## Step 1 of Pool Defragmentation Work – Draining Mechanics

### Overview

A draining request applies to a set of slots in a startd. 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.

A deadline must be specified for the draining request. How should the deadline interact with existing policies such as uninterrupted time promised to existing jobs and allowance during eviction for checkpointing? This question is addressed in more detail in the section on deadlines. In brief, we propose starting with deadline semantics that do not disturb existing policy mechanisms. We call this a “graceful” deadline.

While draining is in progress, we want the default policy to be to only start new jobs if they are likely to get a useful amount of work done within the deadline. User-supplied estimates will be used in this policy.

Once the draining deadline has passed and the slot is in the Owner or Unclaimed state, it transitions to the “Drained” state and the “Retiring” activity. The Start expression is forced to False. Changing conditions may cause the deadline to be shortened so this happens sooner than originally expected. See the discussion of draining deadlines.

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.

We wish to be able to estimate and monitor the cost of draining. The quantities we will consider are badput and idle time.

## Deadlines

Today, there are three different schedules for shutting down the startd: fast, graceful, and peaceful. We envision that multiple types of draining deadlines may also be desired. In this version, we will focus on only “graceful” deadlines, leaving open the option to add others.

A graceful deadline does not cut short time promised to jobs by the startd (MaxJobRetirementTime). Currently, promises made by the startd are for unsuspended time. The startd can advertise the earliest time at which its existing promises will complete, but since the amount of suspension time is not always predictable, the actual completion may be later. The startd will auto-extend graceful deadlines in order to meet promises. The amount of slippage from the original deadline shall be advertised. In the future, a firm deadline could be added that caps the amount of slippage by breaking promises. Rather than hacking that in now, we would like to consider that as a separate step. It may be desirable to have a first-class cap in the policy, advertised and negotiated like MaxJobRetirementTime.

During draining, the startd must automatically adjust its MaxJobRetirementTime expression so that it does not promise new jobs more time than what remains before the current deadline.

When jobs complete, it may be desirable to shorten the draining deadline to more closely fit the remaining jobs. In general, the policy for how to do this might need to take into account information the startd does not know, such as user priorities. Therefore, in the most general case, it is the requester’s responsibility to adjust the deadline. However, for convenience and efficiency, some simple policies could be delegated to the startd. In version 1, we propose the following choices of policy:

1. NEVER: Do not shorten the deadline.
2. ALL\_IDLE: If all the slots are idle, shorten the deadline to now.
3. GOODPUT\_TIME: Shorten as long as more badput is unlikely. (See RequestGoodputTime in draining section.)
4. RETIREMENT\_TIME: Shorten to just what was promised, no more.

Once the draining deadline has been reached, all running claims shall be evicted, following the startd’s usual procedure for graceful eviction. As usual, the time given to evict claims is terminated by the KILL expression becoming true. Unfortunately, this does not allow the draining requester to know in advance how long eviction could last. For now, we do not propose addressing this, but in the future it might be good to replace the boolean KILL expression with an expression that yields a maximum time⎯e.g. MaxJobEvictionTime. This would allow better planning and it would allow the eviction time to be negotiated.

## Backfilling

To accomplish the desired backfill policy, we rely on an estimate of the minimum amount of unsuspended time in which useful work is likely to get done for a given job on a given machine. *How* likely depends on the quality and intent of the estimate. That is up to the user. We do not propose here to create new tools to help improve upon the estimate, but this could be pursued in the future.

The estimate, in seconds, shall be given by the job attribute RequestGoodputTime. This is an expression that may refer to both job and machine attributes. For checkpointable jobs, rather than being an estimate of the total job runtime, this could be the amount of time needed to make the startup and checkpoint overhead worth it.

Note that the naming of this attribute is intended to match the naming scheme for other attributes such as RequestMemory and RequestCpus. Like other Request attributes, when this attribute is not explicitly referenced in the user’s requirements expression, a default requirement is inserted. The default only accepts machines with a sufficiently distant draining deadline (or no deadline).

*When this attribute is not defined, the default behavior shall be for the job requirements to ignore draining deadlines.* Like all job attributes, the condor administrator can configure a default value to be used in case the user does not define it.

