The Fulcrum

Overview

How does work get done in Crux? The answer is complicated in detail, but relatively simple over all. The answer has two layers:

1. at the bottom is a clustering layer, which we normally call *flocking*. This layer gives you two sets of capabilities. One set relates to the clustering protocol; we use *flocking* which gives you
    a. secure membership
    b. leader
    c. unique member and cluster names
    d. a mechanism to pass service registration related info

    The other set of capabilities describes how the cluster can actually get things done. This is basically a set of services that nodes in the cluster can use to load and execute programs. These services include:
    1. ability to move data between nodes
    2. ability to execute Go functions from local files
    3. look up service access information across the whole cluster. This is related to but different from the registration service stuff in flocking.
    4. ability to alter the services running on a node

2. at the top are *hordes*, which are subsets of the underlying flock. The horde is what a user will normally use. Its functionality is open-ended, but we will supply services like
    1. admission control; which nodes belong to this horde?
    2. service registration data, including cluster membership. This service may be the same as the flocking scheme, or a different one.

The rest of this document describes these layers in more detail, and in particular, what information is supplied by the user, and how persistent storage is handled.

Underlying clustering

The low level clustering occurs amongst *computational entities*, or *nodes*. We use the term computational node because we want to include raw servers, virtual machines and containers. Fundamentally, all we need for a node is the ability to run some code (the *fulcrum*) there, a network address for the node, and an initial secret.

The fulcrum

The fulcrum implements the low level control for Crux. It runs a bunch of loosely connected services. The services are arranged such that every component, including the fulcrum itself, can be replaced. All services in Crux are accessed as gRPC connections on TCP/IP sockets, and as such, need point-point public/private passwords managed by the *strew* service. The services needed by a new fulcrum are

- the fulcrum itself. The available functions are
    - execute a new service
    - ask a service to exit
    - reset back to an initial state (as though we were just initialised)
- pastiche (data mover). The available functions are described elsewhere.
- flocking. This allows this node to talk to other nodes. Currently, flocking does not support a gRPC service (though it could).
- strew (password access to services). The gRPC interface is currently up in the air (but for now, we're tracking Chris's proposal).

Currently, the fulcrum needs a couple of parameters, namely a human-usable name for the node, and the IP/port address for the inter-node transport.

There will normally be other services as well, installed as needed. Generally, the pattern is to move the file containing the code to the node, and then execute the service as a Go plugin (from that file).

Services

The issue of identifying and designating services is surprisingly complicated:

- there is no direct relationship between the service defined in a config and the services offered by that code. A single module might offer multiple services, and the module name might have little to do with the names of the services supported by that module
- there is no mechanism to stop a service started by the fulcrum. That would need to be supported by the service itself.
- the intent is that services generally advertise unique names. For example, a module might offer the **`ntp2`** service. A newer module is added, supporting **`ntp3`**. You don't want either module to offer a more generic name, such as **`ntp`**, because they will clash.
- this means that the mapping from a generic request (say, **`ntp`**) to a specific service (say, **`ntp3`**) is done by the client (or maybe some middleware), but not by the service itself.

Each node has a service config. This is a list of tuples:

```
type Service struct {
      Type        int         // type of service
      Name        string      // name of the service
      Funcname    string      // entry point of the executable code
      Image       string      // SHA-3 of the file containing the Funcname
}
```

The Type field refers to how these services fit together within the fulcrum. That is, how information flows between the various services; see the code for details.

Services are Go functions invoked in one of a few different ways:

```
function svc1(quit channel bool, pub chan []Service, hbperiod
time.Duration)
function svc2(quit channel bool, pub chan []Service, rcv chan []Service,
hbperiod time.Duration)
```

Most services will be of type *svc1*; they heartbeat, with periodicity of ***hbperiod***, their registrations via the ***pub*** parameter. The *svc2* variant receives all registrations received by the fulcrum both from internal or external sources. Typically, only the networking service will be of type *svc2*.

Please note that services are intentionally loosely bound to the fulcrum. This may change over time.

There are some cluster-wide services needed at this lower level; they are

- configurator

Something else should go here.

Flocking

The flocking service provides inter-node communication; without it (or something like it), our clusters will be a single node. The flocking service is self-contained except for three parameters:

1. a human-readable name (optional) used in diagnostics
2. an IP address/port for the socket used for communication
3. an initial secret

It is intended that these parameters be passed into the node running the fulcrum, probably as command-line arguments and/or the environment.

Configurator

The configurator service is a simple way to handle staged changes to a fulcrum. The functionality it needs to support is

- take a list of current services and return a list of new services. The lists are compared, and services to be dropped have a true sent on their **quit** channel, and new services are started.

