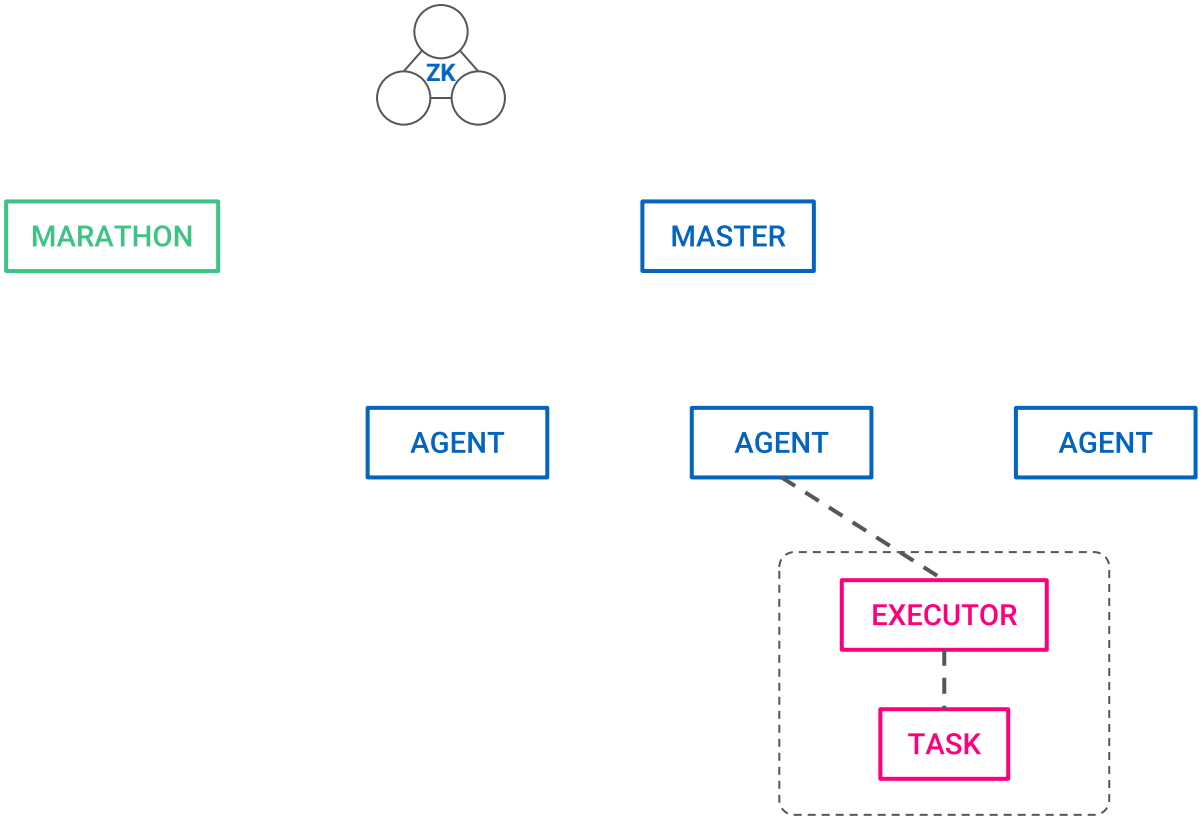
Lab until :35

Monitoring DC/OS with cAdvisor, InfluxDB, and Grafana

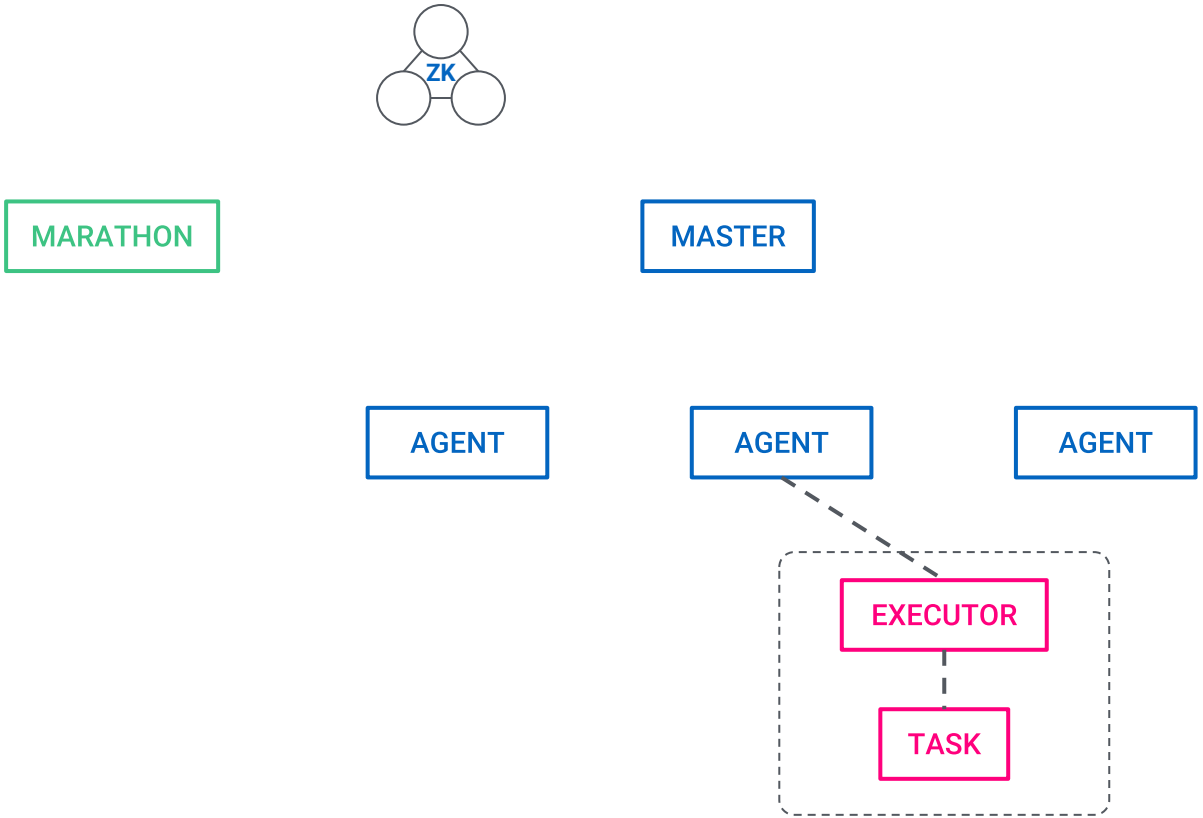
Cluster Failures

Task (Container) Failure

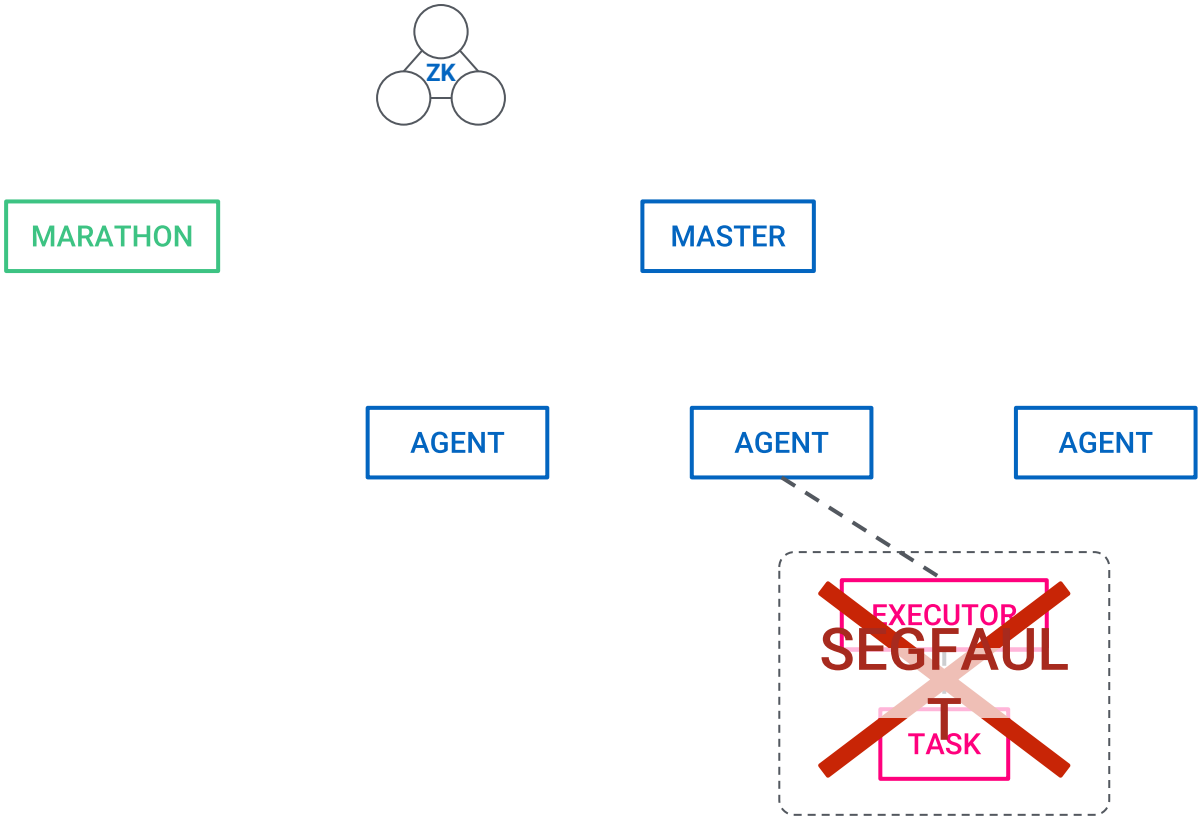
Task Failure



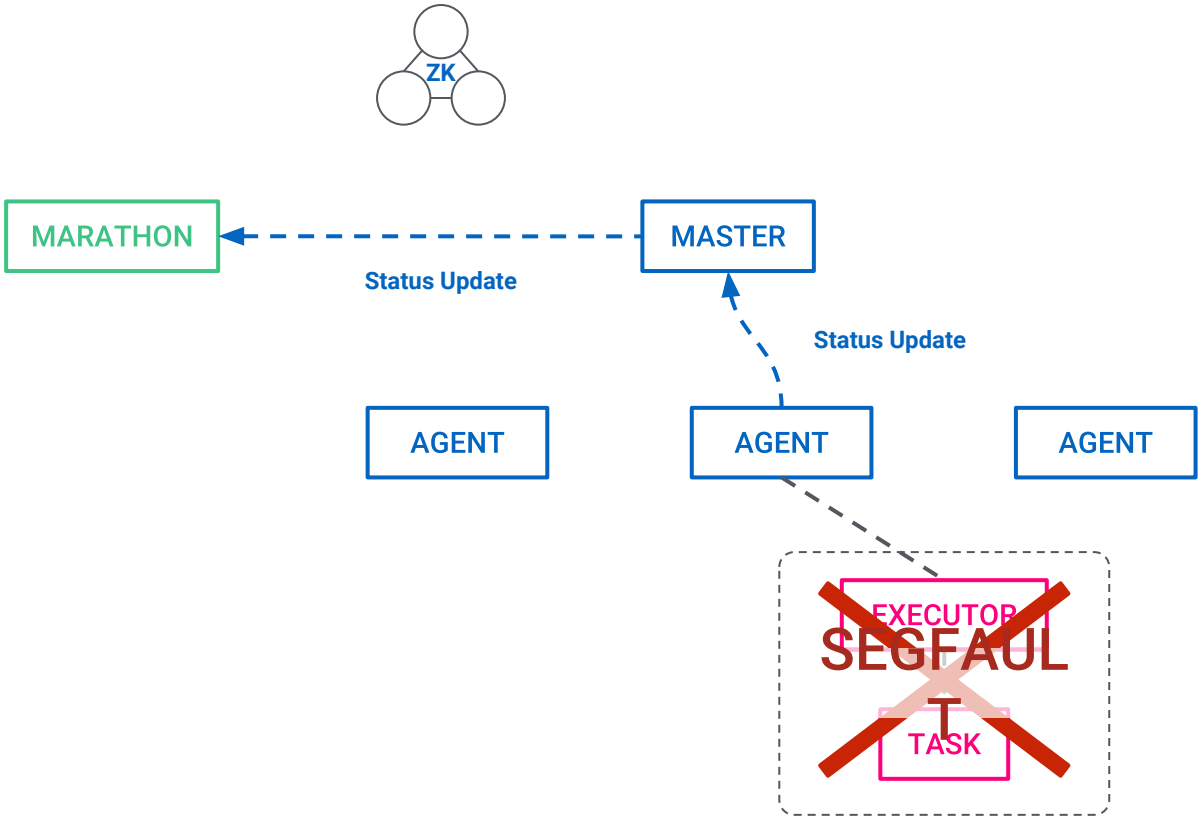
Task Failure



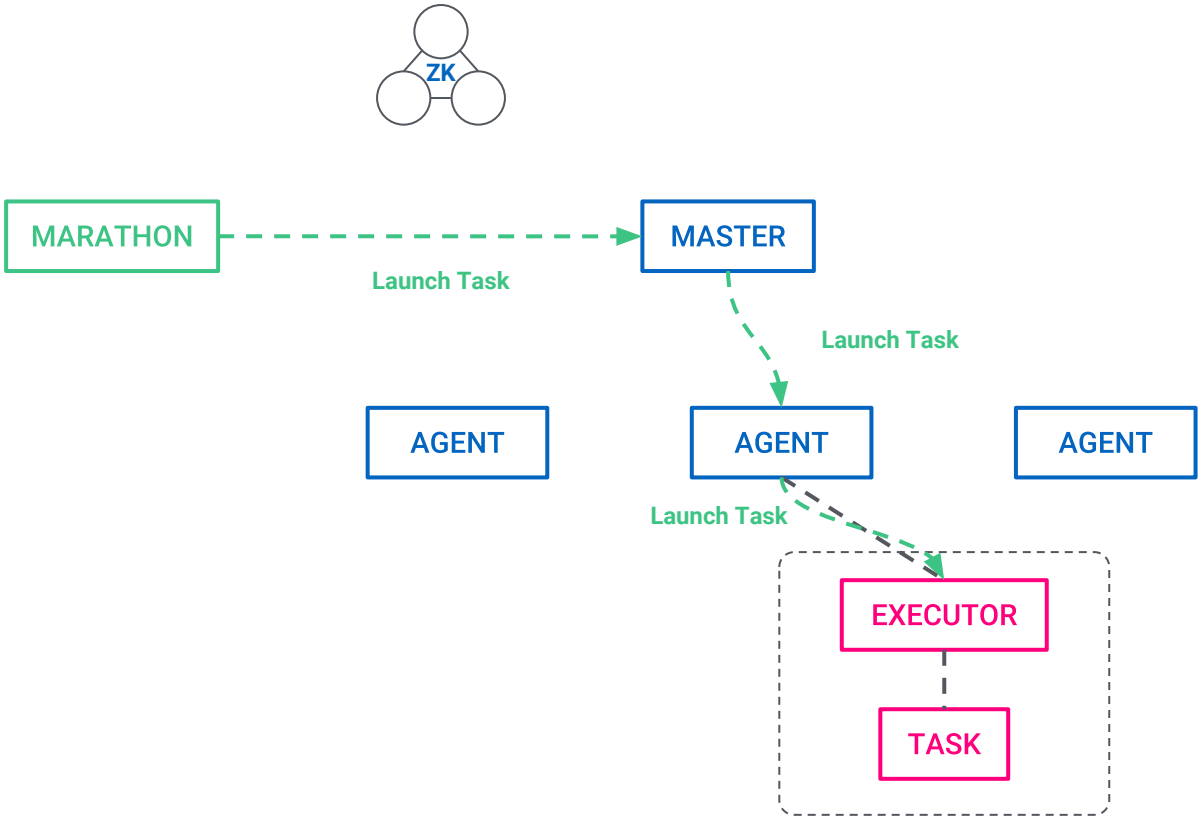
Task Failure



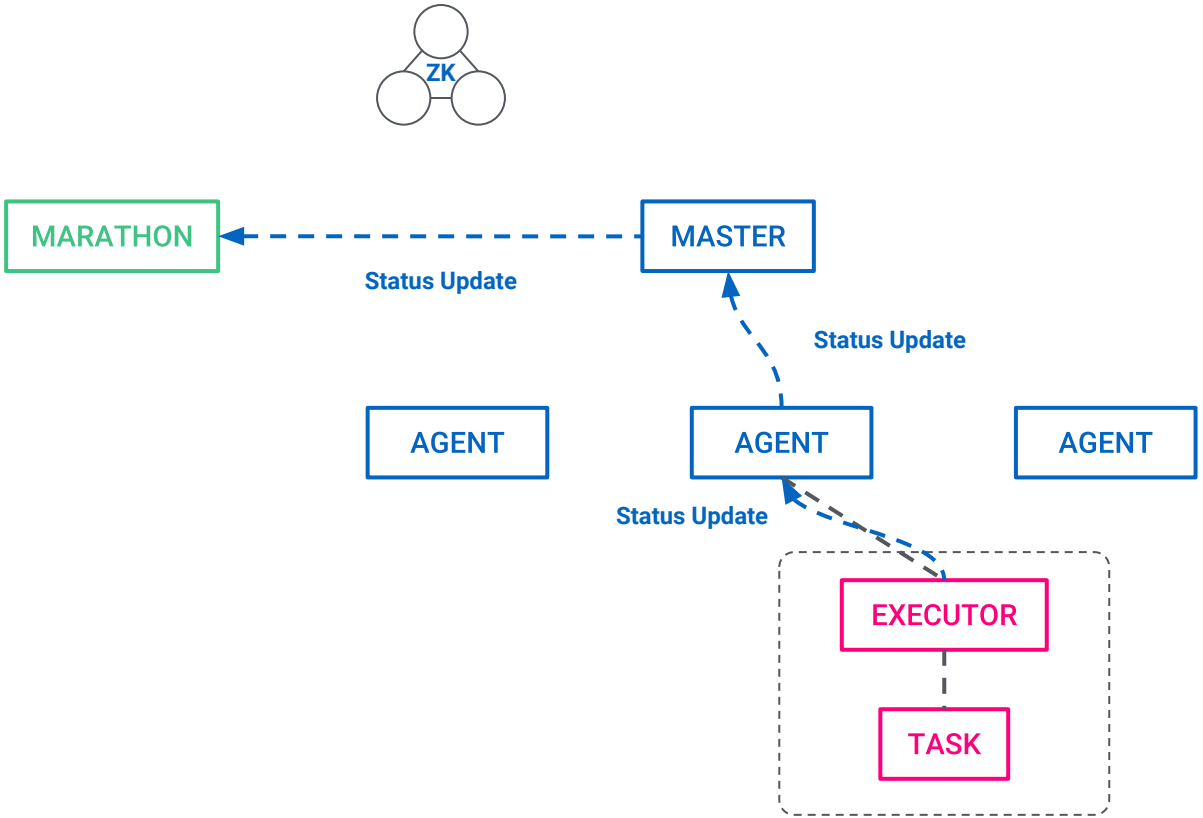
Task Failure



Task Failure

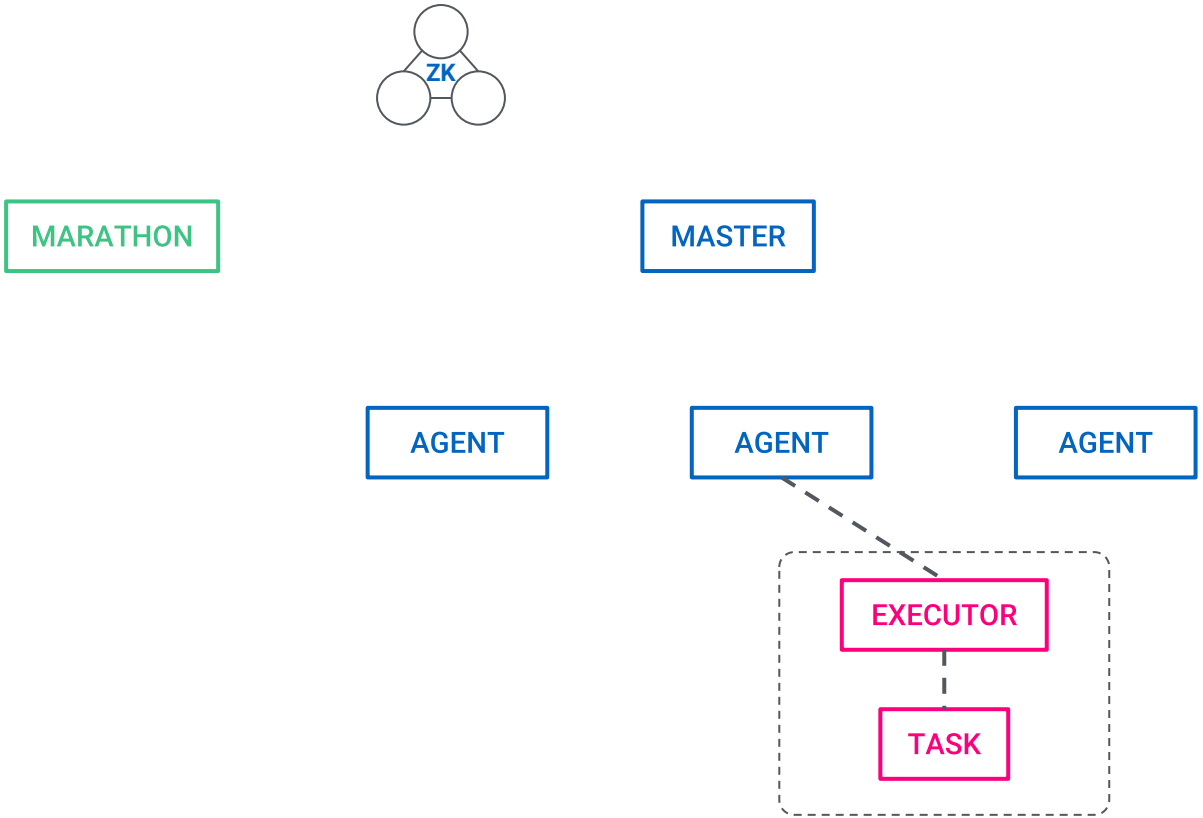


Task Failure

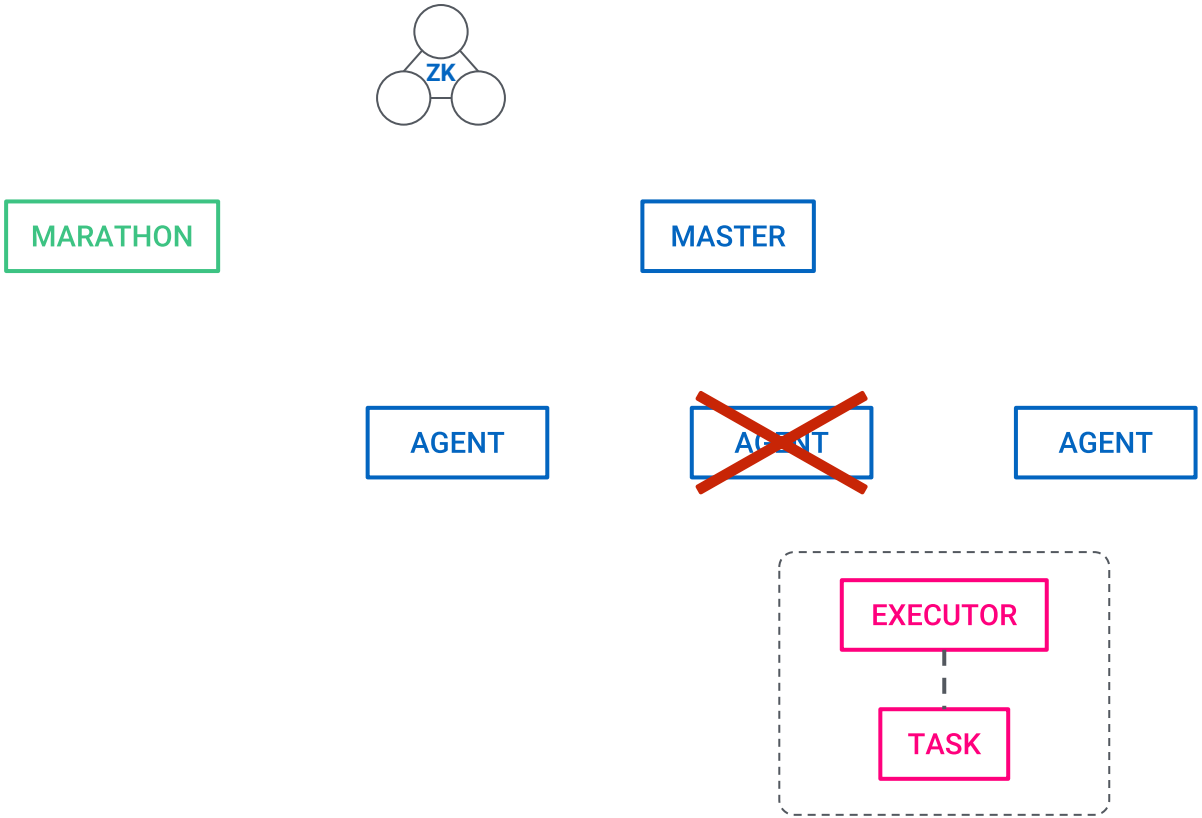


Mesos Agent Failure

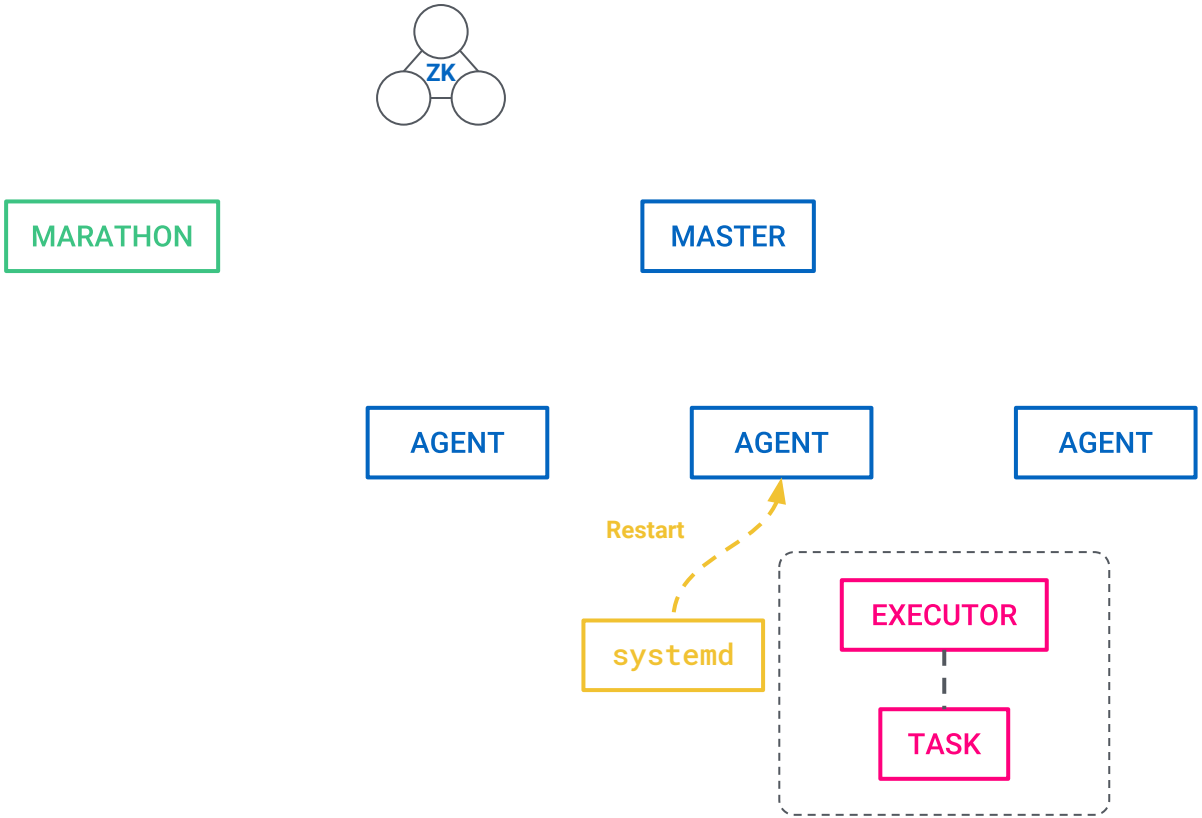
Mesos Agent Failure



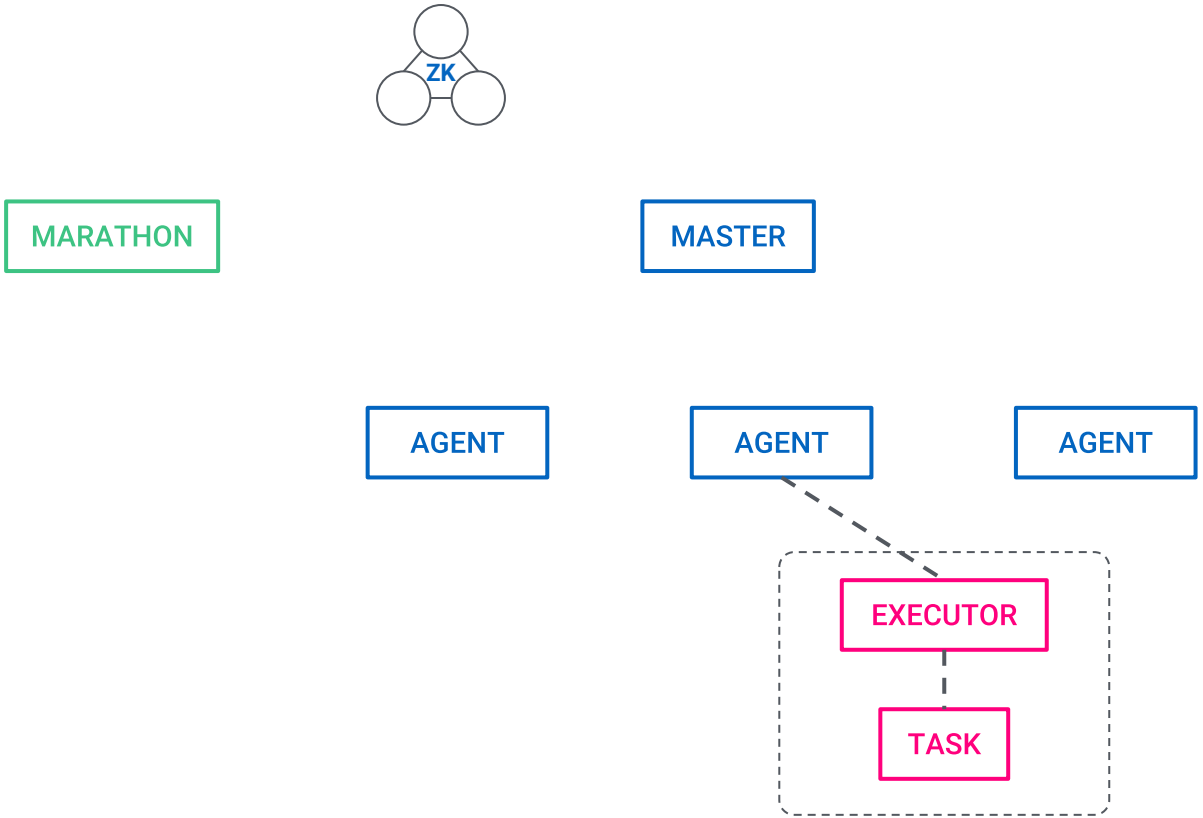
Mesos Agent Failure



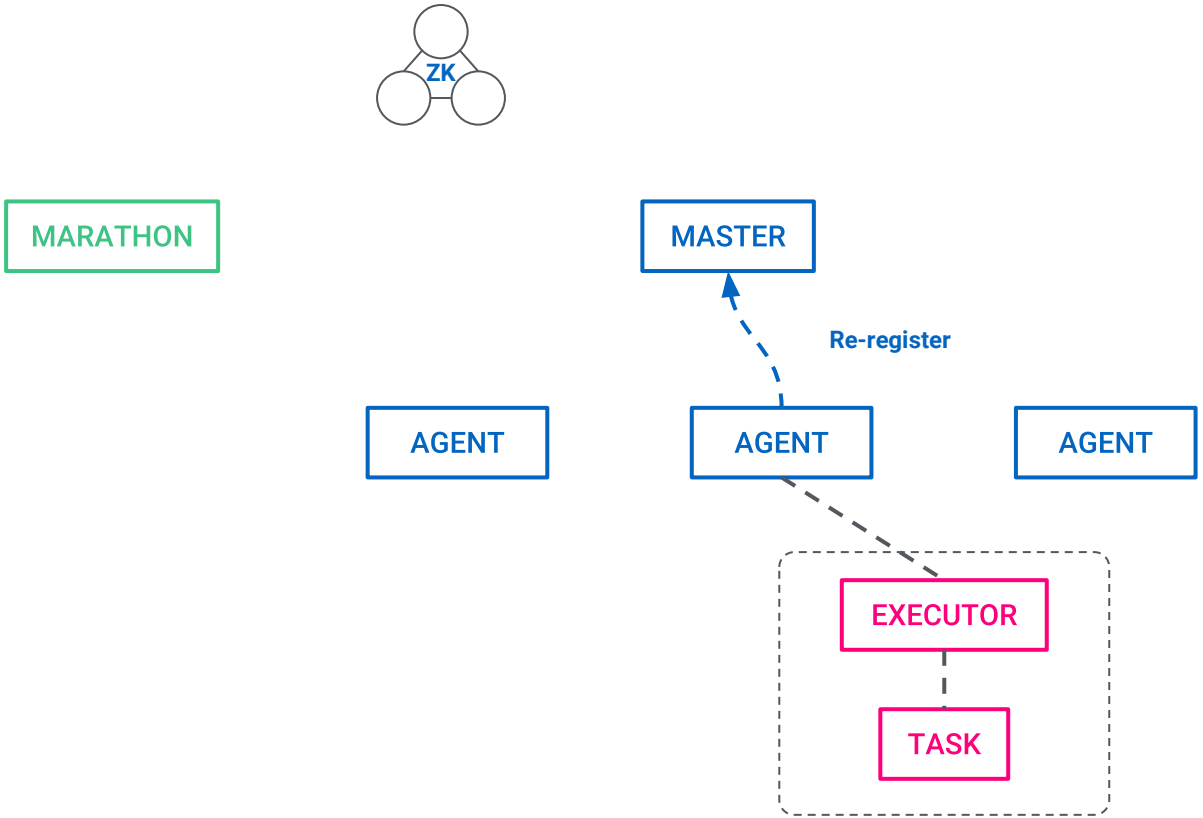
Mesos Agent Failure