Note that RequestGoodputTime is *not* asking for a promise. The job may still be evicted before RequestGoodputTime expires. It is just asking for a slot with no existing deadline that will cause badput, and it is providing information for future decisions that might cause badput. For a *promise*, the job can require a slot offering a minimum MaxJobRetirementTime. This capability already exists, but perhaps it could be made more convenient by adding an attribute RequestRetirementTime that behaves analogously to RequestGoodputTime.

## 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 deadline that is different from what the requestor might have chosen given up-to-date information. The draining request might cause badput or risk idle time that was not intended. The deadline might have to be lengthened to meet promises. 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 address this, we propose that the draining request specify an acceptable tolerance for total expected badput and immediate slippage of the deadline. This can be done via a “staleness tolerance expression” that is sent along with the request. If the expression is not true for all selected slots at the time when the startd instantiates the draining request, the request is rejected and the caller is informed. The startd will publish the deadline slippage and the expected total and slot badput in the slot ads that this expression sees.

A different type of race happens in the case where a job is matched to a slot based on stale information about the draining state of the slot. A job could get matched to a slot that has a draining deadline that is sooner than is acceptable to the job. However, since the job and machine requirements are evaluated by the startd immediately before activating the claim, this race is automatically resolved by rejecting the job. The slot may sit idle until the next negotiation cycle. We do not propose any further refinement of this case.

## Monitoring Cost

To aid in the estimation of costs of new or pending draining requests, each slot shall publish the timestamp at which RequestGoodputTime will expire for its existing job. If RequestGoodputTime is undefined, this attribute too shall be undefined. The slot shall also advertise the time at which MaxJobRetirementTime will expire for its existing job. Both of these timestamps need to be updated when suspension of the job has happened.

To keep track of the past cost of draining, the startd shall publish information for the sum over all draining requests in the startd’s life. In addition to these lifetime statistics, other statistics will be reported, consistent with the proposed treatment for augmenting other sums in the startd. These statistics shall not be tracked per slot but rather per startd, because with partitionable slots, the utility of per-slot statistics seems questionable. The amount of idle time, the slippage of draining deadlines, and the badput shall be reported.

How shall badput be computed? The question is what to do about jobs that are evicted after their RequestGoodputTime has passed. When a job last successfully checkpointed is also relevant but may not be known for all types of jobs.

One badput metric would be to sum the time spent by evicted jobs that ran for less than their RequestGoodputTime. This could be thought of as the pool administrator’s view of badput: “You said you would be happy to get one hour, I gave you one hour, so you should be happy, and I’m not going to penalize myself for the unlucky fact that you aren’t.” Another way of looking at this metric is that it is a relevant system metric when it is the prerogative of the *user* to decide how much risk to take in order to attempt to fit into small time windows, and it is the prerogative of the *system* to manage the breakage of those windows. It is tempting not to complicate this metric with details of checkpointing. If checkpointable jobs have a small RequestGoodputTime, they will tend not to contribute much to this metric.

A different badput metric would be to sum the time spent by evicted jobs after their last successful checkpoint. Whether jobs ran for less or more than RequestGoodputTime is not reflected in this metric. One problem is that users may not bother to instrument their jobs or configure extra job attributes to report checkpoint timings. Standard universe could be handled automatically. Perhaps once vanilla checkpointing is more mature, additional cases could be handled automatically. For now, we propose a simplified version of this badput metric in which jobs identified as checkpointable are simply excluded from the sum. A new boolean job attribute will advertise whether the job is checkpointable. For standard universe jobs, this will default to true.

### Draining Commands

Three new startd commands control the draining of the startd. These commands require ADMINISTRATOR-level authorization. A more specific authorization level would be nice, but seems overkill for version 1.

