Doodle 63
Horde Initiation and Management

Within a distributed cluster in Crux, a horde is a secured grouping of nodes within a flock. Hordes have their own lead node which runs healthcheck services, aggregating the health status information of horde members.

This document describes recent work on crux bootstrapping and solutions to some global cluster bootstrapping issues, including how we assign nodes to hordes as the cluster grows, changes in horde assignments over time, monitoring global cluster health, and starting/stopping computations based on node membership numbers.

To begin, it is intentional that - on the automated startup of a crux cluster, there is no information provided about how many nodes will be joining. It is, at any given time, the size of the dynamically accumulated flock, which may be some subset of the bloc of fulcrum running nodes that are initiated.

When planning for node roles, we use crux "hordes", which are the organizational groupings within the cluster. The assignment of a node to a horde presents a problem of "best filling" for each horde, for each possible value of N, so the desired cluster is completely scalable.

Previously we used the command line argument -horde to specify that the given node will join a specific horde. To make this instead, a self-organizing and scalable system, we can deploy a horde management service on the flocking leader to suggest the horde for a node to join. This service needs an input model of the cluster listing which hordes are to be populated and in what proportions. The model should suffice to fill in a minimal "test" sized cluster, provide maximums for components that are not expected to scale, and proportions of nodes so that horde assignments can be scaled to any value of N.

The horde growth model is specified on the command line as the -hordes argument, for every node running fulcrum and booting into the cluster. Any modification or changes to this model need to be distributed amongst the horde members in the event the leader falls over, and one of them has to assume the cluster leadership role.

The sharing of the horde growth model also provides each node with data that specifies what a minimum operating cluster looks like at that point in time.

We can take advantage of this to provide a means for a node to query the cluster and horde composition, and compare it with the model. This would also be helpful for an individual node to detect a network split event as the model horde/node growth state can be compared with the

current cluster population. As the horde model provides a minimum population of nodes in hordes, each node can assess whether that minimum threshold is met, and thus start or stop computing tasks as required.

So then we want a system of dynamic horde management that has a scalable model of cluster growth. Node to horde membership assignment should follow best efforts to fill this scalable model.

Bootstrap Sequence

The appropriate place on the cluster bootstrap sequence for assigning a horde to a node is right before the 2-factor, 2-channel registration (/pkg/register).
Alternatively a command line argument can still suggest a horde name for a node to join - by treating the argument as a "pinned" horde name, but we recognize that this may be overridden by the horde management service if it is not appropriate.

Horde Management Features

Each fulcrum boots with two related arguments that specify
(a) a pinned horde name on the command line (the -horde argument)
(b) a model of the hordes to be populated and how they should be filled with nodes as a function of N. (the -hordes argument)

The horde management and tracking service runs next to the Registry and needs to:
(c) parse and expand the -hordes argument from (b) into a tracking structure
(d) dispense horde name hints to cluster nodes that are booting up using (c)
(e) on Registration, add the node to the horde tracking structure
(f) provide for an unassigned node pool - the "NoHorde" horde
(g) provide for allocating a new horde from (c)
(h) provide for freeing up nodes back to (c)
(i) a way of notifying a node that it's horde assignment is changed
(j) a way for a node to stop its current horde services and reset/re-register in the new horde
(k) a way of distributing the current global horde model to each node

Beyond these, we would also like some additional features:
(l) a reserved horde name "CruxHalt" that shuts down each fulcrum - to shut down the entire cluster
(m) a way for each node to know if the entire cluster is populated at a viable minimum number of nodes in each horde for the cluster to be "operational". In the case of a flock split, this provides each side of the split with some basic information to know whether the entire

distributed system should continue computing (i.e. has minimal membership of each horde satisfied?).
(n) better ways of providing/computing service rules between horde members than the present compile-time table.

Bootstrapping - Fulcrum Command Line

For each node's crux startup (fulcrum), there are two crux arguments for initializing the horde structure:

-horde Originally specified the horde name. Modified to be a "pinned" or "desired" hordename, but now subject to control and override by the horde management service.
e.g. "compute"

-hordes The "growable" horde structure model as a function of N.
Specifies each horde names, (not including the Admin horde),
the minimum number of nodes required,
the upper limit of nodes (if needed),
and the ratio of nodes to add as N grows.
e.g. "control:1,1,0:database:1,3,0:storage:1,0,1:compute:3,0,3"

Unassigned Node Pool

The unassigned horde node pool "NoHorde" is provided for nodes that are not assigned to any other particular horde or specific use, so that these nodes can participate in the cluster, receive updates, and be alerted for assignments to a working horde. When the -hordes argument is passed with all 0 values for the ratio (third number after the horde name), the system automatically grows using "NoHorde" as the horde name. There is no upper limit imposed on this automatic "NoHorde", but it can be specified - e.g. "NoHorde,0,5,1" would cap the pool to 5 nodes on boot. In this case (TODO check this expected behavior) additional nodes would be able to join the flocking protocol (which has no upper limit) but be prevented from registering for grpc-signature key distribution. This would be a kind of cluster "purgatory" state, from which the only escape would be the removal of another node.

Flocking Leader Horde

Flocking leader nodes get promoted to the "Admin" horde, and their -horde argument is ignored. At present the horde management model and service do not operate on the "Admin" horde.

Horde Reassignment

Horde allocation/allocation and freeing/shrinking/removal of nodes proceeds by horde reassignment operations. New horde assignments in a running cluster come from the "NoHorde" pool, and are returned there after they are unassigned. Without the -horde or -hordes arguments, all nodes join the "NoHorde" pool. Horde reassignment operations are cluster wide operations that are locked so as not to proceed concurrently, and are eventually consistent. An API "busy" indicator is provided to indicate when hordes are being reassigned.