Passwords and GRPC access

We have a problem with how to start up and maintain Crux microservers. The issue is a combination of two disjoint problems:

- every pair of customer-server processes needs to share a public/private password pair ***BEFORE*** the service is used
- we'd like to use the resulting servers in several ways:
    - stand-alone for testing
    - in a small testing setup for small scale testing
    - within a cluster constructed through flocking
- we'd like the permissions to time out after a "while" (say an hour)

In the current scheme you basically supply a name, called **USER**, and you get back

- the USER name, needed by both customer and server
- a fingerprint, needed by both customer and server (ssh-keygen)
- private keys (in a file), needed by the customer (via ssh-add, ssh-agent)
- public keys (in a file), needed by the server (via a custom bolt.db database)
  See pubkeydb

The question is, how do you do this? This seems straightforward, but with a twist:

- pick a USER name that is basically (customer-node-name, server-node-name, service)
- ship the public keys to the server node
- hide the private keys locally on the customer node
- data is sent to the "signing" service on each node (customer and server)
- the trick is that the initial access to the signing service goes through the flocking protocol. That gives you a code that lets you send other stuff to the signing service through its normal (signed) API.

Let's walk through a scenario of a new node entering the cluster and accessing two services (**foo** and **bar**) on node B:

1. it sends a message through the flocking protocol to the signing service (running on node B) containing (USER, fingerprint, public key)
2. the response to 1) is access to the normal signing service on B. So now we send the data (USER, fingerprint, public key) for service **foo**.
3. Node A can now access service foo. We can now send our data (USER, fingerprint, public key) for accessing service bar.
4. we can now access bar.

The bottom line is this: when node A wants to talk to a service on node B, it either

- (the first time, or after the current signing key is invalidated) sets a signing password through the flock protocol. This would have to be faked out for testing.
- otherwise, we use the current password and talk directly to the signed API for signing.

Best practice would be use a node-to-node service for doing this; that is, all processes on node A would talk to a process on node A that collects and reuses passwords.

The actual details

Strew is a key,value store that lists the available services in the flock. You can look up a tuple (servicename, nodename) and get back the information you need to access that service (actualservicename, node_ip, port). The actualservicename is a (possibly random) string name guaranteed to be unique across the flock. The actualservicename, as well as the other information, is known to the service being described, and who is responsible for supplying the information to *strew*.

How services are referred to is more complicated. We anticipate multiple methods of looking up a service:

1. the actual name, such as ntp_2.3
2. a generic name, such as ntp

These names are supplied to the generic *call_a_service* routine for accessing a service.

Failure modes

The scheme outlined here involves a single executable (the fulcrum) that adds and executes services as Go functions (loadable from an external file) running as goroutines. We are thus susceptible to bad things (like panic's) in those functions. Ignoring those, the most awkward failures are the *walking dead*, functions that appear to be working (including heartbeats) but have in fact become wedged in some way. What do we do here? Just restart?

TBD

1) where do we get khan spec?
2) how do we find a logging service? (are there other services like this? block-chain?)
3) do we need a K/V store? (teh flocking protocol will get ugly with a lot of nodes)
4) are there other base fulcrum services (other than pastiche)? (i don't want to expand this ad infinitum, but do we need an admin function that does a reboot and such-like?)

Hordes

We anticipate most work is done at the horde level. A horde has persistence and the ability to restart automatically across power cycles.

khan

somethin on khan

Config

somethin on config

Vault

We need a place to hold secrets. (These days, you can barely take a step through infrastructure without needing various keys, tickets, IP addresses or other secret stuff.) We can use the same technique of using a small amount of preloaded keys to get things started, and once we have enough facilities (for example, persistent storage!), a more complete vault can start but, again, it can continue to offer services at the flock level (through a service registered there).

External publishing

We take as a requirement that flocking needs to happen without relying on access to an external network. That's all well and good, but how does the flock, or applications running on the flock, announce themselves to the outside world?

Our proposed answer is that an external address (IP or domain) of a rendezvous point is given as an attribute to the base computing environment. Just like the initial secondary key for the base flocking protocol is a specified argument. This address would be updateable via the basic flocking protocol (as the secondary key is now).

Once we have a way of publishing flock-specific stuff, then the access problems are similarly solved. For example, operator access can now be done via publishing an Ops portal address.

Work to do

This is meant to be a precise list of stuff to do.

1. on every node, the fulcrum runs a mind process (allows plugins to run)
2. leader needs to start the global strew. every node starts a local strew as well.
3. the global strew publishes (over the flocking protocol) its information, namely its netId and a reacquisition encryption key. it does this through an interface defined below, but will likely only happen every *n*th heartbeat.
4. after all nodes have gone through previous step, the global strew is populated and good to go via the normal gRPC whitelist interface
5. once we have strew running, we can then load pastiche, a plugin execution module, and

a flock-wide software configurator.

