Where we are

We've been working on Radix for a while now (see doodle42xx for our thoughts there). Lately, our efforts have been focussed on the beginnings of things, and in particular, how do we do stuff securely and being able to start from scratch (and thus avoid the Google infrastructure problem). The rest of this note is a map for how we think of what we've done.

Groups

We think of computers as groups of cooperating nodes connected by a group identity and a set of services offered by nodes within that group. The services are TCP-connected to gRPC services that are whitelist controlled. To be considered a group, there is a set of basic services shared by the members of that group:

- ***reeve/steward***: service registration/heartbeat/whitelist
- ***proctor***: service placement
- ***pastiche***: data movement
- ***picket***: (plugin) service invocation
- ***muster***: what services are supported
- ***confab***: networking

We'll describe these below.

We implement two sets of groups; an underlying basic group, which we'll call a flock, and upper groups, or hordes. These groups are very similar and offer identical facilities to run applications and other services. They differ in their audience and scale, and how hard they are to bootstrap.

Flock/Bloc

The primary scenario for flocks is a datacenter with thousands of nodes that has just been powered up. How does that set of nodes bootstrap up to something that you can use?

Nodes can be divided into groups, called *blocs*; a bloc is a set of nodes with the same *bloc ID*. Flocking then occurs with a single bloc. This allows nodes to be separable and independent of other flocking activities.

The *flock* protocol, a UDP-based gossiping protocol that detects other nodes, runs inside a bloc with the goal of ending up with a single group (modulo networking issues) with a leader and a service registry. All the flocking networking is encrypted with symmetric keys; primarily a transient key (lifetime of 30-60 seconds, updated automatically by the protocol) and a secondary key (lifetime of hours/days, updated on demand). The flock protocol is implemented as a Go interface; it would be trivial to implement using other techniques.

Horde

A horde is a subset of a flock. It is primarily an administrative unit, (will have) admission control, is named by users, and is the normal place to run applications. We anticipate that multiple hordes (possibly from different flocks) will federate together. While our gRPC is whitelist protected, we anticipate encoding the content as well by a gRPC plugin, such as TLS.

Group services

These are the standard group-level services.

Reeve/Steward

This service does two jobs:

- it registers services and distributes their public keys. The (public) keys are unique to the tuple (originating node, destination end-point)
- it is a lookup service for all services in that group

This service is implemented as two parts: a local (per-node) service called *reeve*, and a single group-wide service called *steward*. These services can be started with empty content and are updated continuously. The steward is not meant to be invoked directly; there are proxy routines within *reeve* to support steward calls.

Muster

For now, *steward* will also support the *muster* functions. This is essentially an inventory of available functions, where you can register a tuple

(horde, api, instance, filename, functionname, timestamp)

and then search for it by either/both of

(horde, api, instance)

This should be good enough to do what we need.

Proctor/dean

This service arranges origination and placement of services. Periodically, proctor examines the list of services from steward and conforms them to a global list. These lists are teleological; they are what should be and not necessarily what is. For now, we are using *khan*.

Proctor is used in two distinct ways: locally (per-node) and cluster wide. The local version (*dean*) runs locally on the node and does not require networking and is meant to just support applications running on that node. The global version (*proctor*) is supported by a single entity and handles applications running across the cluster.

Pastiche

Pastiche moves data between nodes. It runs as a per-node instance and uses proctor to find other pastiche instances. Data are regular files, named by their checksum. Mainly, the data in the local cache are pulled from other servers, but can also include files loaded from local directories. (These will be added to the cache upon request).

Picket

The final basic service is *picket*, which allows you to run a service (as a Go plugin). It does some plumbing such that all services can be updated during a running system.

Networking/confab

We have captured the networking into a Go interface `confab`. It assumes that nodes have addresses and that the node knows its own address, and that the addresses are "reasonably dense" (a node has a reasonable way to guess other addresses).

General

How do we invoke remote services? It turns out there are several different answers.
But before we show them, let's define a simple service in **crux/protos/srv_galumph.proto**:

```
service Galumph {
	rpc Register(GalumphInstance) returns (RegisterResp) {}
	rpc Table(TableReq) returns (TableResp) {}
}
```

At the lowest level, a service is implemented by a call like this

```
	func Galumph2_3(quit <-chan bool, alive chan<- []crux.Fservice, network
**crux.Confab)
```

By convention, this implements version 2.3 of the galumph service (more on versioning below). The quit channel says when to quit; this is independent of other ways to quit (such as a protocol message). The alive channel is where heartbeats (same as registration) are sent. A service routine can implement multiple service points; a heartbeat needs to be sent for each one. The network parameter can normally be ignored; it is mainly for the underlying networking code.

Server-side

The server code might look like

```
func Galumph2_3(quit <-chan bool, alive chan<- []crux.Fservice, network
**crux.Confab){
	g := Galumph{quit:quit, heartbeat:alive, version:"galumph2.3"}
	g.addr, err := pb.ServerGalumphServer("", &g, 5*time.Second)
}
```

First, we initialise g, an internal data structure passed to all the service requests. We then execute the gRPC server loop, which returns the network address for the service. The server arguments are <some database thing for whitelist>, the internal data structure, and the time is the heartbeat interval.

