Chronicles of Radix (v2)

This document describes the Radix, an exploratory system for construction of distributed systems with special attention to composition of topologies and production of artifacts with provenance, reproducibility and integrity.

1. Radix: writing and running distributed applications

This first section gives context for Radix, followed by an explanation/description of the model behind Radix and its major parts. The following outline captures some of the core questions and properties we are working toward solving with components of Radix.

1.1. Overview

Like everyone else, we want, and need, to do distributed computing. We believe that fundamental differentiators in this area are to be found in provenance, integrity and advanced control primitives. We'd like to do this in as much a platform-agnostic way as possible.

Notes from review:

- Telco being driven by NFV
- Security , performance guarantees (real-time issues too)
- Legacy apps not written as distributed apps. Concurrency, but not cluster-ized.
- “Just use kubernetes” not an answer to these people. They’re making slow progress towards distributed.

- Goal: give people a model on how to write distributed applications.
- Underverse is this model.
- Security more important than when rnodes, network and software all proprietary. (provenance of builds, etc.) Legal issues involved with being infrastructure and/or govt entities.
- Industry hasn't done a good job with integrity and provenance.

Modern services and solutions requires that its software and distributed applications can be developed, constructed, and run on distributed clouds. Ericsson needs to be able to build, deploy and upgrade systems running on these clouds that solve customer's problems. It needs to do this nimbly, and at scale (from tens to tens of thousands of rnodes), and in a way that it can certify (as best we can) the environments as well as computational results.

How do we progress to that vision? We will start to evolve a definition of a control and management system named Radix, a loosely coupled collection of tools and services. For the near term, the Erix team will explore several domains of activity:

First, we need to build our software. We start by constructing ***begat***, a build system that lets us construct software efficiently, reliably and with provenance. Part of that efficiency comes from doing distributed builds, where computation is spread across multiple containers. How do you build those container images? We'll use ***hermetic***, a system for reliably building system images from specific, verified inputs.

Second, we need a way to host these containers. For now, we will use ***myriad***, a tool that lets you conveniently spawn a number of containers using a variety of support environments, such as Docker and EC2. In the "Threshold" section below, we discuss other methods of hosting containers, such as Kubernetes or schemes based on "cold iron" (bootstrapping from a reboot).

Thirdly, we need to build clusters on top of these containers, and application to run in these clusters (*begat* itself will be such an application!). Much of this will come from a group of packages and tools called ***crux***. This will start with some software we found useful in building other projects, such as *stix* and *cloudhelm*, and extend that with blockchain technology and ideas from the micro-services world. In particular, blockchain will allow an improved level of certifiable provenance. We also extend and formalise the relationship between the application and Radix by introducing the ***envoy***, a counterpart (to the application) that operates farther down the stack.

Fourthly, we need understand how all the above can integrate with the concept of a ***ledger***; a write-once, verifiable, permanent, externally inspectable record. Its is just not an issue of what to store, and when to store it; it is also about what tools you need to make effective use of it. Currently, we anticipate using blockchain technology for the underlying implementation.

We work this problem from the particular focus of the needs and lifecycle of distributed applications, with the possible outcome that there might not be one uniform platform to be found at the end - but rather an ecology of federated systems to be assembled into regions of stability in which we can run application functions and services.

1.2. Basic terms

A ***rnode*** is a collection of resources. It has a *name*, its *cluster's name*, and a few micro-services, including a control service and a service that executes arbitrary executables. A rnode has the following properties:

- exactly one ***fulcrum***, or Radix agent. There may be more transiently during an upgrade.
- a persistent ***memoir***, or audit log. This will be available for some time after a rnode finishes (in order to support post hoc analysis).
- a rnode manages a set of resources, and this set of resources is managed only by this rnode.
- a rnode belongs to exactly one cluster
- a rnode advertises a number of services, as well as a heartbeat, to all the rnodes in its cluster.

There are no other implications; for example

- multiple rnodes can share a computational context, such as a VM, physical server or container.
- multiple rnodes can share an IP address

A ***cluster*** is a set of rnodes with the same *cluster name* field. All the rnodes in a cluster share a limited amount of information. A cluster has a *leader* rnode; its only distinguishing property is that there is exactly one leader (ignoring transitions from one leader to another). Leaders are used in cluster formation, but outside of that, their main use is provide an easy way to ensure that only one instance of specific processes are running in a cluster. Clusters can also join together; see below under "Federation".

Clearly, if rnodes are to form clusters, they need to share information with each other. There are several choices:

- a simple message-based scheme based on the **Ken** protocol
- the One-Hop bandwidth-efficient messaging scheme (ref: A. Gupta, B. Liskov, and R. Rodrigues. *One Hop Lookups for Peer-to-Peer Overlays*. Ninth Workshop on Hot Topics in Operating Systems (HotOS), 2003.)
- HashiCorp’s Serf implementation of SWIM Gossip protocol (underlying Consul)
- a cluster wide key/value store, such as **etcd**, **Zookeeper** or **Consul**

These differ in cost of maintenance, scalability, bandwidth usage. Note, too, that applications have similar needs, and they could use the same schemes.

Services are accessed through gRPC sited at a specific address (an IP,port pair). (There is an isomorphic version of this implemented as a REST interface.) There is a cluster-wide registry that maps a service identifier to its address (perhaps addresses once we figure out how we handle load balancing and HA stuff). This registry is likely a distinct service (a la Consul or somesuch), but the address of this service would be part of the cluster-wide shared data.

We anticipate that rnodes will be implemented as a computational entity, such as a bare server, or VM, or container, running an agent which we call the ***fulcrum***. The intent is that the fulcrum be, and remain, a simple thing, basically

- maintaining cluster membership
- sharing cluster data through the Ken protocol (for simplicity)
- maintaining per-rnode services
    - remote execution
    - resource management
    - fulcrum functions (for example, configure remote execution, upgrade, exit)
- leader election

We expect to simplify and unify these "low level" functions by thinking of them as a set of micro-services. Furthermore, we can also constrain what commands can be executed remotely by configuration (through the `fulcrum` service) before use by specifying the actual services offered. For example, if our cluster is only running the *stix* application, we would constrain the rexec service to only run stix commands:

rnodename/`fulcrum`	*ip*/port1
rnodename/`rexec/stix`	*ip*/port2
rnodename/`fileio`	*ip*/port3

Obviously, any such configurations, as well as any commands executed would be logged in the rnode's memoir.

1.3. Services

We follow the Hickey theory of service versioning, namely that a new version of a service either extends the service, or it breaks the service. That is, if code expects a certain functionality from a service `foo`, it will always get that functionality from `foo`. If a new version of `foo` cannot do this, then it can't be called `foo` any more. There is more to this than might be obvious. When a client uses the NTP service, it is actually doing multiple things:

1. it is expecting and using a specific bundle of functionality; let's call this **ntp-a**.
2. the client is tested against a specific version of the service; let's call this **ntp-2.2**.

3. because no one wants to manage the compatibility matrix of functionality bundles and server versions, the client accesses the server by the string "ntp". This insulates the client source code from harmless changes in the server version, but at the cost of determinism and accuracy in testing.
4. in truth, there has to be a way to denoting server versions by some sort of checksum; it seems begat and alembic will need this. Chris's work with *faceup* partially investigated how services might be characterised with multiple checksums.

Conceptually, we expect a client to access a service through a handle, such as

```
srv := crux.Service("ntp")
tim,err := srv.Req(&ntp.GetTime{"los angeles"})
```

The service handle would implement retries and deal with underlying changes in the address for the "ntp" service.

The implementation of a service has 3 main aspects:

1. when ready to offer its service, it (normally) registers its address under a specific name
2. a list of requirements which constrain the possible rnodes where that server can run. These requirements can be
    a. rnode resources (such as local persistent storage or GPU)
    b. metadata properties of the above resources
    c. access to cluster resources (such as shared persistent storage)
    d. properties (such as that user on that rnode can access some other service or resource)
3. a list of requirements and/or attributes needed to drive upgrading the service. This is a superset of (2) above (because we need to know where the new server can run) and some reasonably generic set of properties that can be used by a service-agnostic tool for managing the service upgrade.

We observe that documenting a service's requirements (as in 2 above) need not be just a one-time check prior to starting a service. It can also be monitored periodically, detecting some failures in real time, such as

- access failures (such as an expired certificate or failure in the connection to shared storage)
- reachability of certain rnodes (perhaps defeated by a network partitioning)

We anticipate a suite of tools to help manage different kinds of upgrades; these are described below in the Transformations section.

1.4. Data

Any cluster aware system has to deal with a large variety of schemes that support persistent data. Before we delve into this, it is worthwhile to take a step back and clarify what we're talking about.