6. this means the initial fulcrum binary must contain the local and global strew, pastiche, plugin execution, and configurator modules.
7. at this point, every part of the fulcrum can be reloaded; we are bootstrapped. every node is running
    a. mind
    b. pastiche
    c. local strew
    d. plugin executor

    and the flock is running

    e. a global strew (on the leader, netID to be published)
    f. configurator (somewhere)
8. we're done.

A plugin runs is either

- normal case: the plugin is just given the global strew netID
- flocking module: the plugin gets, as well, a bunch of networking-specific stuff (like node name etc)

Health checks: we will have two independent schemes for doing health checks. Scheme 1 is the normal gRPC health check mechanism. The mind plugin will advertise a health check service that matches the gRPC mechanism. It will advertise (more or less) the health of the node.

Scheme 2 is service-node-specific; each service running on a node will heartbeat to the global strew service the tuple (time, status). The periodicity of the heartbeats is given by an input parameter (the so-called quit channel); it is initially 10 seconds.

The configurator essentially works by sending a node's current functionality (in terms of service names), and returning a desired set of services. This may involve starting and/or stopping various plugins. In addition, the returned list will have the desired heartbeats for each service.

We note that normally all interactions with the global strew server take place over its gRPC interface. There is a rare exception, used only for startup and reacquisition, which uses a separate interface that is NOT gRPC whitelist protected. This interface is protected by a symmetric encryption key shared over the flocking protocol. The data itself is encrypted. The information encrypted is a grpcsig protected callback service and public key information. This amounts to a 2-factor interface. Factor 1 is the symmetric encryption key which must be successfully decrypted on the global strew server; factor 2 is the callback, which must provide a successful connection, a reply, and a grpc-signature which is authenticated by the global strew.

Coding tasks

Repo Organization
crux/protos
- put in grpc health check (with extra enum) into all .proto files

crux/pkg
- set up pkg/stewart, pkg/reeve, pkg/proctor, .protos

Strew API doc - For reference on Strew tasks

MIND
mind()
- reeve reports local plugin services - extracted from mind.
- mind.Services() locally - extract netIds from mind = running services on node
- mind() [via reeve?] calls proctor.State(flockid, []netId) and gets back OK, or a bundle of deltas to implement.
- mind().GRPCHealthCheck running (empty service string) is the definition of a node being "up" (metaservice is up)
- when I am a newbie flock leader - start up steward and Consul here
- server that listens to proctor for changes/updates to running plugins - invokes picket to make changes

LOCAL Strew (aka reeve, previously "stru")
- gets flockid, []netId from mind (stub in to test)
- gets flock encryption key, steward netId from flocking protocol.
- reports []flockId, bundle of keyId ([]current, []deprecated), []netId
- maintains local public/private key pairs
- maintains local grpcsig whitelist
- RollKeys() - handle "roll keys now" command from steward
[- HeartBeatListen() aggregates all local service heartbeats (or grpc Health Check calls)
- PushHeartBeats() - Push Heartbeat aggregated bundle to steward().HeartBeatListen]
    This may be grpcHealthcheck data to mind/picket interface via channels...
- netId to Dial() generic function (sets up ssh-agent for signing)
- asks/listens to Configurator how often to push heartbeats, pull health checks.
- run this as a mind plugin, internal call to mind().Services to collect them from node
- start up and Register stru instance with steward (via reverse grpcsig mechanism)
- calls steward.ClientsAllowed(netId) to collect allowed public keys for a service.
- calls steward.Register() with encryption
- ReRegister() endpoint/or interface to mind for when global steward is lost.

GLOBAL Strew (aka steward)

- maintains flock public keys (keyIds), services (netIds), flock configuration (flockId) data
- steward - gRPC initial API into .proto
- Register() non-grpcsig function, encryption, decryption
- stewart callback to reeve mechanisms (reverse grpcsig validation)
- HeartBeatListen() collects []service-heartbeats off node in bundle.
- steward - k/v store connector (lift from Stix)
- Consul startup on steward node (lift from Stix).
- Consul k/v implementation interface (lift from Stix)
- Populate k/v with []flockId, [][]keyId, []netId
- Query() API interface, calls, as needed.
- query flockId can be wildcarded (e.g. */*/*/servicename/servicerev)
- Given a flockId (or part, e.g. nodename) - get a netId.
- Admin() API calls (add/delete userid, etc)

GLOBAL Configurator (aka proctor)
- uses stewart query API
- maintains a list of flockId/servicerev   as target states.
- maintains deployed nodenames, servicenames, servicerevs, provides heartbeat intervals.
- CheckGoals() API call - recieves running services, compare to target “should be running”
- replies with “deltas” []add []deprecate []delete services.
- Admin() API call - - Upgrade(from flockId/servicerev, to flockId/servicerev)
- Admin.Initialize - bulk information as to the target state of the cluster

Plugin Launcher (aka picket)
- starts plugins
- serves aggregate gRPC Health Check information for each service running in a plugin
- intra-process gRPC health check each plugin via channels to/from plugin
- client that reports health status + timestamp + other to steward

Endor (aka endor)
- plugin formerly thing known as endor
- Begat work runner
- thing that runs exec() on a node for authorized users.

