what's up with blocs?

Prologue

When we talk about networking within and around crux, we really mean three overlapping views:

The first view is simply that we have symmetric (read/write) connectivity between all the nodes in the cluster. All such connectivity is encrypted. In addition, we have external nodes which we access by a normal IP/port address. In these cases, link-level security (such as encryption) is the application's responsibility. The beacon is an example of an external node; in this case, we supply the encryption key on the command line (a clumsy way to implement a single entry vault).

The second view is the flocking view. Here, every node with the same bloc id (sourced from a certificate) participates in building flocks. Networking anomalies may cause multiple flocks within a bloc; we need to be able to talk about these flocks (even if only for diagnostic purposes).

The third view is that of the PKI whitelisting infrastructure supported by ***reeve***/***steward***. This could easily span multiple blocs, particularly if nodes are defined by a bloc/node identifier tuple. Currently, steward runs on the leader node of a flock, but this could easily be changed to a single central point.

In these views, the primary identifiers (bloc, node id) are sourced from certificates, which are preloaded onto the nodes before they boot. Both ***myriad*** and AWS support this notion of preloaded certificates.

Overview

A *bloc* is a group of computing entities (we sometimes call these *nodes*) with a common *bloc_name* (string). It is presumed these are allocated as part of node initialization by processes not under consideration here.

The nodes in a bloc continuously attempt to form a single *flock*. All the nodes in a flock must be able to talk to each other; when this is disrupted, the nodes will form into multiple flocks, each with its own leader and flock name. The flock name is a transient label, kept in the flocking label.

The nodes in a flock may be divided up into *hordes*. Each horde has a human-settable name. For now, hordes simply share their horde name, but in the future, we anticipate that there will be resource allocations involved. Fundamentally, hordes are an administrative thing. We intend to supply horde management software, handling creation and deletion of hordes, and augmenting the nodes and other resources in a horde.

Proposal 1

The function confab.GetNames() will additionally return a bloc name. This will come from the flock's Info structure.

Problem 1: split flocks

Let's review how flocking works. We start off by every node being its own cluster (and its own leader). Each flock continuously searches for new nodes to join its flock (both nodes it has seen before and brand new nodes). Each time it sees a node, the flocks attempt to combine together. Depending on density, this normally means that for a short time at the beginning (after, say, a powerup situation or the like), there is a flurry of activity while nodes change flocks. The things to remember are

- this activity occurs all the time (and is thus resilient to nodes going up and down)
- it is plausible that occasionally (say after a switch breaks) some nodes will no longer see some other nodes, and will naturally form a new flock(s)

What happens when a flock splits and rejoins? The flocking protocol is designed for this, but it affects cluster-wide functions, such as *reeve*/*steward* and *proctor*. Proctor solves the problem of starting up the correct number of processes, but does not help with the issue of how to combine two existing processes (when two flocks combine back together). This issue applies at multiple levels: on a flock level, at a horde level and at an application level. At the horde level, one could imagine the horde might be set up as a group of *n* nodes. If a node goes down for an extended time, one could imagine retiring that node and replacing it with a new node (from the existing flock). You would then have the problem of integrating the new node into the horde (independent of flock-level issues). Within a horde, an application might need similar advice that the horde has either parted or rejoined.

question 1: how do you handle flock splits?

For steward and reeve, the answer is straightforward: when a reeve detects that the IP address for the steward (returned by Confab.GetNames()) changes, it simply marks all its entries as "to go" and sends them all to the (presumably new) steward. This needs a mechanism to retire old entries from steward. Note that this does not require that the steward runs on the leader node.

For other services, it depends on how they were started and managed. We anticipate that infrastructure services are managed by *proctor*, which will start up new services when a horde changes its membership. We need to ensure the proctor driver data propagates to all the nodes.

question 2: how do you handle flock rejoins?

For steward and reeve, the above answer holds; just resend the local information as it is strictly additive. We need to change steward to realise that when it sees the steward address is no longer itself, then it needs to shut down (after a little while).

For other services managed by proctor, proctor will deal with the situation (that is what it does!).

Proposal 2

We need a horde-wide liveness service, with transparent rules for liveness/deadness. (This may easily be the *healthminder* service.) This assumes we equate liveness to the presence of a heartbeat. We probably ought to link steward's retirement rules (for old services) to presence in the liveness service.

We anticipate that any horde- or cluster- or bloc-aware "wellness functionality" would depend heavily on this liveness service.