A ***data file*** is the quanta of data we can access by name; for example, a UNIX file or a stream of bytes identified by a URL.
A ***data set*** is a named collection of data files; for example, a tarball, a macropod pouch, or a chunk of data that can be mounted as a filesystem.
Data can either be ***ephemeral*** or ***persistent***, depending on whether the data is intended to survive a rnode restart. Note that even if data persists, its integrity still needs to be verified.
Some data is ***local***; that is, it is present even if there is no networking working. This distinction matters for early bootstrapping where we need some data (such as an executable) in order to bring up networking and other services. Keep in mind there are many kinds of non-local storage, such as object services (like S3), network file systems (NFS), and remote block device services (Cinder) that can either be the data itself or something that can be interpreted (mounted) as a file system.

So what does Radix think about data? It cares about just a few aspects:

- discovery (of local data): what data sets are local (and accessible) to a rnode
- discovery (of non-local data): data sets or files are available to (accessible from) a rnode; for example, in the sense of prerequisites for a service or other computation.
- altering what discovery would find: changing the data sets or files accessible from a rnode. This most likely means copying files from some source to a specific rnode, but could also involve deleting files, mounting or unmounting filesystems, or accessing different data services such as S3.

There is no implication here about whether or how changes to data on a rnode are visible on other rnodes. Some data services, like network filesystems (NFS) and object stores (S3) and distributed key/value stores (Zookeeper, Consul, Corfu), do make such changes visible on other rnodes, but that is a property of that service, and not Radix per se.

We observe there are a couple of ways to think about (the most common case of) copying files:

- an imperative API that says copy file *f* from rnode *a* to rnode *b*
- a declarative API that says file *f* needs to exist on rnode *a* (and let Radix figure out where to get the file from)

Of course, both APIs can coexist simultaneously (in fact, a daemon might use the imperative form to enforce the declarative form). There is also the pragmatic issue of whether we prefer push or pull techniques; it is easier to manage load issues with pulls, while it is a little easier to make pushes more secure. In both cases, though, a centralised oracle can do a close to optimal job of scheduling these transfers.

By the by, the declarative view easily extends to handling versions, as files are fundamentally identified by their checksum.

1.5. Security

general model; overview

Mutual authentication for endpoints via xxxx
What to do for the data on-disk?

1.6. Example: a higher level view

The mechanisms and interfaces described above are, in some sense, Radix. But this doesn't mean we expect application developers or DevOps to use these facilities directly. We can leverage the growing trend of describing your problem with higher level description files, and then using a combination of declarative goals and generated code, have Radix facilities do as much of the fussy work for you. This section describes one such higher level view. Be warned: it describes a richer version, and some details may conflict with other parts of this document; roll with it!

1.6.1. The application view

We assume the application is a collection of potentially distinct executables, and that these executables need to run on various sets of rnodes within our cluster. The application will use Consul as an application-wide blackboard.

We need the begat-like description for the executables (we'll use make-like syntax for now) in **stix.begat**:

```
include(consul)   # defines cserver and cagent
prog1: rexec/worker prod=/var/stix/prods.csv userid=stix
      rexec/worker -rnode %rnode -ip %ip -task worker1 -prod %prod
prog2: rexec/slave userid=stix
      rexec/slave -rnode %rnode -ip %ip -task slave6
```

We need to configure the rexec service in **stix.rexec**:

```
include(consul)
worker = /var/stix/bin/worker
slave = /var/stix/bin/slave
```

We need to say how many and where these run in **stix.khan**:

```
cserv := pick(cserver, 3, ALL)
pick(cagent, size(ALL)-size(cserv), ~cserv)
pick(prog1, 2, ALL)
pick(prog2, size(ALL), ALL)
```

Finally, we need to say where the rnodes come from in **stix.rnodes**:

```
myriad {
      n = 6
      cmd = /var/radix/bin/fulcrum
      mount /user/andrew/code/stix = /var/stix
```

```
}
```

One could imagine an OpenStack or a Kubernetes version of **stix.rnodes**.

Phew! That was a bunch of stuff, but it all seems needed.

The rest of the action takes place behind the scenes, performed by various Radix services (that we'll call the infrastructure) that achieve cluster-wide goals by a sequence of rnode requests.

1.6.2. Infrastructure: data movement

We assume the rule that any executed program must be a local file. In order to do that, the ***data service*** (for want of a better name) first scans each rnode in the cluster for a list of accessible files. By processing the various files described above, we end up with a list of rnodes and the files (both data and executable) needed on each rnode. The data service then figures out how to optimally copy files from where they are to where they need to be and then schedules the copies.

Typically, the list of files needed on a rnode will come from **stix.rexec**, because the only way to execute a desired program on a rnode is through the rexec service, and we only know what is needed when the rexec configuration is loaded. This allows for some fanciness. The simplest approach would be to only return from setting the configuration when all the needed executables are present. You could also implement asynchronous loading, where copying the executables is scheduled upon loading the configuration and hoping the copies complete before they are needed. In the hopefully rare case of being invoked before the file is present, the rexec service simply delays execution until the file is present.

A couple of notes with respect designating executable files:

1. It would seem wise to avoid the pathname method of identifying an executable file. Unix based systems inherit a complicated mechanism of search paths, viewpaths and so on in the name of flexibility. A simpler and more robust scheme would be to simply refer to an executable by its hash; the specific linkage between a human-usable name (like stix) and the corresponding hash would be established by some exporting mechanism within the build process.
2. The same issue comes up with how an application designates what version of a service it needs. As we described above, an app might use the `dns` service. We test the app using (say) **dns2.2**. We then upgrade the dns service to **dns2.4**. Because the name has remained the same (`dns`), dns2.4 is a strict superset of dns2.2 and everything should be fine. We add a new app that needs dns2.4 (and won't run with dns2.2). We propagate dns2.4 to all the rnodes and everything is fine. A rnode which had been under maintenance is then powered back up but is now running with dns2.2. This is bad; even though this rnode is offering a dns service, it is the wrong one but how can the app tell? There are easy solutions, with varying amounts of paperwork; TBD.

The bottom line is that the data service is driven declaratively; we specify what we want to be where, and it just gets done.

1.7. Federation

This is a set of thoughts about the issues around uniting multiple clusters together, or federation. And what we need to decide (in some cases).

1.7.1. Federation versus “There is but one cluster”

Technically, this issue revolves around how autonomous the clusters remain after unification. Here is a table of contrasting elements:

Federation	Single Cluster
Each cluster has its own namespaces, so fully qualified names include a cluster name.	All namespaces are shared amongst all nodes in all clusters.
Clusters can be connected by slow and/or unreliable WAN links.	All nodes have reliable connectivity to all other nodes.
The clusters may be quite heterogenous; each cluster may have distinct jurisdiction, user identity schemes, policies.	The clusters operate under a single set of policies, user ids, and so on.
Generally, access to cluster services goes through a portal/broker, and the connections may be one-way.	Generally, access to services goes straight to the node providing the service, and the connection is generally two-way.
Generally, the cluster service software merely needs to support interoperable versions of the protocols used.	Generally, the nodes need to run the same version of software.
Supports Availability and Partition Tolerance from CAP	Supports Consistency and Availability from CAP
Generally, service requests will have longer latency and may often be two phase (request and commit)	Generally, service requests can be answered with low latency and certain knowledge.
The clusters may federate some services, but not necessarily all.	All services are unified.
Generally, resources cannot be shared between clusters.	Resources are allowed to be shared between clusters.

Some of the above points can have gradations or nuances, but I wanted to emphasise the differences.

1.7.2. Application Viewpoint

An application interacts with the platform in a few ways:

1. execute a new process with specified arguments, environment. Normally, the user will have specified (even if by default) a policy dictating where the new process runs, but may override that policy for a specific execution. This policy covers various attributes such as
    a. number of cores
    b. amount of memory
    c. I/O bandwidth cap
    d. network bandwidth cap
    e. affinity (or anti-affinity) to other processes and/or clusters
    f. able to access a specific resource (e.g. a shared filesystem)
2. logging service. It must be possible for the user to retrieve all the logs from an application regardless of which cluster it ran on. For example, either by providing a single log, or by providing an iterator for log services to contact.
3. consistent distributed data service (e.g. etcd). Generally, these services are single cluster services, so we need to provide a method for some useful form of federating multiple services.
4. metrics service. It must be possible for the user to retrieve all the metrics from an application regardless of which cluster it ran on. For example, either by providing a single metrics stream, or by providing an iterator for metrics services to contact.
5. generally binding of a resource to a process. Most commonly, this will be storage of some kind (a shared filesystem, or object store), but applies to all sorts of resources.

It is important that the policies above be settable independently of any specific application, for most applications won't care about most of these settings. For example, an application might want 20 instances of a specific process; thats all it cares about. But the user might want at least 5 in each of three different clusters for various reasons (such as jurisdiction).