HeartBeat Push to Steward (not grpc Health Check)
All Plugins implement a HeartBeat client that Dials() steward and registers
{netId, timestamp, status…} (grpc signature provided with private keyid (userid = serviceid)
matching the serviceid in netId.

Steward has an HeartBeat Listener API call that takes {netId, timestamp, status…}, HeartBeat whitelist is all keyIds.

Health Check & HeartBeat Absolutely Final Reification. We mean it.

- **Steward** has both registration data and heartbeat data for services.
- ?? Is there a registration in the **steward** for services that no one has requested yet? No consensus as of 5:30pm Thursday April 26.

- **Plugins** will be launched by the **Picket** with a channel to heartbeat health data back to the [mind| picket | new-health-thingy]. For now we'll say **health-minder**.
    - This health data consists of the service name, and status (up,down,unknown)
    - Heartbeating must begin before the plugin sends a registration request to the global **Steward**. (Which calls back to the **Reeve** to get data)
- **Health-minder** waits for status on the plugin channels, aggregates them and sends them to the local **Reeve**, and stores locally for future query via gRPC healthcheck.
- Also, the **health-minder** has a GRPC health check API that can be used to get the data for any plugin by passing the plugin name in the healthcheck's string argument

Tues May 8:
Skaar: OTOH, it's not bad for each service to expose it's own health…..

Pastiche and Bootstrapping

Skaar: self muck idea
-

Pastiche distributing Binaries? No directly, but called by XXX to find and place a plugin on every node. Maybe use the new healthcheck plugin to test this pathway. Skaar doesn't want healthchecking to rely on a distribution step.
- Building a binary (example in integration test directory of build and run plugin)
- Pastiche muck file for configuration. Will need this to get pastiche running first.
-

Crux Error Proposal

1. Crux routines will return the normal go error interface (like everyone else does)
2. Crux routines will actually return a *crux.Err, generated by the existing routines.
3. There is a crux function with prototype **function Augment(e error) error** that will add any additional information to an existing error (existing fields will not be updated).
4. Generally, any extra fields will not be reflected in the .Error() string value.
5. There will I a standard way to encode our error in a protobuf message (not required).
6. We will use the gRPC error codes.

Example from pastiche/server.go

Now

```
// DeleteAll - Clear all cache directories of all files.  Remove
// entries from cache lookup table
func (ps *Server) DeleteAll(ctx context.Context, dReq *pb.DeleteAllRequest)
(*pb.DeleteAllResponse, error) {
      cerr := ps.store.DeleteAll()    << This returns *crux.Err
      if cerr != nil {
      # Here we pull the stack out of cerr to put in error string for grpc.  Yes, it's not using a
      grpc error type, but lets ignore that for now.
            return &pb.DeleteAllResponse{Success: false}, fmt.Errorf(" %s
%s),cerr.Stack, cerr.Error())
      }
      return &pb.DeleteAllResponse{Success: true}, cerr
}


Func (bs *blobstore) DeleteAll() error {
      err := somestdlibfunc()
      return crux.ErrE(err)
}
```

Proposed

We support two complementary schemes. One is support for encoding an error directly in the returned object. The other is encouraging (demanding) that we use the existing gRPC error codes.

```
// DeleteAll - Clear all cache directories of all files.  Remove
// entries from cache lookup table
func (ps *Server) DeleteAll(ctx context.Context, dReq *pb.DeleteAllRequest)
(*pb.DeleteAllResponse, error) {
	err := ps.store.DeleteAll()
	if err != nil {
		log.Logger(blah blah) // log actual error
		return &pb.DeleteAllResponse{Success: false, err:
crux.Encode(err)},status.Error(codes.Unknown, "delete failed")
	}
	return &pb.DeleteAllResponse{Success: true}, nil
}
```

New code in crux/error.go

```
type Stacker interface {
	// Whatever you want to be able to do.
	Stack() string
}

// IsCrux - is it?
func IsCrux(err error) bool {
	ce,ok:=err.(Stacker)
	return ok
}

// Stack - just sugar
func Stack(err error) string {
	ce,ok:=err.(Stacker)
	if ok {
		return ce.Stack()
	}
	return ""
}
```

Light reading

https://dave.cheney.net/2014/12/24/inspecting-errors
https://dave.cheney.net/paste/gocon-spring-2016.pdf

Grpc status codes package

https://godoc.org/google.golang.org/grpc/status#ErrorProto