StartDraining

* input: an expression that matches all slots to be drained
* input: deadline (timestamp)
* input: deadline type: (graceful)
* input: deadline shortening policy: (NEVER, ALL\_IDLE, GOODPUT\_TIME, RETIREMENT\_TIME)
* input: (optional) staleness tolerance expression that must be true for all selected slots or request is rejected
* input: (optional) one or more DrainingAttr<X> attributes to put in slot ad
* return: (on success) a new draining request id
* return: (on failure) a descriptive failure reason + failure code

CancelDraining

* input is a draining request id or an expression to match against slots
* if an expression was specified, draining is cancelled on all matching slots
* otherwise, draining is canceled for all slots affected by specified id
* returns success/failure and descriptive failure reason + failure code

AdjustDraining

* input: a draining request id
* input: (optional) a new deadline
* input: (optional) a new deadline type
* input: (optional) a new deadline reduction policy

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.

In the future, the startd may support multiple pending draining requests, but for now, we do not propose to support this. If there is already a draining request on any of the selected slots, any new request will be rejected with the response “too many draining requests.”

### ClassAd Attributes Related to Draining

#### Job Attributes

RequestGoodputTime

* unsuspended seconds from start of job after which it is likely to have gotten useful work done
* how likely is not enforced by Condor. Imagine 90%.
* used for backfilling: by default, jobs require slots with a draining deadline that comes after RequestGoodputTime

JobCheckpointable

* true if job should be treated as checkpointable
* job is not required to tell the truth; this is intended to be a policy and accounting toggle, not a toggle that controls actual job behavior
* defaults to true for standard universe

#### 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 on this slot
* undefined if none

DrainingAttr<X>

* attribute given by draining requester

DrainingDeadlineTimestamp

* timestamp at which this slot will begin evicting jobs
* undefined if none
* computed as minimum of all deadlines in DrainingDeadlines

JobGoodputTimestamp

* timestamp at which current running job is likely to have gotten useful work done
* computed as JobStart + RequestGoodputTime + TotalJobSuspendTime
* undefined if job does not define RequestGoodputTime

JobRetirementTimestamp

* timestamp at which promised time will have elapsed
* undefined if no deadline
* computed as JobStart + TotalJobSuspendTime min(MY.MaxJobRetirementTime,TARGET,MaxJobRetirementTime)

JobCheckpointable

* true if current job should be treated as checkpointable
* computed at job start by evaluating JobCheckpointable in job ClassAd

TotalDrainingUnclaimedTime

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

TotalDrainingBadput1Time

* seconds of unsuspended job runtime spent on jobs that were evicted due to a draining deadline that came before JobGoodputTimestamp

TotalDrainingBadput2Time

* seconds of unsuspended job runtime spent on non-checkpointable jobs that were evicted due to a draining deadline

TotalDrainingDeadlineSlippage

* sum of difference between actual time when Drained state was entered and the time that was originally requested
* note that some cases may be negative

ExpectedTotalDrainingBadputTime

* When evaluating the “staleness tolerance expression” this attribute is set to the total badput expected from the requested draining deadline

ExpectedSlotDrainingBadputTime

* When evaluating the “staleness tolerance expression” this attribute is set to this slot’s badput expected from the requested draining deadline

ExpectedDrainingDeadlineSlippage

* When evaluating the “staleness tolerance expression” this attribute is set to the initial amount of deadline slippage, given the deadline shortening policy and adjustments for time promised to existing jobs.
* Could be positive (longer deadline) or negative (shorter).

## Development Plan for Step 1, Draining Mechanics

* [ 3 days ] – Add new draining commands to startd and create command-line client to send them.
* [ 3 days ] – Add startd “Drained” state. Need to adjust state-dependent policies in startd and elsewhere to account for new state. Condor view and condor\_status also need to be adjusted.
* [ 2 days ] – Add support for Drained state 🡪 defragment partitionable slots and resume normal operation.
* [ 1 day ] – Add new job attributes.
* [ 2 days ] – Add new startd attributes.
* [ 2 days ] – System integration.
* [ 2 days ] – Update documentation.
* [ 3 days ] – Write automatic tests of draining:
  + jobs with no time estimate behave correctly
  + jobs with time estimate that is too long do not start during draining
  + jobs with short enough time estimate do start during draining
  + claims still running at draining deadline are retired
  + claims still running at canceled draining deadline are not retired