The Horde tracking model

The flocking leader initializes the crux Registry and Steward services. The Registry service initializes the horde tracking model by parsing the -hordes argument and building a Markov-chain like structure that initially covers the minimum and maximum number of nodes for each horde specified. As the cluster grows, the repeating unit (derived from the node ratio) is added to this chain so that the proportions specified are reasonably managed as N grows.

A Registry service HordeHint() API call provides a lightweight service to suggest horde names to nodes before they register for grpc-signatures key distribution with the Registry service. If that suggestion is not fulfilled, the "hint" times out and is given to another node. HordeHint() is secured with TLS and the shared registry key, as are the other Registry services. As nodes request HordeHint(), they are marked and timestamped in the data structure. As nodes successfully carry out the 2-factor -2 channel Registry handshake, the NodeID and NetID of their reeve is added to the tracking data structure.

Horde Manager API - Son of Ghengis

The Kublai service (son of Ghengis) is a plug-in that runs on the same node as the Registry service, and provides the horde management API and operations with grpc-signature security. Given that there is an Information API which should be internally open to user services, and a Management API which contains calls that should be restricted to administrators.
So this is split into 2 separate grpc-signatures secured services - Ghengis and Kublai.

Kublai Information API

HordeList() - provides a simple list of (non-Admin) horde names in the cluster, and the "busy" status when the cluster is working on Ghengis operations.
MyHorde() and HordeInfo() - provide a summary of a horde including:

- the horde name
- the owner of the horde

- number of nodes requested when allocated; 0 is unspecified
- timestamp when horde was allocated
- timestamp of the summary report
- NodeID of the leader of the horde (the horde's HealthCheck service)
- NetID of the HealthCheck service on the leader
- current list of nodes in the horde
- 'busy' indicator - true when an active HordeAlloc()/HordeFree() operation is running

HordesModel() - extracts and returns the present -hordes formatted string - which changes as Ghengis processes HordeAlloc/HordeFree and alters the horde tracking data structure. Any node that becomes a leader will need this updated information, so this should be distributed to all nodes after the cluster makes changes to the hordes. This may wind up being an API call in Reeve, using a push model like Steward.

Ghengis Horde Management API:

HordeAvail() - total present number of nodes, total present number in "NoHorde" available pool, busy status

HordeAlloc() - move n nodes from "NoHorde" to specified horde - locks out other HordeAlloc/HordeFree (busy=true)

HordeFree() - move n nodes from specified horde to "NoHorde" - locks out other HordeAlloc/HordeFree (busy=true)

CruxShutdown() - move all nodes from current horde to "CruxHalt" - which is a reserved word that triggers a workflow stop and a clean exit from each fulcrum.

Local Horde Persistent Storage and Fulcrum Restart.

In fulcrum, the muck system (pkg/muck) holds local node storage. Data for services like Reeve, Register, Steward, and grpc-signature keys are stored here so that a node can checkpoint and restart back to its last state for supporting grpc-signatures protected services. In order to accomodate a fulcrum's ability to switch hordes, the local persistent storage, ".muck" has been changed to a longer pathname prefixed by the bloc name / horde name. In the case of a fulcrum restart, the LastMuckPath() function returns the last active bloc name and horde name. As a node changes hordes, it will end up reversing the bootstrap workflow and that may or may not remove data relevant to the horde. In some cases I would argue leaving the data in place would be helpful for debugging cluster state after failure, especially for the Steward database. It will be necessary to audit the leftover information to ensure that, for example, private keys are being deleted upon horde changes.

Additional Points/Features/Things to Consider.

1. The -hordes command line argument establishes the horde model to bootstrap into.
As hordes reorganize with HordeAlloc() HordeFree() the data structure changes in Ghengis. It can be interrogated with HordeList() and calls to HordeInfo() for each horde.

2. HordeAlloc() needs to provide a Min/Max/Ratio to allow reconstruction of the -hordes string, so it can be pushed around to each node in case of leader turnover, and the establishment of a new Registry/Steward service, as are the horde rules currently managed by Steward().

3. Nodes that are dropped on leader failure can re-register with their last horde assignment as the pinned -horde argument.

4. Any node's Reeve could can poll HordeList()/HordeInfo(), compare the node counts to the minimums specified in the -hordes argument, and determine whether the minimum number of nodes is satisfied.

5. The combination of the distributed "-hordes" model - the target cluster and minimally operational custer, and the actual numbers in HordeList()/HordeInfo() queries allows each node to interpret whether or not the cluster is in a minimally operational state. This may be distilled down to a boolean go/no-go in the bootstrapping and operational workflow engine.

6. The horde "rules" structure may be populated from a combination of the -hordes model data structure and the services registered within the Steward database, rather than be compiled-in as currently done.

7. Futz testing with "Admin" and "CruxHalt" in the -hordes argument string to ensure these are rejected. Test limit on "NoHorde" when command line specified - understand the "purgatory" situation.

8. Horde Reassignment operation ongoing when a leader dies. New leader has no memory of the last HordeAlloc() HordeFree() operation. There is no audit trail of operations in memory, should be logged yes. On failure of HordeAlloc() to reach its value of n, do we return some partial set m < n nodes or 0 nodes? Here the Min/Max/Ratio would help.

9. API should provide a way of seeing the repeating unit, in the event new nodes are added later, so their allocations can be understood. - This is now HordeModel()