Problem 2: forward secrecy

How does this fit in with forward secrecy? Especially, how do we add new nodes and detect and recover from network splits? (Most of this derives from details in doodle60.)

Each computing entity will have its own certificate containing bloc and node name issued by the CA. New nodes are therefore not a problem; they will run a Diffie-Hellman exchange to get new (transient) session keys and communicate with the bloc members. The initial certificate chain will be stored on the node as part of bootstrap. Getting new cert's before the current ones expire is seen as an admin task. A bloc will run with an accessible *certificate authority* (CA); we will ship a small example. We intend to avoid revocation lists.

The leader will be able to tell when nodes leave (because they will time out). The nodes that have left will determine their new leader(s). Most likely, this will happen with the current keys, but if not, they will get them via Diffie-Hellman.

When the split heals, we rely on nodes probing the network to encounter nodes from other parts of the split. When they do, the nodes will do the Diffie-Hellman exchange and gain the keys they need.

Notes

Feb 13 Synch

Doodle42xx - Chronicles of Radix, Basic terms defined here. Including fulcrum. Probably needs rework since thinking has shifted according to Andrew.

Node = container, vm, or bare metal

Could run multiple fulcrums (say in k8's) but they'd be using the same resources.

- Skaar doesn't think we should bind fulcrum to node names.
- Loren /Skaar think there could be multiple fulcrums per node
    - Was mentioned in earlier doodle 42xx
- Skaar: fulcrums not tied to nodes.
- Do we have Steward per fulcrum or per flock?
    - One steward per confab

Liveness service
Servers heartbeat to the liveness service. No guarantees that service can make progress since heartbeating likely in a separate thread.

Andrew:
Proctor handles what _should_ be.
Liveness service tells you what _is_. (healthminder?)

Which direction should liveness info be initiated from for central liveness record?   Healthcheck style "pings" or heartbeats from servers.
Skaar: Maybe both.   Could even have components checking with each other (for required communication patterns).

Andrew: Liveness server could use chris's Ping service, and then heartbeats for other direction.

Forward Secrecy
Triggers node hierarchy discussion.

Tom: info gets loaded in from certificate when container started.

Skaar: info embedded in keys didn't hold up well in his experience. DC (Domain Component) naming allows you to arbitrarily grow path prefixes. X509 certs allow.

Bloc structure

There are at least a few bloc-wide services, including

- steward
- horde administration
- certificate authority
- muster

and any other services that operate on an inter-horde basis, such as secrets shared between hordes.

The proposal is to have a horde that supports these services; it would have some conventional name (such as **crux**). There would be a simple access service, with a couple of methods:

- add a service
- list all services

This horde would have all the other normal services, including liveness. A service would "time out" when it times out in the liveness monitor.

The next section describes how to bootstrap this thing.

Bootstrapping

The question arises: how do you boot up a bloc? Let's assume we call the administrative horde **ADMIN**. Then the bootstrap pseudocode looks like:

```
start_flocking()
for {
      read flock.horde
      if (flock.horde == "") && flock.stable && (flock.admin == "") {
            // nothing set and we are stable; become the new ADMIN horde
            flock.horde = "+ADMIN"
            continue
      }
      if flock.horde[0] == "+" {
            // start a new horde
            flock.horde = flock.horde[1:]
            start_node_services("Dean", "Reeve", "Heartbeat", "Pastiche",
"Picket")
            if flock.horde == "ADMIN" {
                  // new ADMIN horde
                  dean.add("Proctor")
                  Proctor("Steward", "HealthCheck", "Genghis", "Yurt", "CA")
                  create_a_single_node_cluster("ADMIN", mynode)
                  // vestibule will set admin field
            } else {
                  // new regular horde
                  members := genghis.ClientNodes(flock.horde)
                  if mynode == members[0] {
                        dean.add("Proctor")
                        Proctor("HealthCheck")
                  }
            }
      }
      // hang out for a bit
      sleep(1*time.Second)
}
```

Notes:

- We assume that reeve and other like programs will continue trying to contact their destination services even that fails initially.
- yurt is a simple gateway function that lists a small set of services.
- we assume we can stash a single variable (horde) inside the flocking service
- we assume that the admin comes through the flocking protocol as "+name"

yurt

Yurt is the gateway entrance to the **ADMIN** cluster. It is purely passive, in that it simply shows what services are heartbeating in its horde.