Mesos Agent Failure

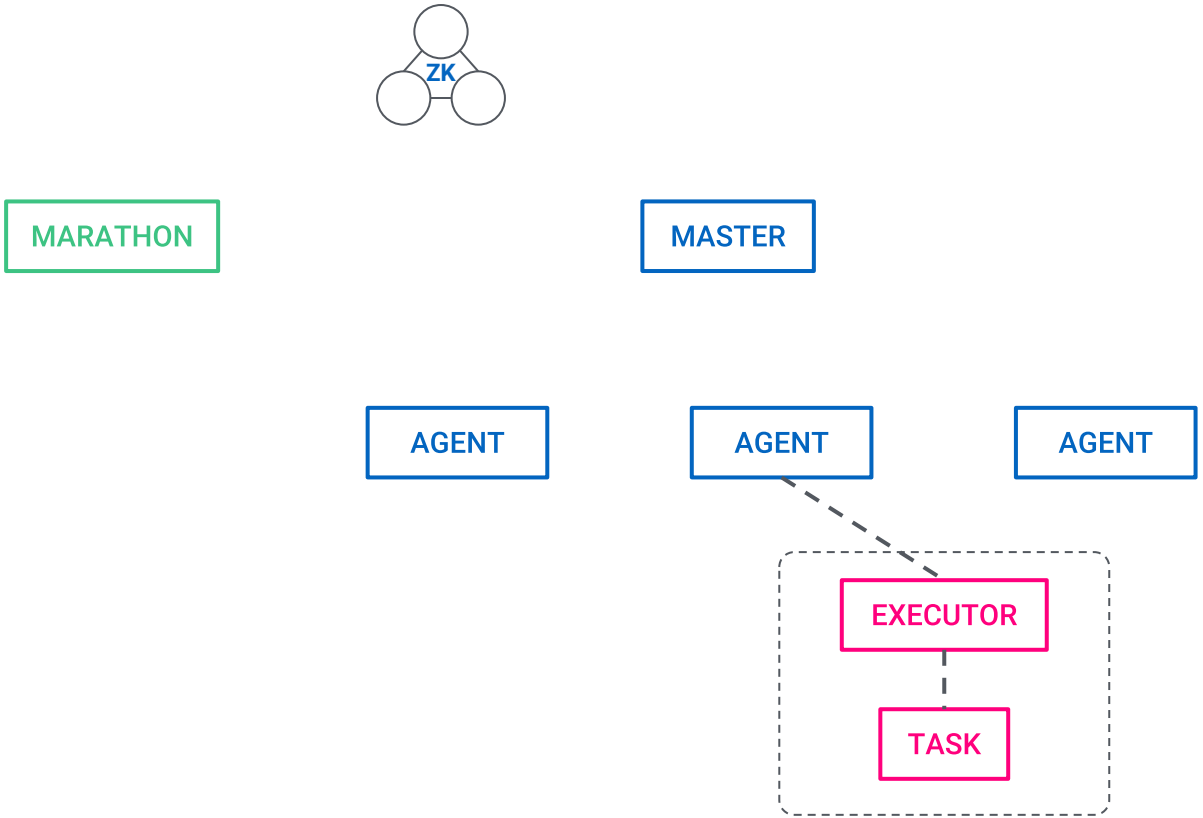


Mesos Agent Failure

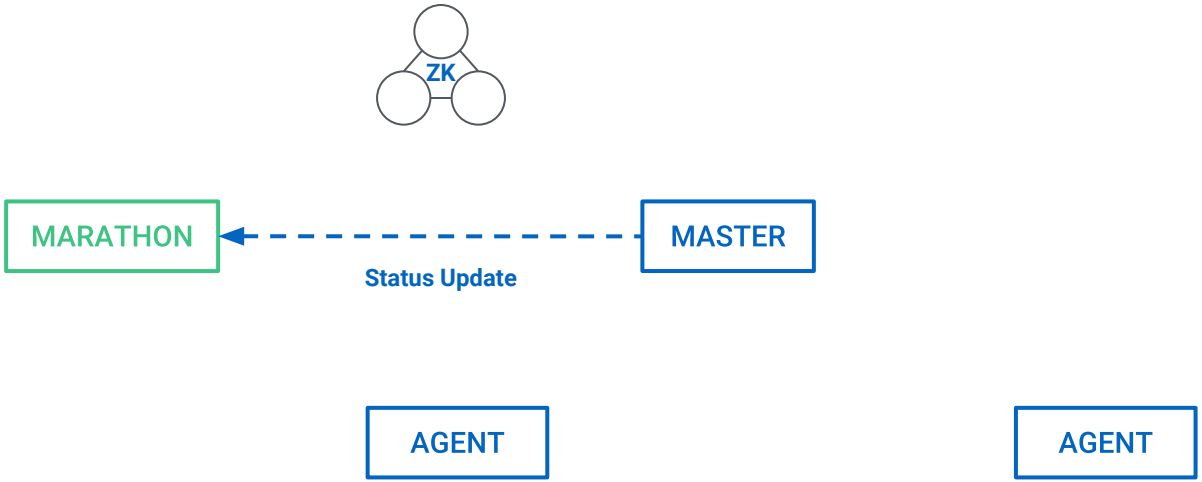


Agent Host Failure

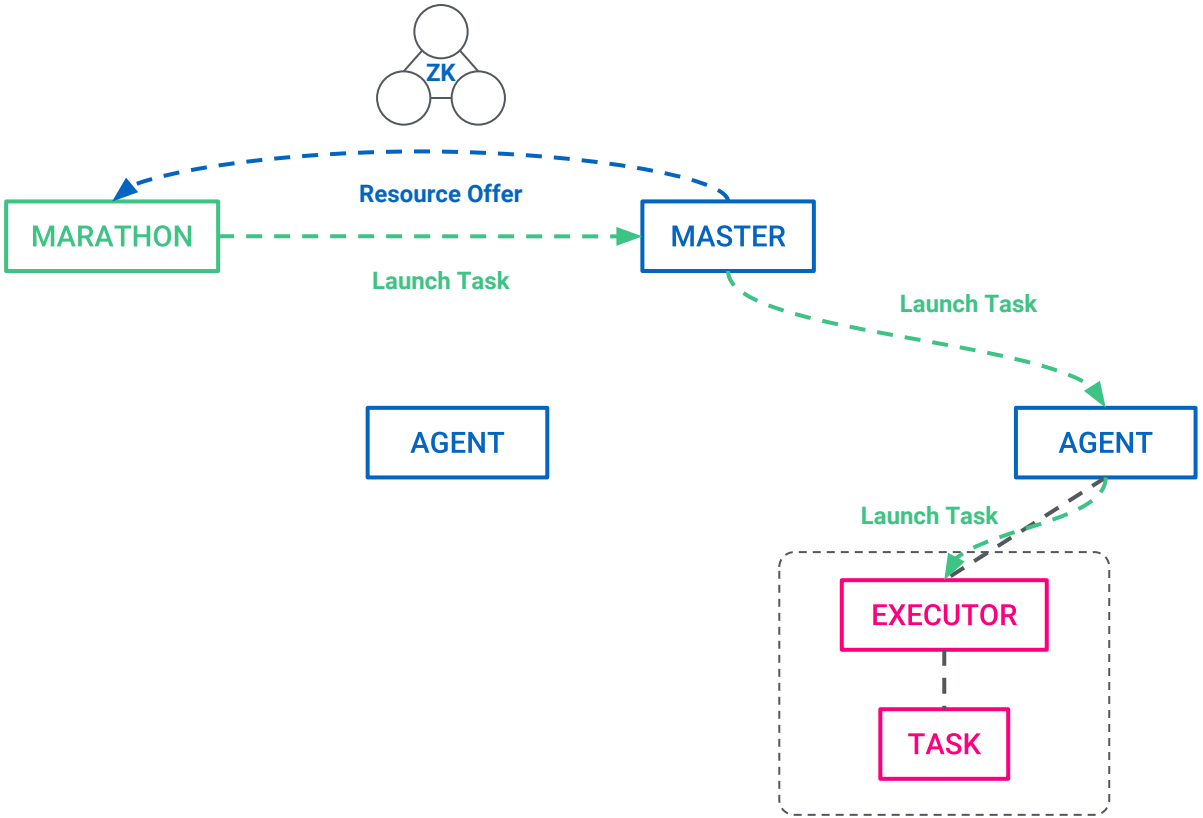
Agent Host Failure



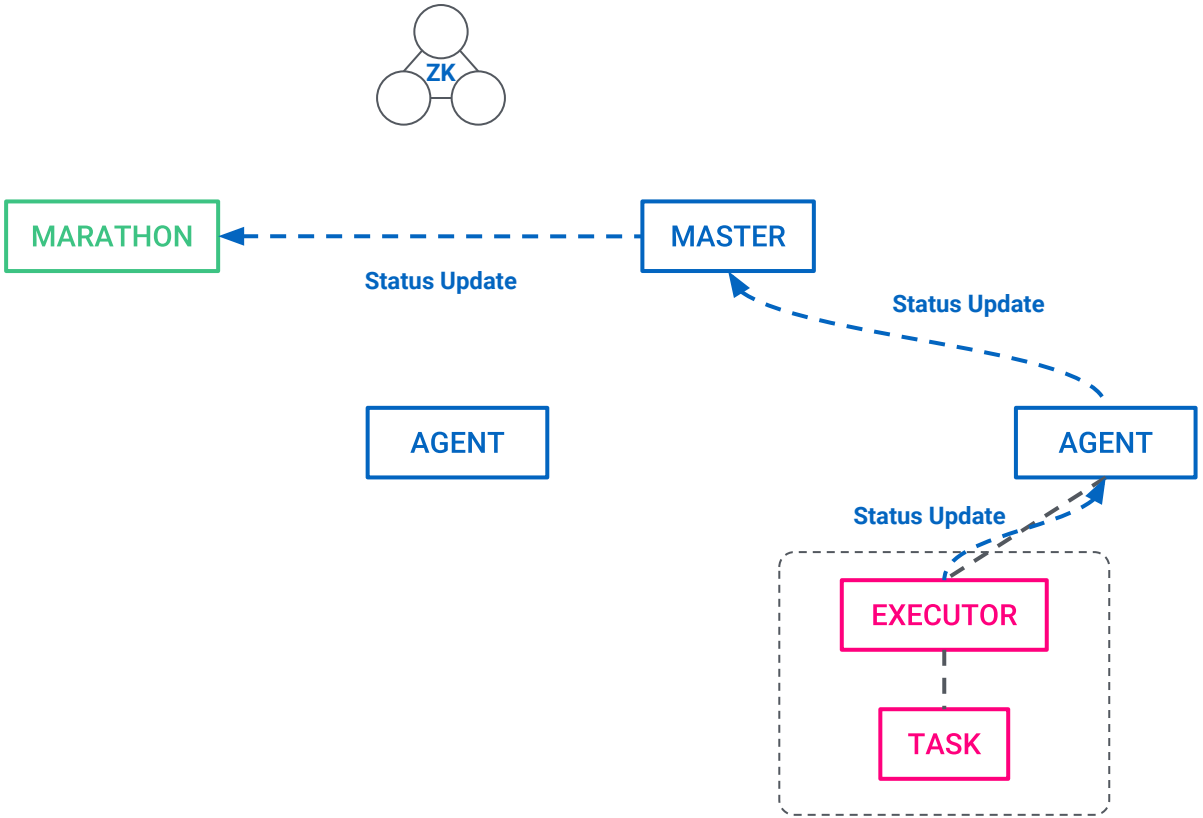
Agent Host Failure



Agent Host Failure

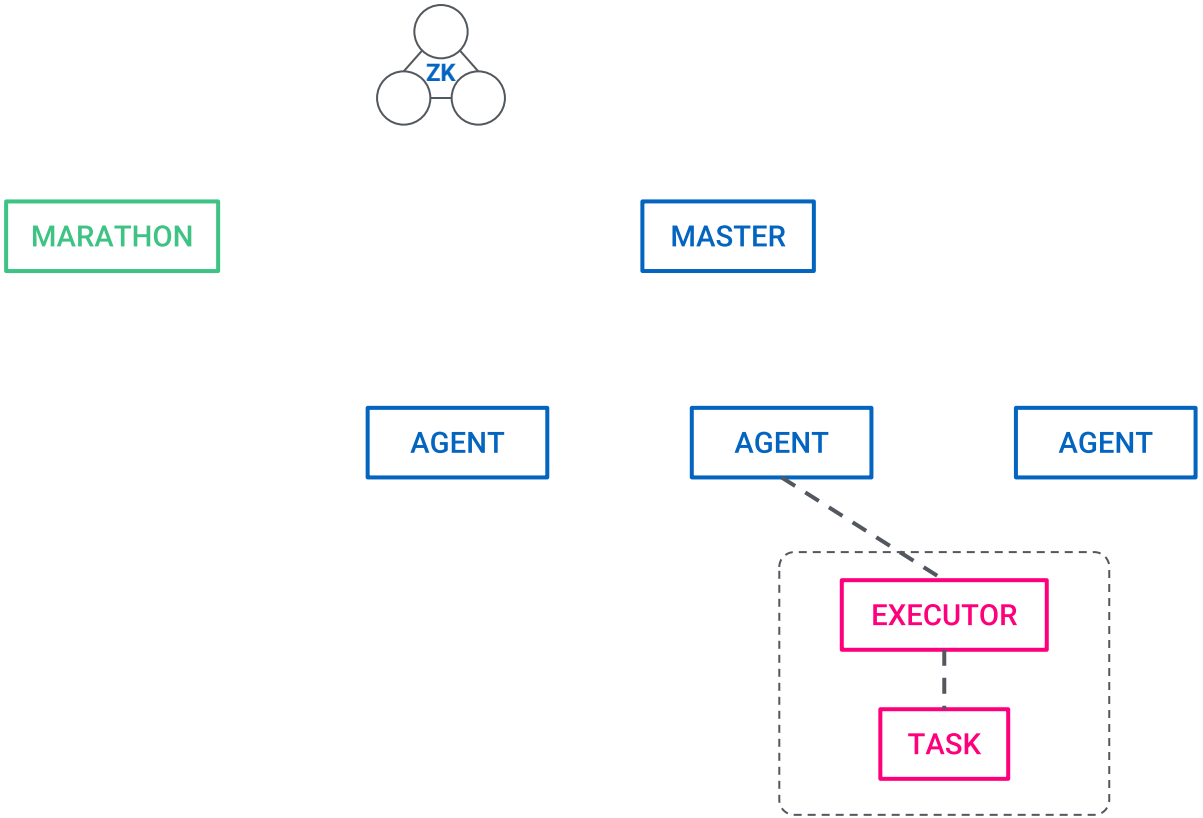


Agent Host Failure

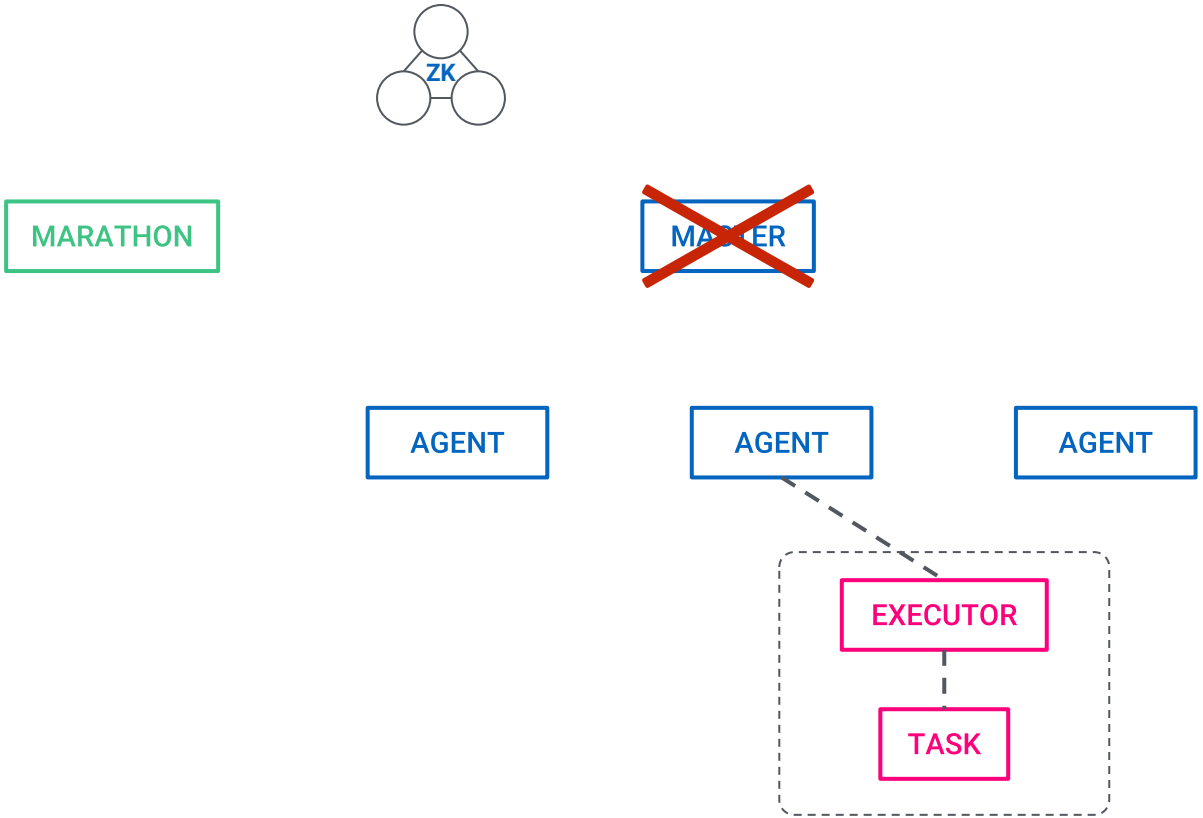


Mesos Master Failure

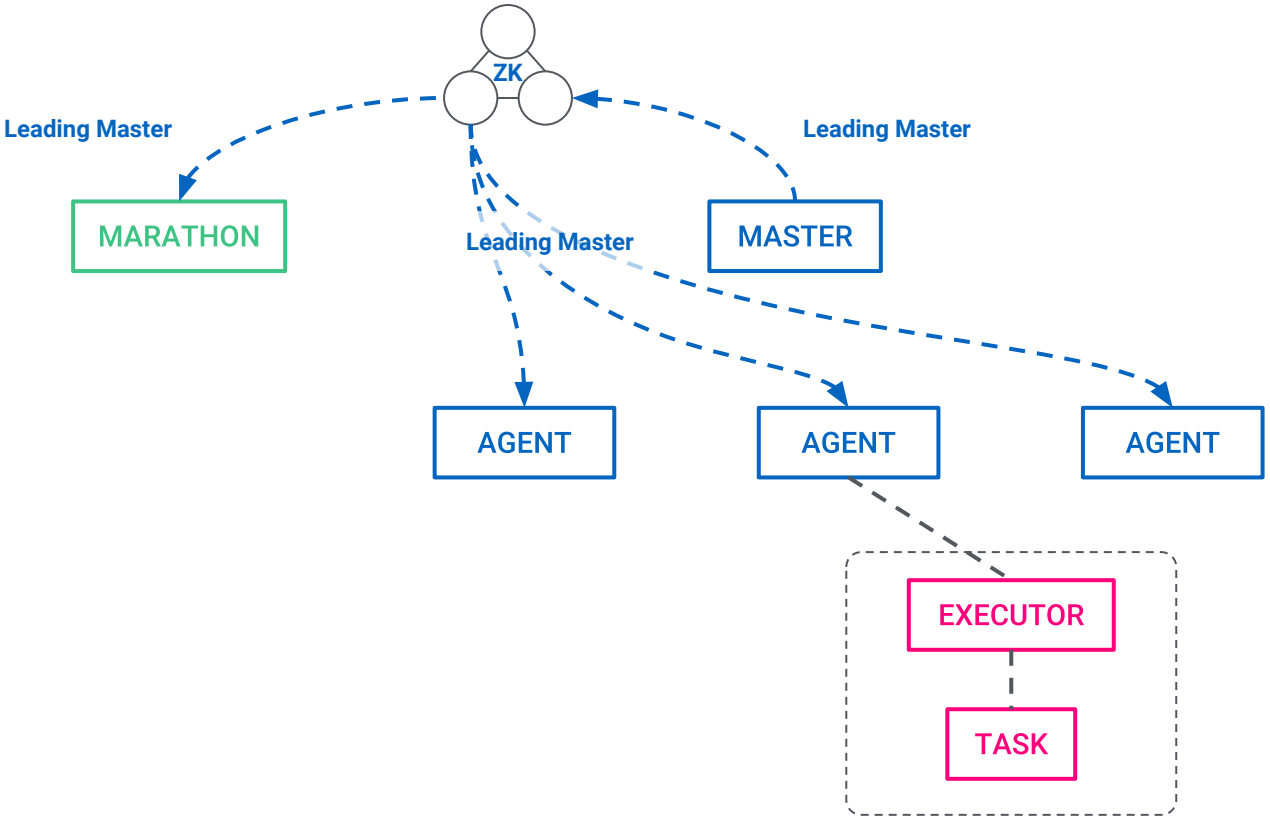
Mesos Master Failure



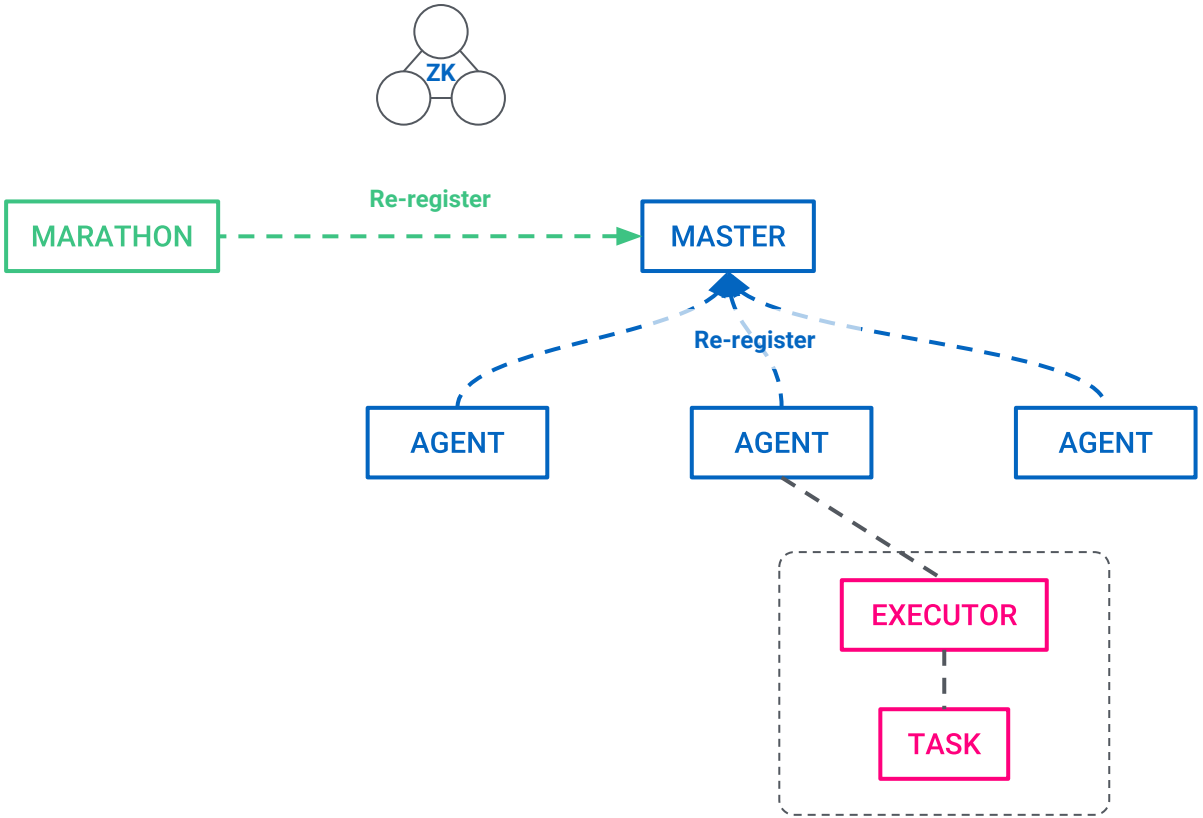
Mesos Master Failure



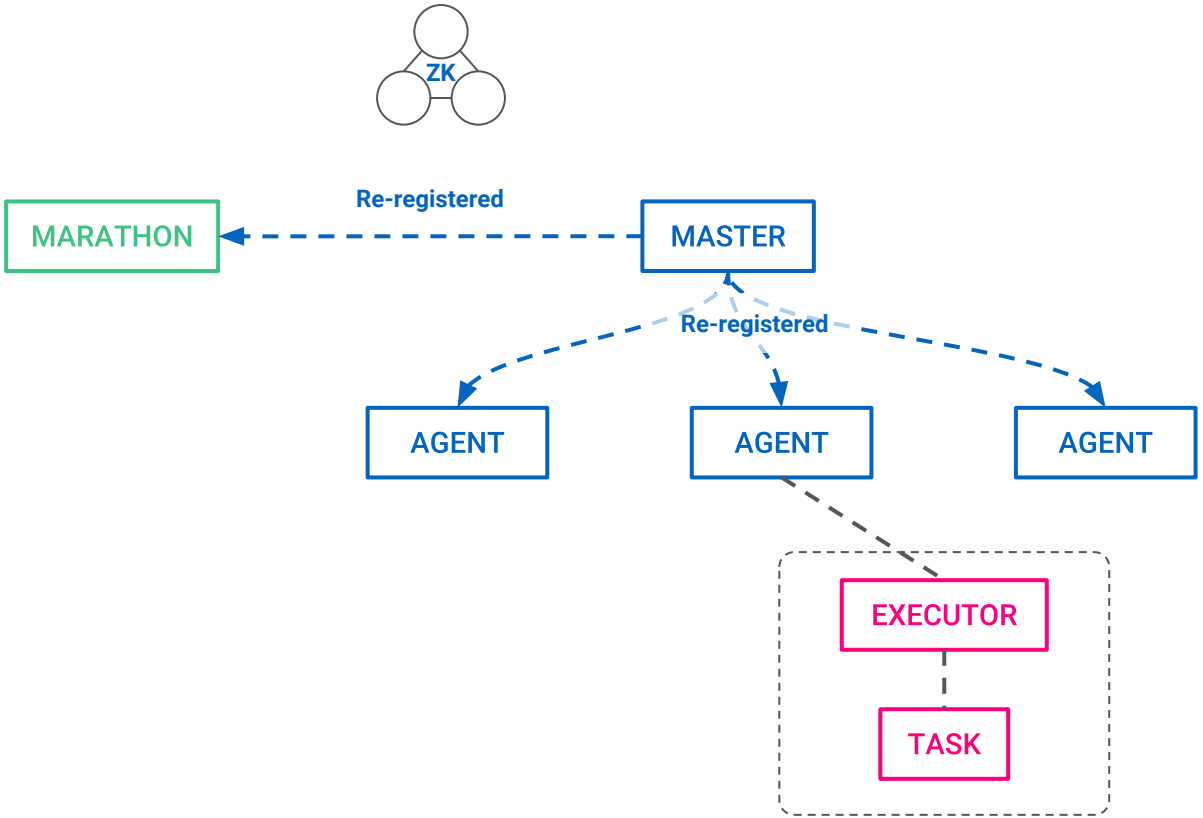
Mesos Master Failure



Mesos Master Failure

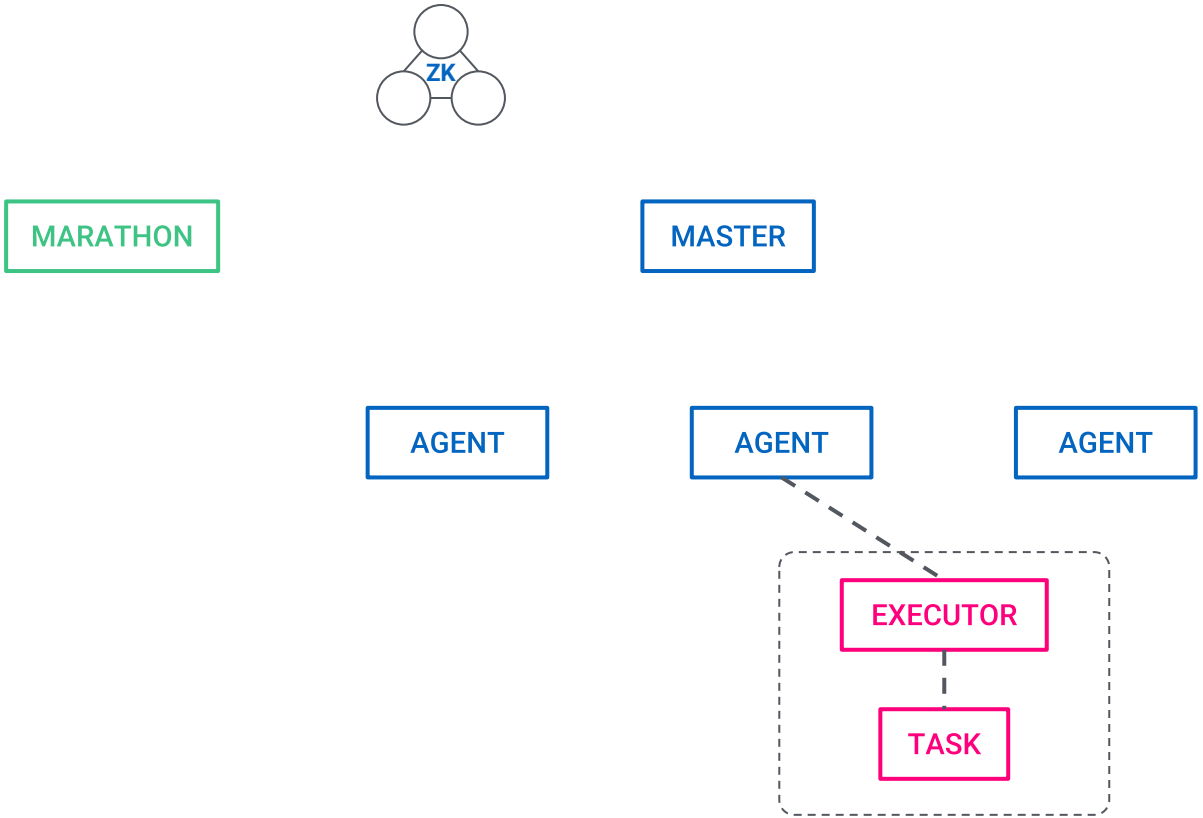


Mesos Master Failure

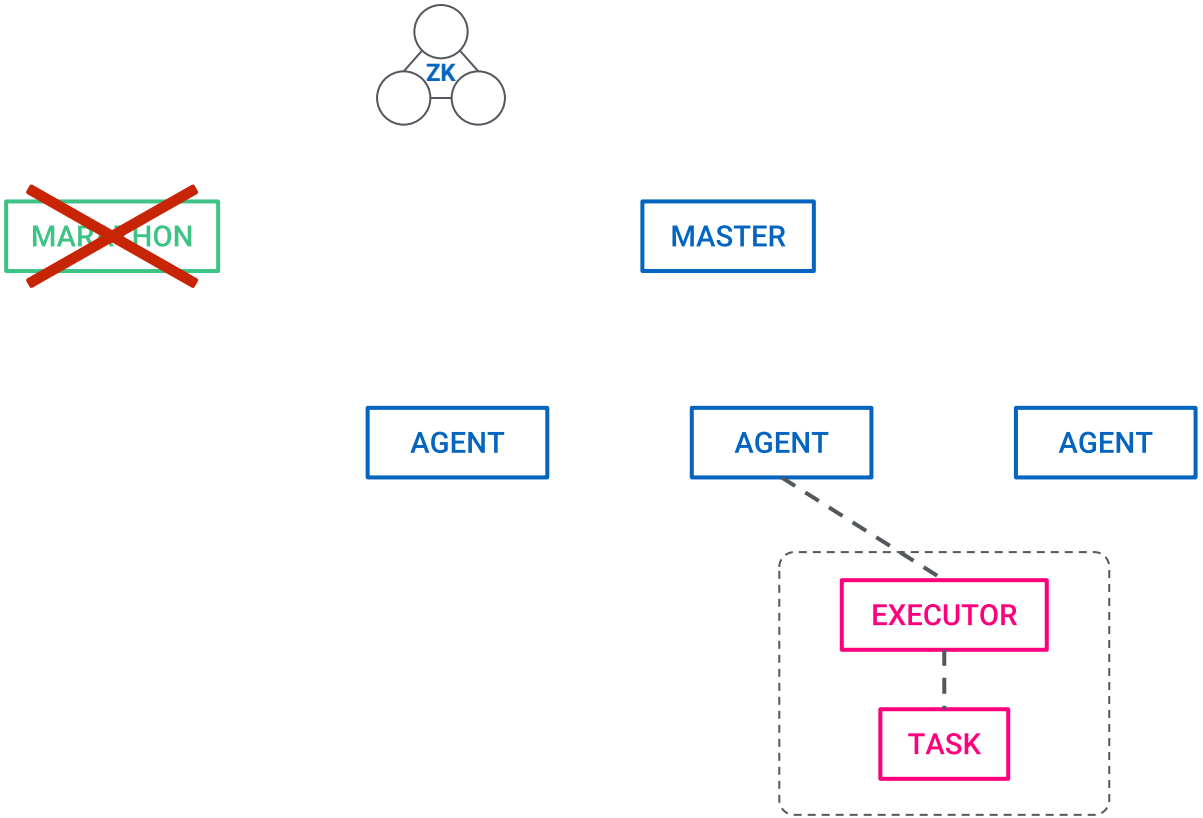