1.7.3. Resource Scheduler Viewpoint

Obviously, scheduling such processes with all their associated resource requirements is an extremely hard problem that is not yet solved. There are many considerations; a very partial list:

- processes might be classified as either interactive or batch
- negotiating between clusters might require a longer-than-usual amount of time
- there might be multiple rounds of scheduling; for example, admission control into a cluster, followed by dynamic reassignment within that cluster

1.8. Why should I care about Radix?

It depends on how you relate to the application: do you write the app? Do you run and manage it? Do you use it?

(Succinct answer to Martin's question about what this adds to K8's)
Scale across? Forward looking things vs what's useful for our target apps (see Section 2, For want of an application)

We believe distributed cloud applications required a base set of services. Different container orchestration tools offer some of these services already. Kubernetes, for instance, would have a smaller implementation of (underverse? radix?) than Openstack, Swarm, or AWS.

1.8.1. App developer

If you already have a distributed application, designed for a platform like OpenStack or Kubernetes, porting it to Radix should be straightforward.

If you are either developing an application from scratch, or adapting a non-distributed application, then Radix provides a model for how to do and manage "distributed-ness" that is easier to use and enough tools to help develop, test and manage the distributed application.

more?

1.8.2. Devops

Radix addresses many issues important to deploying and operating applications. The emphasis is on declarative specifications which allow for minimal (if not zero) touch mechanisms for placement of application parts, adaptive scaling of capacity and facilitating application upgrades. It could also help with sampling various internal dataflows for debugging and monitoring.

1.8.3. User

In some sense, we hope that Radix is invisible to the user, except that the application is more responsive and less prone to outages etc.

1.9. Not quite Radix

There is some stuff which is not part of Radix proper, but that which we use in conjunction with Radix:

- bootstrapping stuff

2. Application types being targeted

Describe application(s) that we're targeting to help frame the rest of the document.

From a Magneto perspective, this is just the list of applications we'll be using to guide and test Radix's design and implementation.

However, if there is a application type that Ericsson is targeting, this would be an opportunity to look at how we can design Radix to assist that type of application, adding value over things like stock Kubernetes and AWS. (This is related to "why should I care about radix" above) Ericsson has recently said publicly that they're shifting to target Telco applications, and not general purpose computing. Should we run with this idea?

Even within a given app subdomain, there's the difference between old applications and new ones or partial rewrites. How to port old apps to the Radix system with minimal changes. What do new applications gain from deeper integration with Radix services?

2.1. Cloudhelm

CloudHelm is a simulation system that generates models of cloud datacenters with Ericsson HDS8000 hardware components, geographical placement, and models of IT cloud customer workloads. Models created are in JSON and PostgreSQL formats. Analytics can be performed by connecting Tableau to a PostgreSQL model. Datacenter racks can be visualized with SVG graphics.

Tables from the CloudHelm data model for a particular set of data centers have been converted into multiple, simple/experimental read-only REST based endpoints with BoltDB. The topology of UUID and NUID primary and foreign keys allows one to construct workflow with Tomaton across these endpoints.

3. Threshold! Threshold! Take us to the threshold!

You want to build and run a distributed application. So, how is Radix involved?

- from one viewpoint, an application is a number of containers, each running application-specific code. Radix offers software and tools, called ***crux***, to help that code work together.
- from another viewpoint, an application needs to manage those containers and their lifecycles. The underlying infrastructure (upon which the application containers are running) also needs to talk to the application. Conceptually, both these functions are performed by the ***envoy***. Although we'll talk about the envoy as a distinct entity, it could

implemented in a number of ways, ranging from an integrated part of the application to an executable running in its own container. Thus, it is more useful to think of the envoy as an API, rather than a separate thing.
- Radix handles the variety of environments it runs on by defining the ***underverse***, which is an abstraction of the services those environments generally offer. In principle, a Radix user would need to provide their own implementation of the underverse. We intend to provide such implementations for some common infrastructures (such as Kubernetes).
- Radix will provide application-agnostic tools that help with issues of life-cycle management. These tools, together with the underverse interfaces and envoy interfaces, make up the ***threshold***. Note that the tools in the threshold are applications as well. They differ from regular applications only in that they register themselves in some catalog, and that they likely have unusual authorisations (that allow them to manipulate other applications etc).

We expand on these concepts below. But first, a note: it is commonplace to view this structure as a traditional layered architecture. But it really isn't. The threshold is really just a set of applications that act on the underverse and envoy APIs. How they implement their functionality is private. For example, one such service would be a load balancing scaler. There would be a generic version of this that uses just the underverse API. But for the Kubernetes infrastructure, we would likely use a special version that knows how to talk to the builtin Kubernetes scaling functions. This is not a violation of some layering principle; it is simply a different (presumably better) way to implement that functionality.

3.1. How much Radix with that application?

It is also possible to use differing amounts of Radix in building and running your application; it all depends on the details of your environment. We have found this difficult to explain abstractly, so we describe below a number of different scenarios. This is not an exhaustive list, but is meant to illustrate a possible way to handle a variety of situations.

3.1.1. Very little Radix please, I'm on a diet

You could use almost no Radix at all; for example, just use some of the Crux libraries. In this scenario:

- you would have to rely on some external mechanism, like myriad or Kubernetes or Swarm, to manage the containers within which you run your application.
- there is no instance of the fulcrum, no rnodes, and no cluster
- you construct your own service discovery, if needed (potentially using some Crux facilities)

3.1.2. Can I get that Radix on the side?

You've looked through the Crux documentation and decide you'd like to use the Crux heartbeating and service discovery stuff. And khan looks useful, too. So we construct our application, but how do we deploy it?

- we use Kubernetes (say) to start up 10 (say) pods, each running a fulcrum
- from the Crux horde code's point of view, we're now running a 10 rnode cluster, and we'll use the Ken protocol method for sharing data (because it's the simplest)
- inside one of the pods, we run a small program to initialise the khan spec for running our application
- once khan has started up all the parts, our application is now running
- if we want to dynamically change the size of the cluster, we need to perform some external Kubernetes magic and start up some more pods running a fulcrum. Once the new pods start up, khan will recognise the new rnodes and do any resizing automatically.

3.1.3. That smoked Radix looks good!

Dagnabbit! You've just been told you need to run your application in a few new situations, including a bunch of EC2 nodes, a set of VMs in the lab, and some brand new Linux boxes. Is there a way to avoid writing a harness for each of these environments? Luckily, yes! We could go full Radix. Let's start with baby steps; what does that mean for our Kubernetes environment?

- going the full Radix means that instead of manipulating Kubernetes directly, we need to manipulate our environment through the underverse API.
- we need to write an envoy for our application (there is an example below) that will request, configure and start a horde that covers a subset of those rnodes. These will then run the application as above, except that the envoy itself does the khan spec work. Note: your envoy will most often use fulcrums running Khan in order to start your application. (We should talk here about the benefits this gives you.)
- when we started Kubernetes, we started each kubelet running a fulcrum, and on one pod, start the underverse (Kubernetes flavor) service. This is Go code supplied as part of Radix. This daemon/service implements the underverse API.
- the underverse service starts the init service. This service is analogous to `/etc/init.d` and starts various inhabitants of the threshold, in particular, the *console* service.
- the console service is a normal Radix application with an envoy interface. Its primary purpose is to allow operators to "login" and perform various functions such as starting applications. There is nothing particularly special about the console service; it merely serves as a way to bootstrap the system (from Radix's point of view).
- someone attaches to the console service and authenticates. They instruct Radix to start our application.
- through either command line arguments or some system catalog, the console service determines that our application envoy is a specific container invocation and starts that container.

- our envoy then figures out what it needs and requests those resources from the underverse and then starts the various containers. This can be accomplished via the following actions:
    - The envoy requests a horde of fulcrum agents from the underverse API. (This is a horde specific to our application, which is run on top of the base-layer fulcrum agent which is already running on each rnode.)
    - The underverse rexecs these 2nd-level fulcrums on the requested number of base-level rnodes.
    - The envoy creates a khan spec and applies it to the khan that's running on our new application fulcrum, which will cause khan to start our app.
- at this point, our application is running.

That wasn't that hard, but what did all that work buy us? With little extra effort, we can do that same process for an EC2 setup, or even bare metal. As an example, we'll look at the bare metal case next.

3.1.4. Radix from scratch tastes just as good

As we saw above, once we use the underverse to do all our work, then there should be no changes needed in our code. So the differences for the bare metal case are invisible to the user and the app developer. But there will be a different dance in how the underverse is implemented (which makes sense because implementing this on bare metal will obviously be harder than implementing on top of Kubernetes (say)). So let's follow the action for a set of 8 (say) servers rebooting after a power cycle:

- each server comes up and runs a user program (perhaps via SystemD or somesuch) which is a fulcrum.
- these fulcrums aggregate into a **troop** (see below section on "Bootstrapping") with a leader.
- if the number of servers is large, then we might start up a Consul or similar to support cluster-wide shared data. We then start up an underverse server based on the capabilities of these low-level rnodes.

