Interacting with low-level flocking

Flocking is a low-level activity that runs continuously underneath a horde. The question is how do we talk to this activity? Or rather, what do we need to know about the current state of flocking? Keep in mind that we may need to debug failures to flock.

What can we do right now?

There are currently two classes of interactions: user-initiated and logging.

User-initiated probes

You can ask your Confab interface a few questions:

1. GetNames() (cluster, leader string, stable bool, node string) returns
    a. cluster is the flock's idea of the cluster name (machine generated; transient)
    b. leader is the leader's node name
    c. stable is true if the flocking code thinks the leader is stable (it may take a while)
    d. node is the node's name (machine generated; transient)
2. SetRegistry(ip string, port int, key string) sets the registry address
3. GetRegistry() (string, string, error) returns the registry address
4. SetSteward(ip string, port int) sets the steward address
5. GetSteward() (string, error) returns the steward address

Logging

We log MonInfo structures into their own capture stream (beacon):

```
type MonInfo struct {
      Op      MonOp
      Moniker string
      T       time.Time
      Flock   string
      Oflock  string
      N       int
}
```

where the MonOps are one of JoinOp, LeaderStartOp, LeaderHeartOp, LeaderDeltaOp, ProbeOp, RebootOp.

This lets us trace the rise and fall of flocks.

Weaknesses

There seem to be a few problems with this:

1. there is no way to find out who is in the cluster.

2. you have to be in the cluster to find out about the cluster.

What could we do?

Fundamentally, flocking works by passing around an Info struct:

```
type Info struct {
	Flock       string // name of my flock
	Dest        NodeID
	Leader      NodeID
	Lvote       int32
	Vote        int32
	Me          NodeID
	Expire      time.Time
	Prim0       Key
	Sec0        Key
	Beacon      string
	Steward     string
	Registry    string
	RegistryKey string
}
```

If Lvote is zero, then these packets are interpreted as special purpose commands:

1. CmdReboot (reboot the flocking after waiting a randomised interval)
2. CmdCandidate (send our Info to a specified node)

These are used primarily for testing. But otherwise, members ping their leaders periodically, and on a different period, leaders ping their members.

Forming clusters is done by comparing Info's that come in with our (node's) notion of cluster information and doing the right thing. Leader's do a few more things:

1. maintain a list of members with expiry times.
2. capture a list of members.

Low impact stuff

There is a bunch of stuff you can do with very little impact:

1. you can run the same Confab functions described above: GetNames(), SetRegistry(), GetRegistry(), SetSteward(), GetSteward(). This can be run independent of flocking per se.
2. GetNames() gets you the leader and stability, which means you could go to the leader and get their member list. (Except for now, that list does not go to disk, but it easily could.)

Stuff we actually need

When flocking works, it works quietly. Occasionally, though, you might need to figure out what is going on, should it fail (usually, through some weird split network anomaly). So the scheme we might propose is Prometheus-like:

- all the flocking debugging goes into a (per process) ring buffer
- you can ask the flocking protocol to send the contents of the ring buffer to a specified ip/port address. This transmission will take pace asynchronously.

On the other hand, the modern (Google?) style is dump a lot of debugging (because who knows when you'll need it?) and just look at it when you need to.

It is worth pointing out that the MonInfo structures are relatively few and far between; they only highlight changes in the flocking structure. So there is normally a flurry at the start of things, but then it calms down.

The proposed answer

The main issue is not the volume of outputs, but a temporal sequence: the beacon needs to be ready before nodes start sending to it. (The worry is whether nodes in a cluster can boot, cluster, act and exit ***before*** the beacon comes up!) So here is the proposed solution.

We typically run cluster tests with a tool like myriad or on AWS-like clusters. In these cases, the nodes are running with locally known IP addresses that are not visible outside their environment. So how can we get any information from the nodes back to the person running the test? The answer we used in Stix was to run a beacon in our IP address space. The beacon is a read-only dump of certain information from the cluster. The nodes are passed the beacon's IP address (and security keys), and they use that to send the information out. The assumption here is that the beacon's IP address is usable by the nodes directly.

The precise details are:

1. the nodes talk to the beacon via the low-level symmetric encoding used by flocking.
2. when the beacon starts, it forks. The background fork starts the monitoring of input at an IP/port of its own devise and relays that IP/port to the foreground fork, which then prints the IP/port and then exits.
3. this means the beacon is up and running before we actually start firing up the nodes.
4. the symmetric key information is already passed to each node (as part of a node "booting" up), so we can just use that.

The end result is something like

```
addr=`beacon -key $key`
# now we can use fulcrum -beacon $addr -key $key in a crux.mf or somesuch
```

And as a final note, we can send a command to the beacon to finish up neatly.