Scheduler Failure

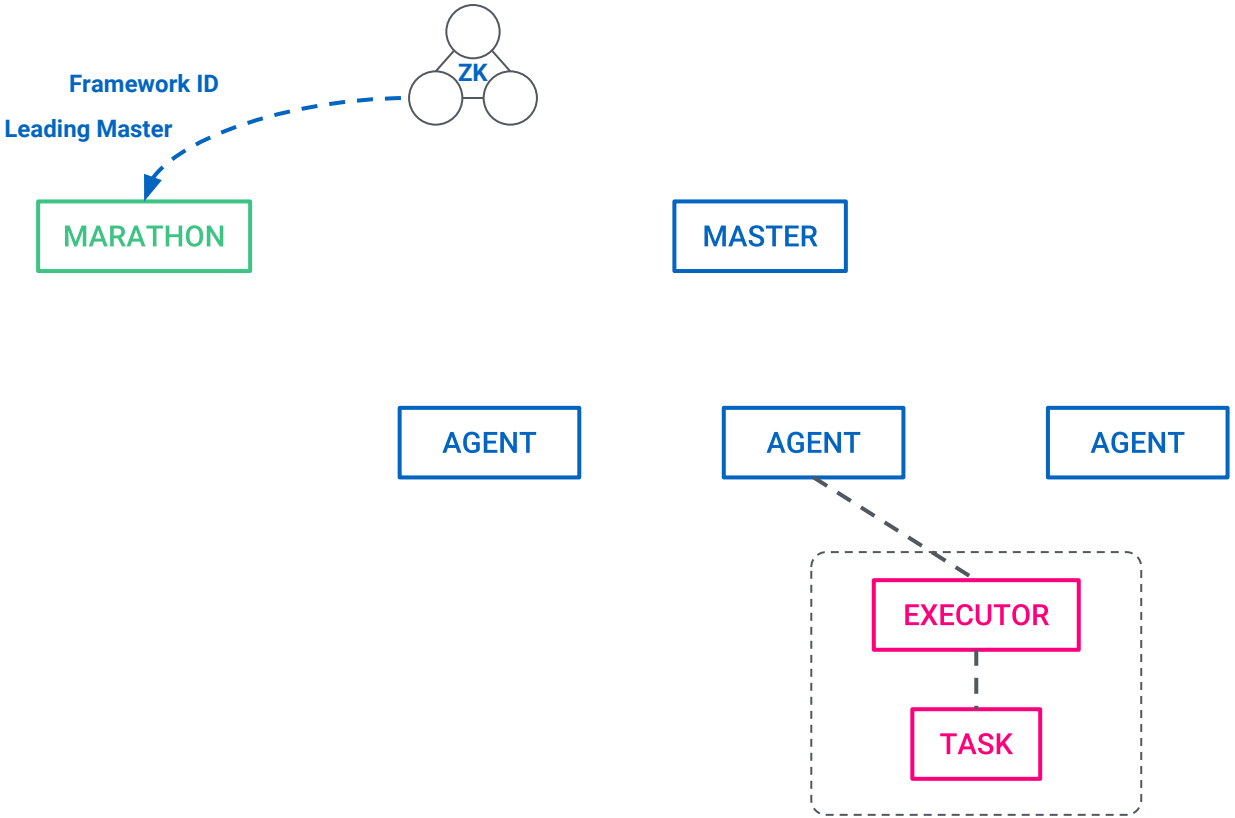
Scheduler Failure



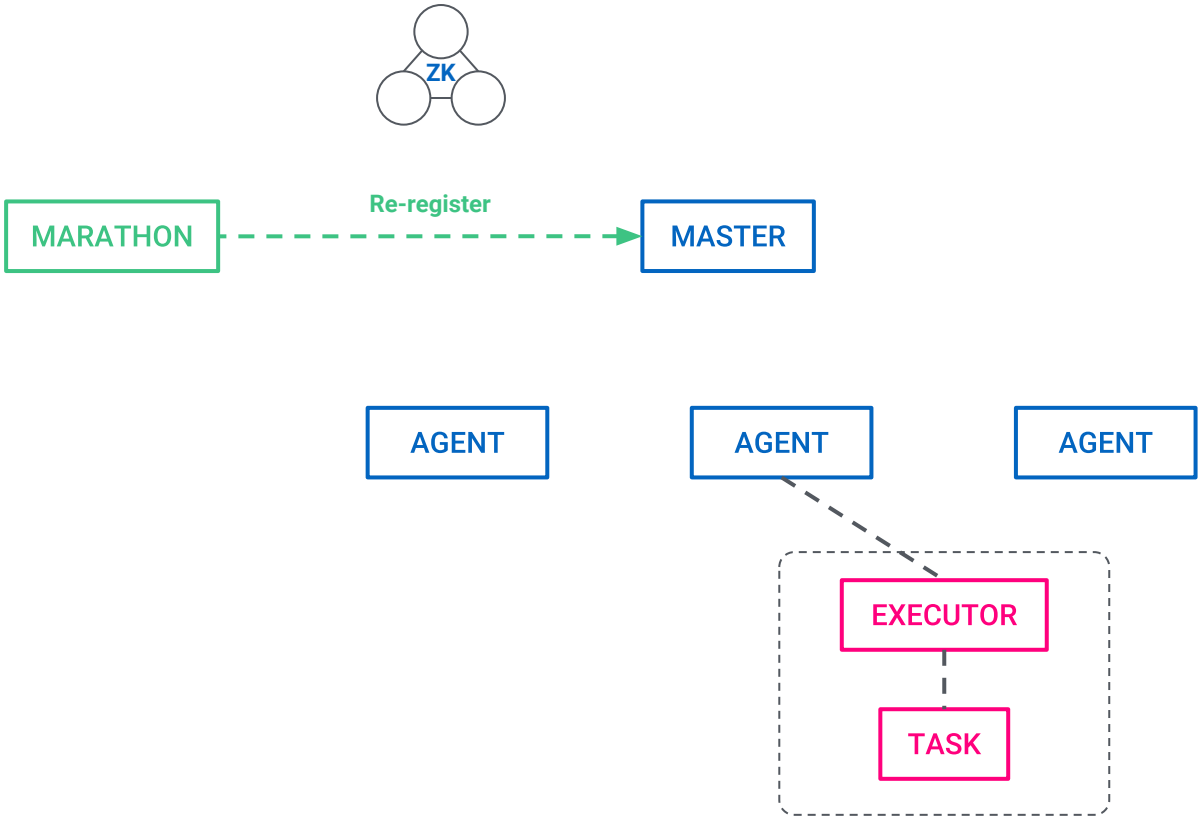
Scheduler Failure



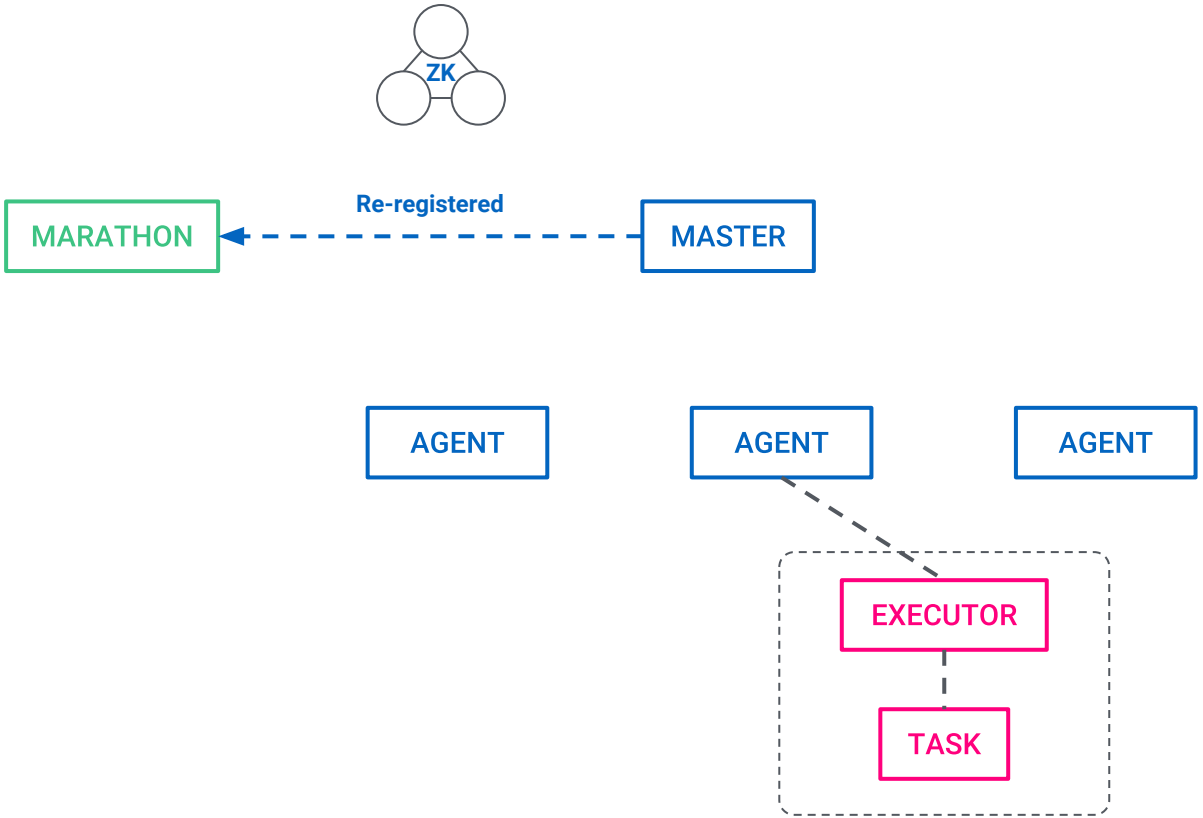
Scheduler Failure



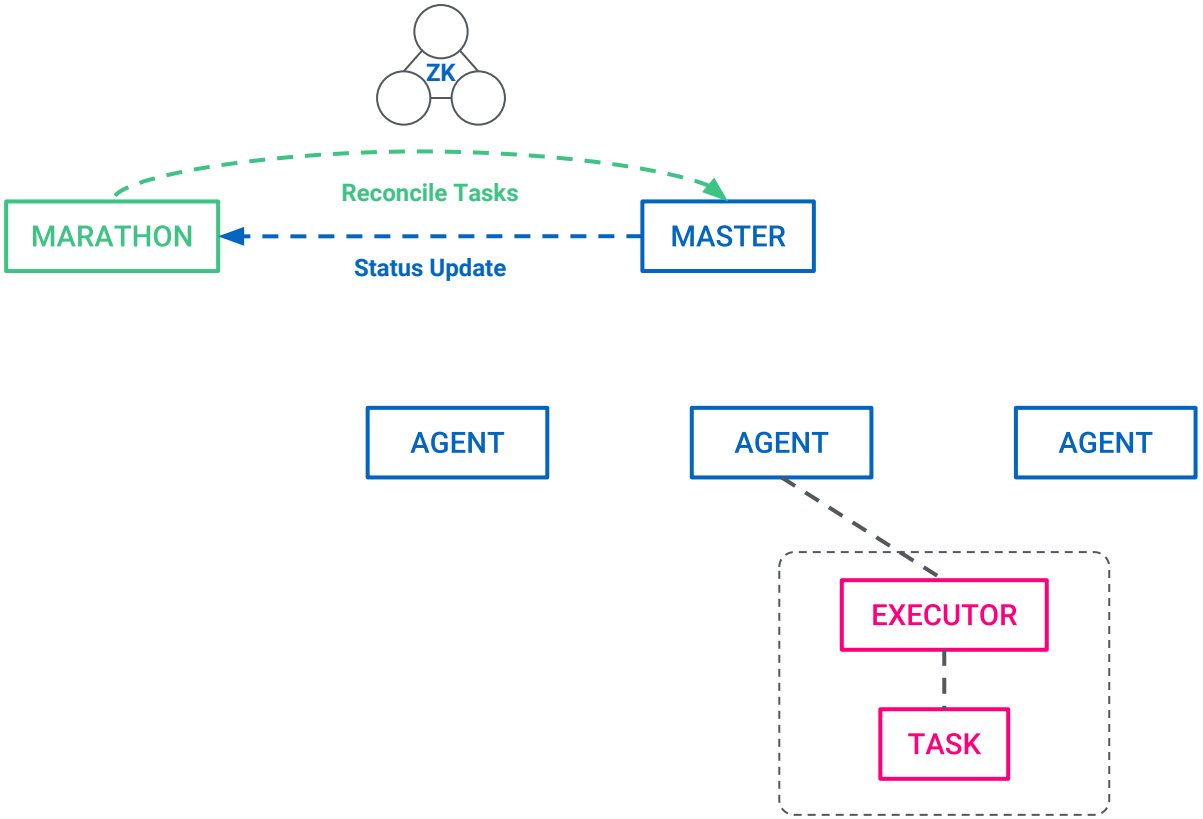
Scheduler Failure



Scheduler Failure



Scheduler Failure



Production Checklist

Mesos

- Configure ulimit settings appropriately
  - `open files`: A value of 32,000 (soft) and 262,144 (hard) have been used successfully in large production deployments
  - `max locked memory`: Increase to account for huge pages, if required
- Monitor both masters and agents for flapping (continuously restarting)
- Monitor the rate of changes in terminal task states, including `TASK_FAILED`, `TASK_LOST`, and `TASK_KILLED`