Note that in this case, we may have two independent sets of rnodes:

1. a low-level set that provides bootstrapping and other support for the underverse server
2. an application-specific set of rnodes to help manage activities within the horde for the application

And in fact, there might be the latter set for each application.

3.2. The underverse

Unhappily, the underverse API is potentially really large. Even if we focus on just containers, storage and networking, that is a lot. So we will start with a smallish subset that can support many applications, and gradually grow this as demand and resources dictate. (This is definitely

work in progress, and we know there will likely be some aspects of the API relating to security and integrity/provenance, but we simply don't know at this time.)

The details are given below, but the thumbnail sketch is that the primary entity is a set of containers called the *horde*. You incrementally build up the horde description by specifying details of what containers to use, what storage is needed and various networking properties. You then *realise* that specification which causes resources to be allocated and started up. This is typically done by Go code running in an application's envoy.

Some examples of the infrastructures underlying the underverse are:

- inside a single VM running on a laptop, we use *myriad* to allocate the needed containers and use the VM for persistent storage.
- given a running ***kubernetes*** installation, we use *kubectl* and a manifest to start the required containers.
- given a set of bare-iron servers or VMs, we might use Mesos to manage and allocate needed resources and docker-compose to start the required containers on those resources.

Note that although the API is uniform, there might be different implementations of the API for heterogenous environment.

3.3. The underverse API

The organising concept of the ***underverse*** is a set of containers (or other executable entities) called a ***horde***. The normal usage is to create a horde, add requirements to the horde, and then ask for it to be realised (allocated). The model assumes that the underverse will run these containers on entities called servers, and that these servers will have associated tags (like GPU) that indicate significant properties.

```
type UnderverseMsg interface {
      // TBD
}
type Horde struct {
      FromUnderverse chan <- UnderverseMsg
      ToUnderverse -> chan UnderverseMsg
}
NewHorde(name string) *Horde
(*Horde) Realise() error
```

`NewHorde()` creates a new horde with the given name. `Realise()` actually implements the horde as specified, and starts the specified containers. No guarantees are made about order or timeliness of the various activities involved. Note that the binding of storage to Execables is

done within Realise() and may yield errors (rather than when the storage was defined or requested).

```
type Execable struct {
	N int
	Name string
	ID string
	Image string
	Prog []string
	ReqTag []string
	AddTag []string
	Storage []Mount
}
(*Horde) SetExecables([]Execable)
(*Horde) GetRunning() []Execable
```

Set the executable entities for this horde. Each entity (for now, just container) specifies an image, a program to run in that image, and associated tags. The Name field is intended as a generic name, or role. The ID field will be set to a unique identifier for each Execable. The ReqTag field specifies tags required for the underlying server. (A tag starting with a "!" means not that tag.) The AddTag field specifies tags that will be added to the tags inherited from the underlying server. After a horde has been realised, `GetRunning()` returns information about what actually happened.

Generally speaking, the issues around persistent storage are nontrivial. For now, we want a realistic model with a limited set of implemented scenarios. There are three parts to this: someone (flock administrator) needs to define available storage resources, a user needs to request various types of storage, and a user needs to specify the binding of storage to container. We will distinguish between filesystem storage, called ***FileTree***, and extents of raw blocks, called ***Volume***.

```
type StorageTag string
ExposeFileTree(rnode string, dir string, bind string, tags
[]StorageTag)
ExposeVolume(rnode string, vol string, bind string, tags []StorageTag)
```

These calls will typically be interpreted by a plugin. The tags will be used later on to indicate placement restrictions.

```
type StorageID string
AllocFileTree(capMB int, properties string) StorageID
AllocVolume(capMB int, properties string) StorageID
```

These calls return the name of a storage request. That is, this declares an intent; any allocation will be done later when the horde is realised. The properties string is a collection of attributes like `"fast"` or `"robust"`.

```
type Mount struct {
      Storage StorageID
      Path string
}
```

This structure specifies where some storage is to be mounted in the container.

```
type PortConnect struct {
      ExternalIP IPAddrID
      ExternalPort int
      InternalPort int
}
type NetworkSpec struct {
      HordePrivate bool
      FlockPrivate bool
      HordeConnected bool
      ExternalConN int
      Ports []PortConnect
}
(*Horde) SetNetwork(NetworkSpec)
(*Horde) GetNetwork() (NetworkSpec, error)
```

Networking is, of course, unbearably complicated. And rightfully so, because networking reality multiplied by user demand generates a very large space of possible scenarios. We finesse all that for now by supporting a modest networking model described by the above `NetworkSpec`.

In our model, we have three sets of entities (each represented by an IP address) of interest: the ***flock*** which is a set of entities, our ***horde*** (which is a strict subset of the flock), and the ***internet*** (which are all the entities not in our flock). Normally, the flock would be all the entities in our underverse. HordePrivate, if true, means that only entities in our horde can make network connections to the entities in our horde. FlockPrivate means that only entities in our flock can make network connections to entities in our horde. HordeConnected means that all entities in our horde can connect to all entities in our horde. ExternalConN specifies the upper limit of the total number of connections from all entities in our horde to the internet. Traffic from anywhere (internet, flock or horde) to an allocated external IP and port will be routed to any entity in the horde listening for the specified internal port. (Currently, we do not specify the nature of that distribution, such as round-robin.)

In any case, the user can specify what they would like, but should check what they got by calling `GetNetwork()`. Currently, we expect that HordePrivate will be false, and FlockPrivate will be true.

Externally visible IP addresses need to be allocated.

```
type IPAddrID string
GetIPAddrID() (IPAddrID, error)
MapIPAddrID(IPAddrID) IPAddr
```

These can be used in a NetworkSpec as described above.

3.4. An example

We'd like to run an application that services requests coming into port 28735 on a single externally visible IP address. The app needs 3 containers, each running a slightly different command, but using the same image. The executable listens on port 45271. Each executable needs 200GB of robust storage at `/data` and 5GB of fast storage at `/usr/tmp`. Setting up this stuff is typically the work of the envoy, which executes in the threshold. The envoy's code would look like (we'll mostly ignore errors):

```
ip,_ := GetIPAddrID()
inPort := 28735
svcPort := 45271
svcPortS := fmt.Sprintf("%d", svcPort)
fmt.Printf("servicing requests on %s:%d\n", MapIPAddrID(ip).String(), inPort)
h := NewHorde("the_shire")
args := []string{"frodo", "samwise", "meriadoc"}
ex := make([]Execable, len(args))
for i := range args {
	ex[i] = Execable{Name:"hobbit", Image:"shireV2.3",
			Prog:[]string{"server", "-p", svcPortS, "-a", args[i]},
			Storage:[]Mount{
				Mount{AllocFileTree(200000, "robust"), "/data"},
				Mount(AllocFileTree(5000, "fast"), "/usr/tmp"},
			}
		}
}
h.SetExecables(ex)
h.SetNetwork(NetworkSpec{
		HordePrivate:true, FlockPrivate:false, HordeConnected:true,
		ExternalConN:0,
		Ports:[]PortConnect{
			PortConnect{ExternalIP:ip, ExternalPort:inPort,
InternalPort:svcPort},
		}
	})
if err := h.Realise(); err != nil {
	fmt.Printf("failed to realise horde: %s\n", err.Error())
	os.Exit(1)
}
fmt.Printf("horde %s up and running\n", h.Name)
```

Hope that was clear enough.

3.5. The envoy

As we've described above, traditionally, applications compute stuff. They occasionally request and release system resources, normally through something like POSIX system calls. But there is a different part of the application that involves setting up and maintaining those environments, and acting as a contact for when the underlying infrastructure (the underverse) needs to contact the application. This contact can range from notices that a container failed and another was started, to warnings about various service degradations or scheduled interruptions. Collectively, we call this part of the application the ***envoy***. We anticipate that it will likely be a separate program running in its own container, but in principle, it could simply be spread out throughout the application. The envoy concept is found in ***Macropod*** and is somewhat equivalent to ContainerPilot (see the description at Joyent). Note that Istio has a thing called an envoy, but it is more like a sidecar in Kubernetes, a network proxy based traffic shaper.

Because in general, the envoy needs intimate knowledge of the application, it needs to be written as part of the application. The division of work between the application and its envoy will likely vary for each application. We can illustrate this with two examples, each towards an end of the range of possibilities:

- a simple web farm app, which gets requests and returns the computed answer for those requests. All the envoy need do is obtained the required number of containers, arrange for the required binaries to be available in those containers, and then run the necessary binary in each container. In this case, the executing code is unaware it is part of a cluster; the envoy is responsible for the horizontal scaling and overall management.
- ***stix***, an S3 proxy application. Although stix has a complex internal structure, it is self-organizing given a set of rnodes sharing a K/V store. By default, *stix* will start up Consul for its K/V store. Thus, its envoy just has to obtain the required number of rnodes, run the *stix* agent (called ***fulcrum***) on each rnode, and on one rnode, run the *stix* startup program.