The service routines are done the normal gRPC way, with definitions like

```
      func (g *Galumph) Register(ctx context.Context, inst
*pb.GalumphInstance) (*pb.RegisterResp, error)
      func (g *Galumph) Table(ctx context.Context, tab *pb.TableReq)
(*pb.TableResp, error)
```

In addition, you need to define a heartbeat function like this

```
      func (g *Galumph) Heartbeat(ctx context.Context, in *pb.HeartbeatReq)
(*pb.HeartbeatReply, error) {
            g.alive <- []crux.Fservice{}
            return nil, nil
      }
```

Client-side

The client side is quite straightforward. E.g.

```
      gal := ConnectGalumph(<req>, 0)
      err := gal.Register(pb.GalumphInstance{}, gal.RegResp(*pb.RegisterResp,
error))
      err = gal.Table(pb.TableReq{}, gal.TabResp(*pb.TableResp, error))
      gal.Close()
```

Except for the initial connect call, this is standard gRPC stuff. The initial connect call does quite a lot for you! It takes a selector structure and a time duration t; every t seconds, it will rescan the service database for a "better" fit for your selector and connect to that. The intent is that clients can gracefully connect to new service instances (both version and position) without any extra work.

Versions

When a service registers a heartbeat, it sets these fields (amongst many others):

api	"ntp2"
inst	"ntp2_3"
netID	node345:1234

We expect a selector to either specify an api or an inst. All instances for the same api service the same protocol; you might start off with api:"ntp2" and get a "ntp2.1" but over time, you might end up talking to a "ntp2.7" instance. When comparing instances for a particular api, we pick the lexicographically greater name. If you specify a specific instance, say "ntp2.4", then you will only run on that specific instance.

Heartbeats

We want to clarify exactly how heartbeats are handled. As we described above, heartbeats are sent over a channel and eventually end up in both the steward and the *healthminder* (Google availability) service. We use an internal *heartbeater* monitor which sits on the sink end of the heartbeat channel. This monitor will maintain its own view of what services are available (that is, it decides what is alive and dead). It will broadcast its view of what is alive and dead to both the steward and google availability service. In this way, it can

- reduce traffic outwards to steward et al
- maintain a consistent view of service availability

The fields in a heartbeat are

- NetID
- FlockID
- timestamp (when heartbeat was issued)
- timestamp (when heartbeat is valid until)
- status

For now, the heartbeat server is considered to be part of the *picket* service.

Bootstrapping

Every node should bootstrap in the same way. The overall scheme is

1. start flocking
2. start initial services (*reeve*, *picket*)
3. start other common (predefined builtin) services (*proctor*, *pastiche*, *healthminder*, etc)
4. use *proctor* to maintain a cluster-wide list of services

There are a few different ways to arrange how the first few steps get done; they vary mainly in the uniformity in how services get started. For example, how many services get invoked directly versus how many get invoked through *picket*. Our initial plan is

1. start flocking
2. start *heartbeater* service (directly from bootstrap)
3. start *reeve* (and *steward*) (directly from bootstrap)
4. start *picket* (directly from bootstrap)
5. preload a builtin list of services, and then start *proctor* (through *picket*) to start those services (which would include *pastiche*)

6. at this point, we need to decide how to get/maintain the cluster-wide list of services, and have *proctor* run from that list

This implies that the initial versions of *heartbeater*, *reeve*/*steward*, *picket*, *proctor*, *pastiche* (and anything else needed for step 5) need to be compiled into the fulcrum binary (so that we don't need *pastiche* to do file copies). Once this initial set of services is downloaded, we can then overlay them with newer versions (using pastiche to transport files around).

===
the remaining is less complete.
===

Persistence

group info acts as a service. add more. manifest come from S3 source (minio).

Other group services

We will need more services than this, i'm sure. This includes
khan
begat
healthminder
group info (manifests?)

TBD

logging: especially online controls
metrics

manifests
how to run containers or other distrib apps
security system for ENM
(bug: what happens when (due to a network error) two distinct flocks join? pollution of artifacts. have a user-designated flock name??)

How to update services

This is a step-by-step to how we should do product transitions.

First, we have a model:

- generate new artifacts (binaries etc)
- deploy the artifacts
- activate the artifacts

Generate the artifacts

This can be done internally within a horde, or externally on a remote system. In either case, we need to do two things:

1. put the artifacts somewhere where pastiche can get at them (and let pastiche know)
2. update the service registry with the tuple (api, instance, filename, functionname, timestamp)

We will likely have some command that will copy a binary generated by buildbot to somewhere where pastiche can get at it; the current thought is to have a minio-like service presenting an S3-style interface.

Deploying the artifacts

There is a question of how to do this in practice. Probably, the cleanest solution is to run a wee job that fetches a file (with a known checksum) from a given S3 interface and then tell pastiche to cache that copy.

Activating the artifacts

For most services, where we are updating a service with the same API, this will update within a small amount of time. That is, the client interface automatically checks for an more up-to-date version. If we talking about a new service, then this is harder. We need to correlate

1. telling proctor what to do (so that we know the intent)
2. asking steward what the observable state is
3. mapping 1&2 into a measure of doneness

This presumes that we can achieve a new state of a service in parallel with running an old state. This is not true in all cases, but for now, we'll say it is.

This also implies that *khan* (or whatever system we use) has to use a little language to select an entity to run.

Manifests

Manifests are basically adjuncts to an application. Typically, they contain a K/V store of variables and values that details various aspects of the application's operation. If this is all we need or want, then the combination of Cobra and Viper offer a per node/application implementation that ties in well with a combination of command line arguments, shell variable and text file sourcing.

There is more to this, though, for a distributed application. This seems mainly due to the fact that a distributed application may need distributed properties. For example,

- placement options (how many copies of different parts, and where they may run)
- how do you actually implement the distributed aspects of this information
- practical implementation issues (for example, different networking choices for small versus large clusters)
- strategic implementation choices (for example, backup strategy)

Our previous thinking has been to have an *envoy*, an underlying single-threaded application (running in the underverse) that manages all this stuff. And in general, you may need the complexity this supports (for example, if you need to start Consul or other multi-node support applications). But this is a heavy cross to bear.

<note to andrew> elaborate on bipartite scheme of control
examine what exactly the kub control scheme is (via loren)