- Use five masters in production. Three is sufficient for HA in dev/stage/test
- Place master in separate racks or availability zones if possible

Mesos Agents

- Set agent attributes before you run anything on the cluster
  - Once an agent joins the cluster, change to it’s attributes may break recovery of containers running on the node
- Explicitly set the resources on the nodes to leave capacity for other services running outside the control of Mesos
  - e,g. Storage daemon that is running on the host operating system

Zookeeper

- Backup ZooKeeper snapshots and logs at regular intervals
  - Guano and zkConfig.py
- Marathon, Metronome, and other frameworks store their states in ZK
- Services launched on the cluster should not store state in the ZK ensemble that is in use by the DC/OS control plane processes
- Monitor ZK's JVM metrics, such as heap usage, GC pause times, and full-collection frequency
- Monitor ZK for number of client connections, total number of znodes, size of znodes (min, max, avg, P99), and read/write performance metrics

Getting Help

Getting Help

- Enterprise and community DC/OS docs
  - https://docs.mesosphere.com
- Github repos
  - https://github.com/mesosphere
  - https://github.com/dcos
- Mesosphere enterprise support
  - https://support.mesosphere.com (requires login)
- Community
  - https://dcos.io/community

MESOSPHERE

Appendix

Unit Name	Description	Node(s) of Execution M = Master A = Agent