We will supply an envoy for generic web farm applications, which need very little setup and management other than scaling.

3.6. The threshold

It is exciting to think about the threshold and what entities it might support. Not just the envoys, but scaling load balancers, programs to manage various upgrade strategies and even automated application-agnostic performance tuners. It is also a natural place for programs to support operations, including controlling (start, stop, pause) applications, scheduling downtime, migrating applications.

There are, of course, pros and cons to this scheme:

- pro: this can run on a wide range of environments, from a datacenter with 100K bare metal servers to Docker running a single container.
- pro: the application environment is quite uniform.
- pro: the envoy documents what the application needs in order to run.
- con: you need an envoy (although for the common case, a generic envoy can be provided)

4. Bootstrapping

This is how Radix bootstraps up into existence. As rnodes restart, they do the following concurrently:

- rnodes organise themselves into flocks, each flock with a single leader
- rnodes discover what resources they have

After flock membership stabilises a little, the leader starts the underverse service described below.

Even as we describe what is going on in the context of a (massive, datacenter-wide) restart, keep in mind that this is an ongoing process, and will apply equally well when we powercycle 25% of an otherwise fully operational datacenter.

One of the difficulties in talking about networks/clusters as that there are multiple networks/clusters existing at the same time within a given set of rnodes. These networks are related in various ways, such as (disjoint or overlapping) subnets, or (incomplete or complete) overlays. Generally, there is enough overlap that it is confusing about which network we actually mean. So let's get to it.

At the lowest level, we have what most folks would call a *network*. Think of your everyday IP network. Rnodes have addresses, and we can send and receive packets of bytes to/from these addresses. For this discussion, we only need to talk UDP facilities (and not full TCP/IP sockets). That is, every rnode needs to be able to send and receive UDP packets on a single specified port.

At the next level, we have *flocks*. A flock has a namespace, a name, a leader, and a set of member rnodes. The namespace parameter allows multiple concurrent flocking on a single rnode, for example during concurrent independent integration tests, but will otherwise be ignored for the rest of this discussion. The leader is mainly to support a singular thread of control, but can also act as an authoritative reference on flock membership. (The details of how flocks work and elect leaders is documented in the code.) A rnode has a name (not used by the flocking process), a numeric vote, a flock name and a network address. The flock name is a dynamic self-configured string; it is not set per se, but rather it is a computed thing. The best

way to think about flocks is that it is a never-ending computation, with rnodes entering and leaving, that takes these inputs:

flock inputs:

Send(dest Address, []bytes)
Recv() []bytes
GetMyName(me *Address)
Admin(cmd AdminCmd)

These functions allow rnodes to send and receive packets, and to resolve naming/address issues. The admin commands are mostly testing-related (quit, restart), but one (mem_hint) is essential: it gives (with reasonable probability) the address of a rnode which could join our flock. The outputs of this flocking process?

flock outputs:

flock leader
flock membership

Again, flocks just give you sets of rnodes (and one leader per flock). The issue of discovery has been sidestepped, or rather, has been delegated to the only realistic source of that information (essentially the operator). For example, rnode IP addresses might be dynamically assigned from a DHCP service (or somesuch). That service would typically be given a range of addresses to allocate from. That range could be given to a few rnodes to seed the discovery process. In any case, this should only be needed the first time rnodes are made eligible to join the flock. After that, this process can be seeded from any recent checkpoint. This can be made more robust by periodically computing the addresses assigned by the DHCP service that are not in the flock membership list, and sending those to a few flock rnodes.

A few words about rnode names and addresses. Most "names" for a rnode, including addresses are more or less transient; IP addresses can be reassigned across reboots, and DNS names can renamed or restructured. Furthermore, a rnode can have multiple addresses (if it has multiple network interfaces). In order to more accurately track flock membership over time, rnodes assign themselves a unique (UUID style) name that is meant to be more or less permanent.

The properties of a flock change over time:

- flock membership changes as rnodes enter and leave
- flock leadership changes
- each rnode will offer various services (by starting a server for those services)
- each rnode will discover and then advertise what resources it has

The leader maintains a attribute of how long it has been leader; this allows various other services to be run after the flock has been stable for a while.

We have not yet fully defined what resources exactly means, but is meant to include CPU, memory and local storage. Likewise, we have not yet fully defined what services might be offered. But the combination of these resources and services should be enough stuff to

implement full-featured clustering software, such as MESOS or Macropod. This is defined to be a gRPC service point which responds to various RPCs (fully defined elsewhere) that describe, allocate and de-allocate rnode resources, copy files, and to describe, execute and kill processes on that rnode.

We anticipate that existing hordes will likely be restarted after flocks are stable. That is, a manager for a horde would figure out which rnodes were part of the horde, and restart any necessary horde processes (probably by running khan or some equivalent). There is an implied gating function, either by initial selection or in a dynamically growing situation, some kind of minimum standard and/or preconditioning.

In principle, an operator would need to restart a horde. But, as a practical matter, we anticipate the horde manager would leave checkpoints around, and that a horde manager would be restarted with one of those checkpoints and see if it is safe to continue on automatically.

4.1. Pesky details

There are a number of minor decision points and details to be chosen in order to nail down this scheme. They are

1. there needs to be a standardised scheme for authenticated data blobs. This is so various services can leave checkpoints around and securely recover using them (after reboots or somesuch). In olden days, this would be encrypted signed blobs. Perhaps we can get by with simple checksums (i'm unaware of a need for privacy) stored in a (block chain) ledger. These may be needed before we have networking set up, so maybe we need both schemes.
2. rnodes are presumed to be computational entities (like servers, VMs, containers) executing a fulcrum (a normal user-mode executable running on top of a normal OS).
3. by default, the fulcrum implements the flock protocol. Otherwise, the fulcrum supports a limited API served by a gRPC server on a (socket on a) designated TCP port.
4. troop behavior will be controlled by a field in the flock protocol packets. Is is on by default, but can be controlled through the flock admin interface. If "troop" is on, the flock leader will run a troop manager service on a designated port and all rnodes will run a troop resource manager that sends troop-related info to the troop manager.
5. if and only if there is a proper, recent troop checkpoint and the flock has the right number of rnodes, the troop manager will reinstate the troops from the checkpoint. The "right number" is a heuristic, perhaps within 20% of the number of rnodes from the checkpoint.
6. there needs to be an informational channel back to the operators (independent of logging etc). I had always assumed this would be a socket of some sort, but it might be simpler to make it an http web page.

4.2. A note on flocking and distributed data

Generally, a cluster (a specific group of nodes) requires a shared set of data. Intuitively, this would include, at least, cluster membership and a service registry. Mostly, this is done via something akin to a Consul or Zookeeper K/V store. But these stores are often a burden to set up and operate, especially if you need to do it per application. Radix supplies at least one method (and plans to support a second method) that can be used instead.

Based on the Ken protocol [refs], this scheme supports eventually consistent shared data within a specified window. Basically, it trades some networking capacity for avoiding running an external K/V store.

Note that schemes like this can be used simultaneously; for example, the underlying container system (like Kubernetes) might use ***etcd***, while a specific application might use Ken to coordinate itself.

4.3. Bootstrapping in gears

In order for flocking and trooping to occur, we need the ability to launch multiple containers that can all communicate with each other. We would also expect that these containers will be spread across multiple operating system instances (multiple servers, multiple vms, etc). This is a simple-sounding request, but it becomes difficult due to the realities of configuring multi-host container networking.

Container networking is tricky to set up before you have a full set of cluster features available to use, because all current solutions all require a reasonably fault-tolerant key-value store to serve as a backend state database. Installing a KV store requires figuring out which nodes will be part of your cluster, establishing trust between those nodes, configuring the database software with information about your cluster, and configuring the network overlay management daemons. Attempting to automate this work results in something that resembles a Rube Goldberg mousetrap (e.g. the Macropod pouch and ACPs for Kubernetes).

We find it helpful to think of this bootstrapping process as occurring in “gears”, with the metaphor of going from 0 to 60 in an automobile.

- Gear 1: Results in a set of hosts running some OS. May include iPXE booting, DHCP requests, and other such work. Alternately, may only require an API request to a service that provides VMs.
- Gear 2: Results in the ability to start containers that can communicate with each other, across any of the hosts. May include building a key value store, setting up network overlays, establishing trust, and other such work. Alternately, may only require an API request to a service that provides containers.

- Gear 3: Results in the availability of any other features that are required to support the target app but are not provided by gear 2, such as container orchestration, networked filesystems, and so on. May include Kubernetes, Mesos, and any other products that can be found on dodgy GitHub sites. Alternately, this may not be necessary at all.
- Gear 4 and beyond: Results in the target application running. For Radix, this includes flocking and trooping across containers, and other steps, as needed.

