NOTE: This is a public document.

[WIP] Mesos Cluster Maintenance

This is an updated design, based on the original design here.

Authors	Benjamin Mahler <bmahler@apache.org> Joseph Wu <joseph@mesosphere.io> Artem Harutyunyan <artem@mesosphere.io>
Revision	0.1
Status	Draft

JIRA Epic

MESOS-1474

Terminology

- *Operator* - A person or external tooling/scripts which manages the Mesos cluster.
- *Maintenance* - An operation that makes resources on a slave unavailable, after which, no guarantees can be made about the state of the slave.
- *Unavailability* - A period in which the associated resources may not be available. In the context of this design document, unavailability is due to maintenance. No guarantees can be made about the resources after the unavailability period.

- *Drain* - A period for the framework to reconcile its own requirements with a maintenance schedule. The framework should be taking preemptive steps to prepare for the unavailability, or communicate the framework's inability to conform to the maintenance schedule.
- *Inverse offer* - A mechanism for the master to ask for resources back from a framework (the opposite of regular offers). This notifies frameworks about unavailability and gives frameworks an opportunity to respond about their capability to comply. The same mechanism can be generalized for use towards non-maintenance goals such as reallocating resources or resource preemptions.
  Note about the term: The allocation of resources in Mesos is called an Offer; taking back allocated resources is the *inverse* of allocating. Additionally, frameworks can decline the inverse offer, hence the *offer*.
- *Maintenance schedule* - A list maintained by master (and persisted in the registry for failover) consisting of proposed unavailability for each of the slaves participating in the maintenance.

Disambiguation

- *Revocable resource* - A resource which could be taken back (using TASK_LOST and Reason) at any time. Frameworks may not be asked before revocable resources are taken away.
- *Preemption* - A concept where an allocator asks for some resources back from a framework.

Motivation

Operators regularly need to perform maintenance tasks on machines comprising a mesos cluster. Most mesos upgrades can be done without affecting running tasks, but there are situations where maintenance is task-affecting. For example:

- Host maintenance (e.g. hardware repair, kernel upgrades).
- Non-recoverable slave upgrades (e.g. adjusting slave attributes/resources).

In order to ensure that maintenance operators do not violate frameworks' SLAs, maintenance information needs to be conveyed from operators to frameworks and vice versa. Frameworks need to be aware of planned unavailability events and operators must be aware of frameworks' adaptability to maintenance.

Maintenance awareness allows frameworks to avoid downtime for long running tasks by (re)placing them on machines not undergoing maintenance. If all resources are planned for maintenance, then the framework can prefer machines scheduled for maintenance least imminently.

Maintenance awareness is also crucial when a framework uses persistent disk resources, to ensure that the framework is aware of the expected duration of unavailability for a persistent disk resource (e.g. using 3 1TB replicas, don't need to replicate 1TB over the network when only 1 of the 3 replicas is going to be unavailable for a reboot (< 1 hour)).

Goals

- Enable maintenance-aware scheduling in frameworks.
- Enable maintenance with persistent resources.
- Enable maintenance to be controlled by an operator.
- Produce metrics that help operators decide whether a maintenance schedule is appropriate or not.

Non-Goals

- Determine optimal maintenance schedules:
    - This is a hard problem, and requires evaluation of all frameworks' fault constraints. For now, the creation of the schedule is in the hands of the operator. Operators should not be creating schedules with maintenance occurring concurrently across fault domains. Note that there is nothing preventing frameworks from exposing information to guide schedule creation.
- Perform the maintenance actions through Mesos:
    - For now, the actual maintenance procedure that is needed will be done externally through an operator.

Constraints

- Ensure frameworks can decline maintenance (e.g. SLA violation).
- Ensure operators can enforce maintenance when it's mandatory.

