Whole Machine Draining

Overview

Add basic mechanisms for draining whole startds.

Once draining begins, the Start expression is forced to be False. No new jobs may start.

While a slot is draining, this fact is advertised via `Draining=true` in the slot ad. We do not introduce "Draining" as a state, because most of the functions of the existing states and activities are still required during draining.

Once a slot is in the Owner or Unclaimed state, it transitions to the "Drained" state and the "Retiring" activity. Once all requested slots are in the Drained state, the slot transitions from Drained/Retiring to Drained/Idle. Now the request is ready to be finalized. We could provide several options for what to do at that point. The desired option shall be specified in the request. For example, all drained child slots could be merged into the parent partitionable slot and normal operation could resume. Or the startd could be stopped or restarted. Or the slot could just sit in Drained/Idle status until the requester intervenes.

Two draining schedules will be supported:

- fast – do not honor retirement time promises; immediately evict jobs
- graceful – honor retirement time promises, then evict

The startd will publish the following information for both fast and graceful draining scenerios:

- How much badput will result if draining is initiated now.
- When draining will be finished.

The startd will publish statistics on how much badput and idle time has resulted from draining in the past.

The information published by the startd will go in all slot ads. In the future, if a startd ad is added, this information could go there.

Estimating time to completion of draining

Suspension is problematic when making an estimate of when graceful draining will complete. MaxJobRetirementTime specifies the total *unsuspended* runtime the job can expect, so if any retirement time remains, and suspension happens, there could be a delay of arbitrary duration. The suspension policy is controlled by configurable boolean expressions that start and stop suspension. There is no general way for Condor to inspect these expressions and estimate how long the job will be paused or to understand the relationship, if any, between suspension of one slot and activity in other slots.

In version 1, we will assume no suspension when making the estimate of draining completion time. We will provide a policy mechanism for admins to choose between the following behaviors:

- treat suspension and retirement time as it is treated now (deadline estimate may be too short)
- evict suspended jobs when draining (deadline estimate is accurate)

Another issue in estimating time to completion is the unknown amount of time it will take to evict the job. The eviction policy is controlled by a configurable boolean KILL expression. There is no general way for the startd to inspect this expression and estimate how long the eviction could take. Furthermore, how long the eviction is *likely* to take depends on details of the job. We propose ignoring eviction time in version 1. We will assume eviction time is 0 when making the estimate.

Estimating badput

In the fast draining case, badput shall be computed as the sum of the runtime of all existing jobs. We will ignore details of checkpointing in this computation.

In the graceful draining case, we might like to know whether jobs are likely to finish within their allotted retirement time or not. For this, we would need an estimate of the runtime of the job. If this estimate comes from the user, we may wonder if the user would estimate a longer runtime in order to make draining of the node they are using seem expensive compared to other nodes. In version 1, we propose that graceful badput be computed with the assumption that existing jobs will run for the whole promised retirement time and will then be evicted.

Race Conditions

What if a draining request is made based on stale information about the state of the slots? This could result in a requested draining schedule that is different from what the requester might have chosen given up-to-date information. The draining request might cause badput or risk idle time that was not intended. The draining might take longer than was expected. The requester might even have chosen a different machine to drain, given up-to-date information. This whole issue is likely to get worse as machines get bigger and can offer more slots.

To partially address this, the draining request will be implemented as a transaction. The request will be sent, the startd will respond by giving up-to-date estimates to the requester, and then the requester will respond with “commit” or “cancel”. The startd will block operations that would change state during the transaction. In practice, this is most easily implemented by having the startd block all actions while it is waiting for the requester’s response.

Draining Commands

New startd commands control the draining of the startd. These commands require ADMINISTRATOR-level authorization.

StartDraining

- input: schedule (fast, graceful)
- return: (on success) a new draining request id
- return: (on failure) a descriptive failure reason + failure code

CancelDraining

- input: a draining request id
- returns success/failure and descriptive failure reason + failure code

Initially, a command-line tool will be developed to manually send these commands. The protocol should be implemented in the daemon client object, which can be reused in the future in a pool defragmentation daemon.

Only one draining request may exist in the startd at one time. If a new request is received when one is already in progress, the new request is rejected.

ClassAd Attributes Related to Draining

Startd Attributes

State (existing attribute)

- this is "Drained" when the slot is finished draining

Activity (existing attribute)

- while in the Drained state, the activity is "Retiring" while waiting for other slots to drain and "Idle" once all are drained

DrainingRequestId

- id of active draining request
- undefined if none

ExpectedMachineGracefulDrainingCompletion

- timestamp at which graceful draining should complete

ExpectedMachineFastDrainingCompletion

- timestamp at which fast draining should complete

ExpectedMachineGracefulDrainingBadput

- how much badput would be caused by graceful draining
- assume jobs run for full retirement time and are evicted

ExpectedMachineFastDrainingBadput

- how much badput would be caused by fast draining

TotalDrainingUnclaimedTime

- seconds spent Unclaimed while draining + time spent in Drained state
- does *not* include time spent in Owner state or Claimed/Idle

TotalDrainingBadputTime

- seconds of unsuspended job runtime spent on jobs that were evicted due to a draining deadline
- no special treatment of checkpointable jobs is made