The bootstrapping process can be tested beginning at any gear. For example, one can start from relatively clean VMs (gear 1), and test gear 2 through any other gear. The major benefit of this kind of test is that it can be accomplished in much less time, improving development velocity. The major caveats are:

1. You can never be entirely sure that you started from a “clean enough” state, and
2. It becomes too tempting to stop testing lower gears, and they break without you knowing about it.

I currently propose to use Docker Swarm (the post-1.12 “Swarm mode” version) to provide the majority of the features of gear 2. I will be testing it in the lab and reporting back in this space with the details.

4.4. Gear 2 details

The goal is to select manager nodes and turn them into a quorum.

> [Note: It's tempting to do the manager selection automatically, but in reality, cluster managers are going to want to decide which nodes should be the managers, because it’s important that they be distributed across various failure domains (e.g. geographical locations, power circuits, network hardware, etc.)]

Docker Swarm bootstrap (run as a program, on all hosts):

1. pull in the most basic of parameters from some sort of scratch pad
    a. address of a "primary manager"
    b. address of "secondary managers" if we want that
    c. the swarm join token (this gets generated and uploaded to the scratch pad by the manager when performing the swarm init… if it's not there, keep trying)
2. (if we are the manager) run "docker swarm init" and specify the advertise address
3. (if we are the manager) push the swarm join token back up to the scratch pad
4. pull down the swarm join token and join the swarm
5. (secondary managers) promote self after joining swarm

4.4.1. Questions and answers:

How do we pass the swarm bootstrap program to the OS? Can we do this in some way that is roughly equivalent no matter what provided the gear 1 services?

Answer:

- One way to do this is to provide a script which runs after the first boot, in any interpreted language, as long as the image we're booting contains the interpreter. (Bash and Python are safe choices nowadays.) That script could of course run a daemon, service, etc.
    - When using Amazon EC2 or Openstack, we can use the "user-data" feature to accomplish this.
    - When performing a bare metal kickstart, a facility to run something on first boot isn't provided by all distros...however, we can use the "%post" directive in the kickstart file to run things in a chroot before the first boot.
- Our script can pull down and install a package which includes a systemd service for our bootstrapper, for example.

How will the swarm bootstrap script work?

Answer:

Pre-requisites:

1. Run a KV store to be used as a scratchpad for the configuration data that is generated at runtime. It must be accessible by all nodes in the swarm.
    a. Create keys (/swarm/<swarm name>/manager/*) which contain the network addresses of the manager nodes we want to use for swarm management communications. (These should persist after reboot and not be ephemeral addresses. It could be AWS elastic IPs, or IPs that we tie to our nodes' mac addresses...)
2. On each node, download the swarm bootstrapper package and install it, which should include the bootstrapper program and a systemd unit file which enables it to run on boot. (This should be easy to accomplish in the kickstart file %post section or the user-data script.)
3. On each node, pass configuration parameters to the swarm bootstrapper (this configuration can be in the kickstart file or in the user-data script, and can be written to a configuration file in /etc or similar):
    a. The address of the scratchpad KV store.
    b. A swarm name which we use to avoid clashing with other booting swarms that use the same scratchpad (this name needs to be the same for all swarm nodes).

The bootstrapper program:

(runs continuously in a loop)

- Get list of managers from etcd scratchpad. (all keys under /swarm/<swarm name>/manager/)
- If we have a manager address defined on an interface:
    - If we are currently manager of a swarm,

            - If there isn't a swarm worker join token in etcd,
                - Push a swarm worker join token to etcd.
            - If there isn't a swarm manager join token in etcd,
                - Push a swarm manager join token to etcd.
        - If we aren't currently in a swarm,
            - If the etcd scratchpad contains a swarm manager join token,
                - Join the swarm as a manager.
            - Else,
                - Get a "swarm init" lock from etcd (ttl=0). If another manager is already holding the lock, they got here first, so take no action.
                - Run "docker swarm init" and specify our manager IP as the advertise address.
                - If we couldn't init the swarm for some reason, release the swarm init lock. If we successfully initialized the swarm, keep the lock.
- If we don't have a manager address defined on an interface:
    - If we aren't currently in a swarm:
        - If the etcd scratchpad contains a swarm worker join token:
            - Join the swarm as a worker.

Note: the bootstrapper code assumes that IPs are configured and the network is up before it pulls manager IPs from etcd. If you run it, and THEN configure your IPs on some of your manager nodes, the result is that your secondary managers will become workers instead of secondary managers and you'll have to manually promote them from a manager node after they've joined. (Not the end of the world.)

What is the process to resume after a catastrophic situation like an all-node restart? We probably can't re-init the swarm in the same way, so some things will be different…
Answer: When all nodes in a swarm are restarted, the swarm does come back, as long a quorum of masters comes back. (If you're missing 2 out of 3 masters, the swarm stops working, although services don't stop.)

How do we upgrade this thing? What does the "docker swarm update" command do? ...and other production concerns

4.5. Swarmer - what it does, and why

There is now a tool in shed called "swarmer". Swarmer bootstraps a Docker Swarm cluster using an etcd scratchpad. It is intended to safely bring up a Docker Swarm cluster without manual intervention.

It uses an Etcd database (and the new etcdv3 api) for all of its cluster coordination tasks. These

tasks include pulling any centrally-stored configuration, ensuring that only one manager at a time attempts to init the swarm, and passing the secure join tokens to the nodes which haven't joined yet.

The swarmer daemons on each node must be provided two pieces of configuration: 1. The endpoints of the etcd cluster that will be used to pass around swarm info (this is, naturally, the only piece of configuration that can’t be pulled down from etcd), and 2. The network addresses of the nodes which we want to use as swarm managers. Nodes which aren’t part of a swarm do a search for the manager network addresses on their local interfaces, and nodes which have a manager address do a leader election with each other to figure out which node will be allowed to init the swarm. The winner inits the swarm and uploads the join tokens to etcd, and all other nodes pull the join tokens down and join the swarm.

Once a swarm is formed, the swarmer can be safely turned off, or started again, and the running swarm will not be modified. This also means that the swarmer will not cause nodes which are already part of a swarm to leave their swarm and join a new one. The --force flag can be used to change this behavior and leave any pre-existing swarm automatically when swarmer is started, although it is not recommended to use this on a production cluster if you care about your current swarm.

5. Transformations: changing the world

As we've seen, there is a substantial effort involved in constructing and deploying Radix and the applications that run on top of Radix. And yet, that is not the hardest part. This section deals with how we can manage change, both in Radix itself and in applications; the most common examples are adapting to demand (for example, scaling) and upgrading an application.

5.1. Scaling and topology

In the beginning, distributed applications were deployed explicitly. That is, individual processes were deployed on named servers. This is quite fragile (although it is not uncommon to see analogous schemes for supercomputers where the structure of the computation mirrors a physical layout), and so applications moved to schemes with a large number of identical servers chewing their way through a large work queue. These days, it is expected that the platform supplies a scaling load balancer. This is just a special case solution to a more general problem: given a structural description of the application's processing, how do we apply more resources so as to reduce some metric (such as *makespan*, which is the overall runtime)?

The simplest case is the so-called "web farm" case, where there is a single source of pieces of work, and a pool of servers which perform that work, piece at a time. The typical control is the number of processes (hence "scaling"), and the work is assigned such that the makespan is minimised (which implies that each server is equally loaded, hence "load balanced").

The general case is where the application architecture has a number of sizing parameters, and varying any particular parameter effects some externally visible metric, such as memory usage, or processing speed. You can then apply a generic, application-agnostic, process, such as ***claviger***, to systematically optimise these parameters for a given metric (or objective function). The advantages of such a scheme are manifest, but probably the most important one is automatically adapting to changes in workload, both short term (time of day or week, or special event) and long term (increasing number of users).

5.1.1. Claviger

Claviger is a system for managing an application's performance while it is executing. From a control point of view, the application (or rather, its envoy) defines various inputs, outputs and an objective function. Claviger will optimise that objective function, subject to specified constraints, by varying the inputs and observing the outputs. It will do this autonomously and continuously, adapting to both ordinary and unusual changes in the underlying input data. This management is independent of other aspects of the application, such as lifecycle issues such as deployment and upgrading, or even application development.

The heart of claviger is building and maintaining a quantitative model of the application's performance. Generally, this is quite difficult, but we expect to do it because both the inputs and outputs will mostly cast in the application's terms, and not in the platform's terms.

Taking a step back, claviger is part of an evolutionary view of (increasingly large and distributed) applications, where application development, deployment and operations are frequent and complex activities that need to be automated as much as possible. From this viewpoint, the relationship between application and platform has changed from an executable running on top of a runtime operating system to large distributed systems, built on top of an array of cooperating servers under application control and services running in the threshold. In order to do this, we need to define, at least partially, the application's *control interface*. (There will likely be other parts to this interface, such as deployment and migration issues.)