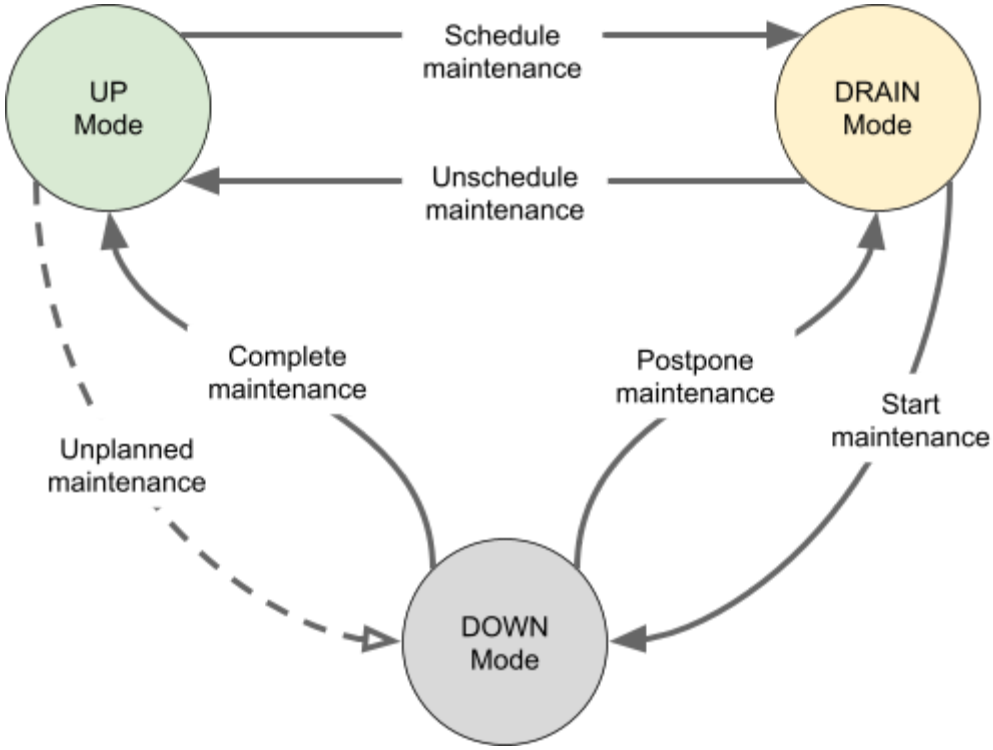
Architecture

Maintenance state transition

From the maintenance operator's perspective a slave can be in one of the following modes/states:

- UP Mode - the slave operates normally, there is no maintenance scheduled.
    - Transition from UP Mode to DRAIN Mode is performed when the operator schedules maintenance.
    - Transition from UP Mode to DOWN Mode is used in an emergency case and is outside of the normal flow of operations.

- DRAIN Mode - the slave is part of a maintenance schedule. It is still operational, however all its offers carry maintenance information (see Unavailability). Any resources reserved or in use on the slave will prompt inverse offers to be sent.
    - Transition from DRAIN Mode to DOWN Mode is initiated by the operator at or after the start of a scheduled maintenance window.
    - Transition from DRAIN Mode to UP Mode is done when the corresponding maintenance schedule is cancelled.
- DOWN Mode - the slave is down for maintenance. No offers are sent, no tasks are supposed to be running on the slave.
    - Transition from DOWN Mode to DRAIN Mode is performed when there is a need to delay (but not to cancel) the maintenance, and the operator wants to utilize resources of the node during the delay.
    - Transition from DOWN Mode to UP Mode is performed when either the corresponding maintenance schedule is cancelled or when the maintenance process is completed.

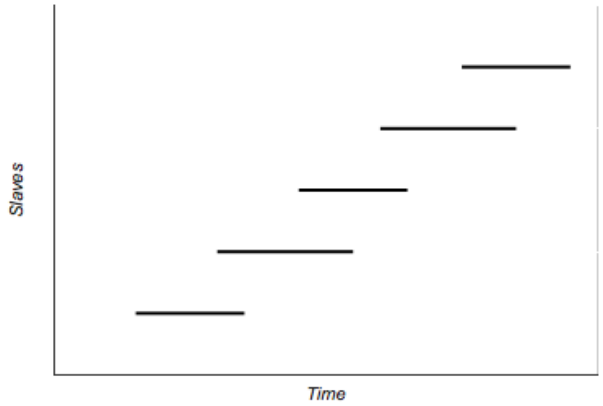


All state transition are a result of explicit requests initiated by the operator. Mesos itself never initiates the change of the state.
For example, to plan and perform maintenance on a slave, the operator must first schedule the maintenance, putting the slave into DRAIN Mode. When the time of maintenance arrives, the operator must make another call to deactivate the slave, putting the slave into DOWN Mode.

Maintenance workflow

A typical maintenance workflow represents a single cycle through the state transition diagram.

- The maintenance workflow is initiated by the operator by sending a maintenance schedule to the master. A schedule consists of a list of proposed downtime intervals for some set of slaves.
  In a production environment the schedule will typically be constructed in a way to ensure that at any given point in time the number of slaves that are operational (i.e. are not part of maintenance) are enough to ensure uninterrupted operation of service (e.g. no overlap across fault domains, like “rack”). Below is the visualization of maintenance schedule timeline.



  The interval is a conservative estimate set by the operator, used only as a scheduling hint for frameworks. It is up to the individual framework to decide how to act given maintenance information.
    - For fault tolerance, the maintenance schedule is persisted in the replicated registry.

- All resources scheduled for maintenance are tagged with an upcoming unavailability:
    - For the resources that are currently in use, inverse offers are sent with the unavailability. Inverse offers are sent as long as the slave is scheduled for maintenance.
    - For newly offered resources, offers include the unavailability. If the offer is accepted, inverse offers are sent immediately.
    - For outstanding offers, the previous offers are rescinded and resent with the unavailability.