dcos-adminrouter.service	Exposes a unified control plane proxy for components and services using NGINX	M
dcos-backup-master.service	Backup & restore service	M
dcos-bouncer.service	Controls access to DC/OS components and services by managing users, user groups, service accounts, permissions, and identity providers	M
dcos-ca.service	Issues signed digital certificates for secure communication	M
dcos-cluster-linker.service	Service for DC/OS Cluster Linker	M
dcos-cockroach.service	Database for the DC/OS IAM	M
dcos-cosmos.service	Installs and manages DC/OS packages from DC/OS package repositories, such as the Mesosphere Universe	M
dcos-diagnostics.service	Aggregates and exposes component health	M, A
dcos-exhibitor.service	Zookeeper supervisor service	M
dcos-history.service	Caches and exposes historical system state	M

Unit Name	Description	Node(s) of Execution M = Master A = Agent
`dcos-licensing.service`	Licensing audit service	M
`dcos-log-master.service`	Exposes master node and component logs	M
`dcos-marathon.service`	Container orchestration engine	M
`dcos-mesos-dns.service`	Domain name based service discovery	M
`dcos-mesos-master.service`	Distributed systems kernel	M
`dcos-metrics-master.service`	Exposes node metrics	M
`dcos-metronome.service`	Job orchestration	M
`dcos-net-watchdog.service`	Restarts dcos-net when it is unhealthy	M, A
`dcos-net.service`	A distributed systems & network overlay orchestration engine	M, A
`dcos-pkgpanda-api.service`	Installs and manages DC/OS components	M, A

Unit Name	Description	Node(s) of Execution M = Master A = Agent
`dcos-secrets.service`	Provides a secure API for storing and retrieving secrets from Vault, a secret store	**M**
`dcos-vault.service`	DC/OS Default Secret Store Backend	**M**
`dcos-adminrouter-agent.service`	Exposes a unified control plane proxy for components and services using NGINX	**A**
`dcos-log-agent.service`	Exposes agent node, component, and container (task) logs	**A**
`dcos-mesos-slave-public.service`	Distributed systems kernel public agent	**A**
`dcos-mesos-slave.service`	Distributed systems kernel private agent	**A**
`dcos-metrics-agent.service`	Exposes node, container, and application metrics	**A**
`dcos-rexray.service`	A vendor agnostic storage orchestration engine	**A**