5.2. Upgrading

We anticipate a suite of tools to help manage different kinds of upgrades. In general, of course, an upgrade is quite service-idiosyncratic, and the only thing to do is to specify how to invoke the upgrade. But there seem a few common situations which are worth handling:

- red/green transitions, where an operator designates a percentage of traffic diverted to the new (green) version. (Presumably stepping through a sequence of steps, say 1, 5, 20, 50, 100.)

- simple sharp transition: start the new server (without registering), stop the old server from reregistering itself, register the new server, verify the new server, direct the old server to drain then exit.
- transformed data: drain/stop old server, transform the service's data (via some begat invocation), and then start the new server.

The issues around upgrading at a more fine-grained level are discussed above in the Services section, but basically, a new version of a service simply announces itself, overlaying the previously advertised version.

Our value add might be a driver for red/green using istio's traffic routing stuff.

5.2.1. A gedanken experiment: upgrading the fulcrum

This is an experiment to see if the scheme described above can handle the case of upgrading the fulcrum. In order to do that, we will construct a plausible design for how the fulcrum is set up; but the details shouldn't matter much, so concentrate on the big picture.

When the fulcrum is started, arguments specify its rnode name and IP address. It starts up some services using ephemeral ports:

rnodename/`fulcrum`	*ip*/port1
rnodename/`rexec`	*ip*/port2
rnodename/`fileio`	*ip*/port3

These services are registered on the low level cluster directory (the one propagated via the Ken protocol). The `fulcrum` service handles requests such as resource allocation/management and fulcrum control functions. The `rexec` service performs remote executions on this rnode. The `fileio` service deals with copying files to/from the rnode.

We have enough to update the fulcrum now:

1. copy the new `fulcrum` executable (with checksum cs1) to the rnode, say `/tmp/WithAttribsLevel` (presumably using the `fileio` service)
2. remember the address for the `fulcrum` service (ip/port1)
3. request the fulcrum service to pause changing resources for several minutes and checkpoint resource info to disk, say /tmp/yy
4. execute (via the `rexec` service) the commands
    a. install -sum cs1 /tmp/WithAttribsLevel /usr/bin/fulcrum # or wherever it lives
    b. /usr/bin/fulcrum -rnode rnodename -ip ip -resource /tmp/yy

5. verify the new fulcrum is up by asking the `fulcrum` for its version. Repeat waiting and retrying until the new one is up. If it doesn't come up, then alert the operator and repeat the whole process.
6. the new version is up and running, and there will be new addresses registered for each of the services in the table above (ephemeral ports and all). For now, we actually have two fulcrums running! But the old one isn't changing any resources, and any new use of the fulcrum will hit the registered addresses (which are the new ones). We then send a request to the old address (we remembered above) to quit. (We had to remember the address because there is no record of it in the directory any more.)

This assumes that the new fulcrum is backward compatible with the old fulcrum. If the new version isn't backward compatible, then we only need a couple of changes to the above process:

1. in order to avoid the problem of coordinating multiple fulcrums managing resources, upgrade the current fulcrum to simply act as a proxy for resource management if the service `fulcrum2` exists.
2. perform steps 1, 3-5 above (changing `fulcrum` to `fulcrum2`)
3. do not send a quit to the old service (we're keeping it running). If the new service points have new names (e.g. `rexec2`), then they will register with the new name. If they don't (say `fileio`), then the new server will supplant the old one (but that's okay because it's backward compatible).

Q.E.D.

5.3. Dust to dust

talk about beginnings (deployment) and ends (shutdown); especially for testing. we need to cover the case of flushing queues and issuing metrics etc after said flushing.

5.3.1. Begat

This seems to imply that *begat* should be extended to manipulate properties (as well the standard artifacts which roughly correspond to things with checksums). The canonical example is whether a (rnode,user) pair has the correct permissions/authorisation etc to access a specific resource.

6. I know he can get the job, but can he do the job?

Assuming you have built the various parts that make up your application, how do you manage and coordinate those parts? This questions applies to both offline computations, which normally are called batch jobs, and online computations.

For batch jobs, the main issues are efficiency/scheduling (minimising the makespan) and sequencing. For online work, the main issues are placement/discovery/configuration and adapting to changing resources in real time. For both types of work, there is the significant question of how to do with errors. Radix does not address all of these issues; some of them are mostly in the application domain. But Radix does have some tools that can help with many of these issues, and these tools are described below. Note that you may need to restructure and reorganise your application to make full use of these tools and techniques.

A note on nomenclature: recent trends in distributed computing are blurring the meaning of the term **scheduling**. For many years, this has concerned how to optimally sequence a set of independent pieces of work across a set of workers; the most common examples are job scheduling (for batch jobs) and process scheduling (within operating systems). This is the way Radix uses the term. More recently, some communities use the term to mean where to run a relatively long-lasting activity. For example, choosing a physical server on which to run a VM, or choosing which VM to run a particular container or service. Radix calls this use **placement** because it it primarily concerns where to do something, rather than in what order. In this sense, the Kubernetes scheduler is misnamed, because its function is placement, not scheduling. And conversely, you can't use the Kubernetes "scheduler" to schedule jobs (you can but you don't want to).

6.1. Caution: bad things happen to good code

One of the most pervasive ways in which distribution affects application design is how to deal with errors. This is not simply the issue that the code that detects an error might be running on a different process, rnode or server than the code that has to recover or otherwise deal with that error. It is more that we don't know what exactly happened:

- did the request to do something actually make it to the destination?
- did the request succeed but the reply got lost?
- did the request and reply succeed but the actual work failed?
- if a request or reply got lost, was this a transient failure for this pair of sender and receiver, or is it indicative of a bigger failure, such as a networking partition?

These issues have no simple answers, but it is clear that trying to deal with these sorts of errors can make normal code difficult to understand debug, and prone to coding errors. There is some promising research about a better way to deal with local errors, represented by the ***tomatom*** tool described below.

Larger scale errors, such as network partitioning, are more problematic. The best answer seems to be that in the infrastructure can detect such problems, it should let the application know. In Radix, this is best done by sending such information to the application's envoy, and let the application decide on how best to deal with it.

6.2. Tomaton

Tomaton (described here https://github.com/erixzone/cloudhelm/tomaton/doc/tomaton.md) is a reversible workflow engine with events and an event-routing hub, that offers workflow state-saving serialization that can be used for checkpoint/restart, or do-overs. Workflow is constructed in Go by attaching user-functions in the forward and reverse directions, so that the workflow engine can, on error, or data invalidation, step backwards, undoing any partially completed steps. So there are forward and reverse DAGs in Tomaton, and the goal is moveable. Tomaton workers run a NeSDa (Nondeterministic Events, State, Deterministic Actions) event loop, which is a pattern found in many other implementations (Ross Simulator, Ken Protocol, Intelligent Agents, Stutsman Rules-Based DCFTs). The current implementation can be used by services that have to run steps through multiple endpoints, or in user-based applications. Feedback data from workers can be routed to a function which can integrate real-time data from thousands of concurrent workers and use that information to control them. Tomaton workers can spawn the fan-out of child workers, which return events with results to the parent worker.

6.3. Begat

Begat (described elsewhere, but not here and not succinctly) is akin to the well known tool *make*. There is a description of work to be done, divided into sections called *dictums*. Each dictum describes what things (normally files) it needs, what it produces, and what to execute to produce those outputs. This is similar to make; the real difference is what we do with those dictums. Begat's model is that in principle, we always execute all of the dictums (the order of execution is implied by a dictum with a given file as an input needs to execute after a dictum producing that file as an output). This would be unusably slow except that begat records all dictum executions, and can utilise previous executions as a cached result.

For this to work, begat characterises everything in its world by a checksum and assumes that identical dictums, with identical inputs and executing in identical containers, will produce identical outputs. It can thus make strong assertions about exactly how and why any particular output was made and under what conditions. Therefore, begat can make provable claims about provenance of specific artifacts.

6.4. Alembic

Alembic is a new tool that addresses how to connect different parts of a distributed application. It models such applications as a directed graph of data flowing (over the edges) between servers (the rnodes) whose API is captured as a gRPC spec. Because of the gRPC spec, alembic knows exactly what data structures are moving and can mechanically

- generate the code for the server (using user-supplied routines for processing specific structures)
- provide data-specific ways of tapping dataflows (for debugging or publishing/distributing to other processes) on command
- generate real time telemetry and statistical data about specific dataflows

While this functionality is useful, the most important point is that by capturing this deep information of how an application works (in terms of dataflow), we can enable tools we can barely imagine right now.