- Frameworks can perform scheduling in a maintenance-aware fashion:
    - Slaves with unavailability are less preferable for long running tasks.

    - Stateful tasks on soon-to-be-unavailable slaves can be migrated to available slaves.
    - Accepting or rejecting an inverse offer does not result in immediate changes in unavailability schedule, or in the way mesos acts. Inverse offers merely represent some extra information that frameworks may find useful. In the same manner a rejection or acceptance of an offer is a hint for an operator. The operator may or may not chose to take that hint into account.

- Operators use the master's maintenance endpoints to communicate to frameworks about maintenance. The master uses inverse offers to convey this information to frameworks.
    - Inverse offers are sent by the master to frameworks using resources to-be-maintained.
    - A framework can accept an inverse offer, which indicates that the framework is ok with the maintenance *currently*, as in taking these resources away is not going to critically affect the framework.
        - A filter attached to the inverse offer can be used by the framework to control when it wants to be contacted again with the inverse offer, since future circumstances may change the viability of the maintenance schedule. The "filter" for InverseOffers is identical to the existing mechanism for re-offering Offers to frameworks.
        - An acceptance also signifies that the framework can kill tasks, shut down executors, and/or unreserve resources prior to the maintenance window. The framework should ideally make sure that, by the time of the maintenance start, there are no tasks running on the corresponding system.
    - An inverse offer can be rejected with a reason based on failure constraints (e.g. SLA, replica availability, etc).

- When the unavailability time is reached:
    - The master keeps the slave in DRAIN mode until an operator makes a decision. This means that the operation of the slave is not disrupted in any way and offers (with unavailability information) are still sent for this slave.

- When the operator transitions into DOWN mode:
    - **If there are no tasks running on the slave**, and there are no resource reservations on the slave, then the operator can safely put a slave into DOWN mode.
    - **If there are still tasks running** on the slave or there are resource reservations on the slave, then the operator can put a slave into DOWN mode regardless. That kills all tasks (TASK_LOST) running on the slave.
        - The operator can determine if this is safe based on (1) what is currently running and (2) the frameworks' latest reasons for declining the inverse

offers. Alternatively an operator can wait until the resources are freed by the frameworks or re-schedule the maintenance.
        - Resource reservations will still be valid after the slave exits the DOWN mode.
    - After entering DOWN mode, the slave is deactivated and its resources are not included in subsequent resource offers.

- When maintenance is complete, or if maintenance needs to be cancelled, the operator will unschedule the slave from maintenance.
    - The “duration” of the maintenance is a guess made by the operator. Hence, the slave is not automatically transitioned out of DOWN mode.
    - The operator must flip a flag in the registry to note that the maintenance was finished (or cancelled).
    - Frameworks are informed about the cancellation of the maintenance.
        - New offers are no longer tagged with unavailability.
        - Existing offers with unavailability get rescinded and reissued.
        - Inverse offers get rescinded.
    - Frameworks are not explicitly informed about the completion of a maintenance window.
        - Once the slave re-registers, its resources will be offered again, not tagged with unavailability.
        - If the slave actually shut down (SLAVE_LOST), then any existing or inverse offers would already have been invalidated.

Maintenance State

- Persisted in the replicated registry:
    - The maintenance schedule.
    - State of each slave’s mode.
- Kept in-memory by the master:
    - Metrics about the accept/decline status of inverse offers.
    - Filters for inverse offers.

Safety Mechanisms

- A slave may only have a single unavailability window. Schedules which have a slave with multiple windows will be rejected.
- Metrics per slave:
    - Inverse offer acceptance or rejection with reasons.

Proposed MVP

For an MVP we aim for basic functionality:

- Operators can schedule/unschedule maintenance on a slave.
- Operators can put slaves in/out of DOWN mode.
- Given a schedule, all frameworks affected by the schedule (i.e. reserved resources on the slave or running tasks on the slave) are given inverse offers for each slave they use.
- Frameworks reply to an inverse offer by either accepting or declining them.
- Maintenance information is persisted in case of master failovers. This will be stored in the replicated registry, (in the same key as the master/slave info).
- No authorization. Everybody who has access to the Mesos Master can schedule maintenance. The maintenance endpoints can be disabled via the `--firewall_rules` master flag.
- The time(s) in the maintenance schedule will not necessarily be synchronized across masters and slaves.
- Frameworks not affected by the maintenance schedule are not given inverse offers.
- The reasons for rejecting inverse offers will be logged.

API

[WIP] HTTP API

NOTE: Mesos HTTP API is still a work in progress. This will be reviewed later when the internal implementation is complete and more information on the Mesos HTTP API is available (for example, it's still not clear what the accepted way of versioning endpoints is going to be).

- GET /maintenance: returns a list of all schedules that are in effect and their respective status.
- POST /maintenance: schedule a list of hosts for maintenance.
- DELETE /maintenance: clear a list of schedule(s).
- POST /maintenance/<slave>: change the maintenance mode of a slave.

Framework API

```
message Unavailability {
  // The approximate start time of the unavailability.
  // If this is in the past, the unavailability would be expected
  // at any time (i.e. Now).
  required Time start = 1;

  // The approximate duration of the unavailability,
  // if this is a transient unavailability.
  // Leave blank if unknown or indefinite.
  optional Duration duration = 2;
}
```

```
message Offer {
  required OfferID id = 1;
  required FrameworkID framework_id = 2;
  required SlaveID slave_id = 3;
  required string hostname = 4;
  repeated Resource resources = 5;
  repeated Attribute attributes = 7;
  repeated ExecutorID executor_ids = 6;
  optional URL url = 8;

  // The resources specified in this offer will become unavailable
  // at the specified start time and for the specified duration. Any
  // tasks launched using these resources might get killed when
  // these resources become unavailable.
  optional Unavailability unavailability = 9;
}

// A request to "deallocate" or "return" any resources already
// being consumed by the framework.
message InverseOffer {
  required OfferID id = 1;
  required FrameworkID framework_id = 2;

  // The slave ID if the resources need to be released on a particular slave.
  optional SlaveID slave_id = 3;

  // The resources specified in this offer will become unavailable
  // at the specified start time and for the specified duration. Any
  // tasks running using these resources might get killed when
  // these resources become unavailable.
  required Unavailability unavailability = 4;

  // Note: The following fields represent future work which may re-use the
  // InverseOffer primitive.  For example, the allocator may want to narrow
  // in on a specific set of resources, or a specific set of tasks.

  // Specific resources that need to be released by the framework.
  repeated Resource resources = 5;

  // The executor and task IDs if the resources need to be released on specific
  // executors and/or tasks.
  repeated ExecutorID executor_id = 6;
```

```
  repeated TaskID task_ids = 7;
}
```

New-style Event/Call API:

```
message Event {
  // Re-use the OFFERS Event for InverseOffers.
  message Offers {
    repeated Offer offers = 1;
    repeated InverseOffer inverse_offers = 2;
  }
}

message Call {
  // Re-use the ACCEPT Call, as it currently exists.
  message Accept {
    repeated OfferID offer_ids = 1;
    repeated Offer.Operation operations = 2;
    optional Filters filters = 3;
  }

  // Re-use the DECLINE Call, with a new "reason" field.
  // Offers and inverse offers can be declined, including a
  // reason message for each decline.
  message Decline {
    repeated OfferID offer_ids = 1;
    repeated Reason reasons = 3; // reasons[i] is for offer_ids[i]
    optional Filters filters = 2;

    // Possible common scenarios for a framework to reject an InverseOffer.
    message Reason {
      enum Type {
        // A few example types that may be included in the future.
        // The MVP will not have specific reasons.
        SLA_VIOLATION = 1;
        QUOTA_NOT_MET = 2;
        ...
        OTHER = 99;
      }

      required Type type = 1;
      optional string message = 2;
    }
```

```
  }
}
```

Existing Scheduler API:

- Introduce separate callback for 'inverseResourceOffers'.
- Introduce 'declineOffers' call overload, with additional 'reason' argument.

Future (post MVP) work

- **[IMPORTANT]** Frameworks use Accept filters to control when (and if) do they want to receive inverse offers again. This item *may* be included in the MVP, as the implementation is theoretically simple and well understood.
- Add permission for maintenance scheduling to ACL's, including authentication/authorization of HTTP endpoints.
- (Limited) Versioning for schedules.
- Support for partial reclamation of resources.
- Support resource reclamation in allocation modules.
- Add a "Drained" call to the Even/Call API so that frameworks can explicitly tell the master that they have drained a slave. Otherwise, it may be possible to infer slave drainage incorrectly (task failed but framework intended to restart the task).
- Check if time synchronization between masters/slaves is important for maintenance schedules. If so, investigate what can be done about it.
- A more sophisticated allocator may send InverseOffers to frameworks unaffected by maintenance, say, in order to free up resources for an affected framework.
- The reasons for rejecting inverse offers can be kept in a more accessible format, which would give operators more complete information.
- Differentiate between revocable and dynamically reserved resources. Some maintenance operations might not need to revoke reserved resources. For example, a persistent volume might not be destroyed during the maintenance.
- Check maintenance schedules for validity in regards to meeting the Quota.
- Implement "whenever you (framework) can get your tasks off of there" semantics as a possible value for Unavailability message.
- Prevent outages from badly constructed schedules: schedules with > x% overlap in resources are rejected.
- Prevent outages from badly performed maintenance: if x% resources not being re-offered, cancel schedule.