6.5. Let's talk about talking: the communications fabric

We are looking into using Istio (istio.io) as a rich networking model to connect parts of the application. Istio is an open platform for providing a uniform way to integrate microservices, manage traffic flow across microservices, enforce policies and aggregate telemetry data. Istio's control plane provides an abstraction layer over the underlying cluster management platform, such as Kubernetes, Mesos, etc. Istio is composed of these components:

- Envoy - Sidecar proxies per microservice to handle ingress/egress traffic between services in the cluster and from a service to external services. The proxies form a *secure microservice mesh* providing a rich set of functions like discovery, rich layer-7 routing, circuit breakers, policy enforcement and telemetry recording/reporting functions.
- The service mesh is not an overlay network. It simplifies and enhances how microservices in an application talk to each other over the network provided by the underlying platform.
- Mixer - Central component that is leveraged by the proxies and microservices to enforce policies such as ACLs, rate limits, quotas, authentication, request tracing and telemetry collection.
- Pilot - A component responsible for configuring the proxies at runtime.
- Galley - A component responsible for storing and distributing Istio configuration state.
- Broker - A component implementing the open service broker API for Istio-based services.

Istio currently only supports the Kubernetes platform, although support is planned for additional platforms such as Cloud Foundry, and Mesos in the near future.

Note that this is a different way (than Alembic) of implementing the service mesh; it seems better for packaging existing applications.

6.6. scheduling/orchestration

The primary scheduling tool in Radix is begat, although begat can report the relationships between the dictums in its work and other (third party?) tools can build on that information.

Orchestration is a more vague term that blends scheduling, placement and temporal-based inition of activities. Radix addresses this in a couple of ways:

- khan (described below), which is focussed on placement, has some rudimentary orchestration capabilities, such as "start this class of processes after at least 50% of some other class of processes have said they are ready".
- conceptually, programs that do this should run in the threshold, where they can interact with applications (or more exactly, their envoys), as well as other programs like scaling engines and load balancers.

7. Hermetic: constructing containers

https://github.com/erixzone/hermetic

8. Crux: helping to code an application

crux is a bunch of basic utilities etc. like logging! but there's other stuff too!

If one were prone to depressive introspection, this seems like a pathetic cry of "look at us! we've done stuff" to people who just don't care. But that's not us, right?

8.1. Horde

the majesty of hordes

8.2. crux lib

the convenience of useful stuff

8.3. KV

everyone likes a K/V store

8.4. Khan

where to put stuff. not sure if this should go here or under scaling section.

8.5. Logging: I am the walrus

8.5.1. The overall model for logging

- generate a message (adorned with fields and tags)
- accrete a sample of these into an in-memory buffer (if buffering enabled)
- upon some mechanism, dump some part of this buffer to the transport
- the transport is either
    - a file
    - syslog
    - socket

8.5.2. Generate log messages

This api supports structured logging like that of logrus. Initial implementations are based on logrus but have added support for a simpler log statement syntax with added “tagging”.
Log statements can specify an additional Tag(tagName) with Log Level to further limit when the statement will output. This is useful for instrumenting uncommon or outputting additional high volume information useful for in-depth debugging information.

Logrus Example:

```
log.WithFields(fields).Errorf(“An error msg”)
```

Walrus Example:

```
elog := walrus.New()
elog.Log(walrus.ErrorLevel,nil,fields,”An error msg”)

// Example with tag control
// Providing a Tag parameter means a statement will not log unless the
// tag specific log level is correct.  The default log level does not apply.
tTestTag1 := "testtag1"
elog.SetTagLevel(tTestTag1, walrus.DebugLevel)
elog.Log(walrus.DebugLevel, tTestTag1, nil, "Displayed only if debug set for tag")

// OR….
cclog := elog.WithAttribs(someTags, walrus.Fields{"food": "rambutan"})
cclog.Debugf("debugf with pre-set fields and tags")
```

```
// OR...
ccclog := elog.WithAttribsLevel(walrus.DebugLevel, someTags, nil)
ccclog.Log("Logging a message depending on tags. No fields.")
ccclog.Debugf(“Logging a message)
```

8.5.3. Go logger compatibility

In addition to the above listed methods, we would also have the go log methods for compatibility.
https://golang.org/pkg/log/

This work may wait until needed.
Note: Any replacement of logging in an existing package would require editing the import statement to use our logger.

8.5.4. Remotely configurable logging

Updating a logger’s configuration via incoming network requests will require a go-routine to run in the background. Updates and logging actions may be happening concurrently, so updates must be safe.

Also, since multiple packages will be used in most binaries, we need to design our packages to take a logger for those packages’ logging output to be controlled by the remote calls.

This could be a package level init, to seed the logger to be used, or it could be a logger argument to “New()” functions for a package’s major components. It’s likely that a package level setter is the better choice, with a default logger used if none set.

<TBD>

8.5.5. Retrograde logging & Buffer control

Retrograde logging allows us to output previous log statements that were not within the current logging level, when a logging statement at the retrograde trigger level occurs. For instance, logging is configured conservatively at “ErrorLevel”, but when an Error does occur, N earlier entries are also output, even if Info or Debug.

```
elog := walrus.New()
// On a error output, output the previous N unlogged entries, including tagged entries
elog.EnableRetroLog(5, walrus.ErrorLevel, true)
```

```
elog.SetLevel(walrus.ErrorLevel)
fmt.Print("=== Start of logging\n")
elog.Info("1 Info log level ")
elog.Warn("2 Warn log level ")
elog.Debug("3 Debug log level ")
elog.Info("4 Info log level ")
elog.Tag("taggy-tag").Warn("5 Warn log level, TAGGED ENTRY") // is buffered
elog.Warn("6 Warn log level ")

fmt.Print("=== Trigger Now:\n")
// Shouldn't see any log messages output before the Print statement.
elog.Error("Trigger 1") // is buffered
elog.Warn("7  Warn log level ")
elog.Error("Trigger 2") // is buffered

fmt.Print("=== Test ring buffer dump\n")
elog.DumpRetroLog()
elog.DisableRetroLog()
elog.DumpRetroLog()
```

9. Appendices

9.1. The underverse

9.1.1. Background

We expect to implement Radix on a variety of infrastructures. And in order to reduce the amount of effort to do that, we've defined the underverse as an abstraction of the features and properties of that infrastructure. Thus, it is useful to enumerate some number of the concerns which likely exist behind this abstraction. (Note: it is not intended to be a complete list, but just a first pass at the landscape.) We hope that this list would help us in the following cases:

1. Figuring out which of these concerns matter to us, and providing some way for their details to be represented in and communicated through the abstraction.
2. Using some subset of these as a feature wishlist when selecting and expanding the services that we choose to make up the underverse for Radix.
3. Improving our understanding in future scenarios where the underverse abstraction leaks, especially in production.
4. Communicating to interested parties which things Radix depends upon, especially those things which we expect would be taken care of by other software which falls outside the scope of Radix.

9.1.2. Minimum requirements of the underverse

Radix expects an API to request the following, at minimum:

1. Multiple instances of containerized applications.
    a. In the case where multiple pieces of underlying server hardware (“servers”) are available, the ability to distribute our containers across multiple servers without having to individually place them.
    b. The ability to request container placement on servers that have certain hardware attributes (such as the presence of a GPU or a special network peripheral).
2. Persistent storage volumes to be mounted into our containers.
    a. The ability to remount these volumes onto other servers if our containers get moved for any reason.
    b. The ability for these storage volumes to be at least somewhat hardware fault-tolerant.
3. A minimum amount of networking support, including:
    a. The ability for all of our application containers to access each other using IPv4 unicast.
    b. A DNS service that includes up-to-date A records for our application containers.
    c. Outbound internet connectivity from our containers.
    d. External exposure of services, probably load-balanced across multiple instances of an application in a round-robin fashion.

9.1.3. Further potential requirements of the underverse

- Advanced networking
    - Broadcast or multicast support
    - IPv6 support
    - Network segmentation (VLAN, VXLAN, etc)
- Security features
- Storage features
    - Object store as well as block store
    - Ability to specify service levels/”quality” of storage (speed, reliability)
- Databases
    - KV store
    - SQL databases
    - NoSQL databases

9.1.4. Details abstracted away from us by the underverse

Unless otherwise specified, we care about installation, configuration changes, and upgrades of each of the items below.

- firmware on all hardware components... which doesn't need to be updated, until it does
    - power-related components (UPS, rack power units, etc)
    - network device firmware (not part of the server hardware)
    - network peripherals (ethernet cards, infiniband cards, etc)
    - storage peripherals (raid card, hard drives, etc)
    - motherboard BIOS
- operating systems on non-server components
    - network device OS
    - storage device OS
    - security device OS (firewalls, vpn, etc)
    - other fancy appliance OS (load balancers, etc)
- operating systems on "bare metal", including:
    - the bootloader we're using
    - kernel
    - Basic OS utilities, or "base image"
    - Other software packages that need to be installed on top
- network configuration on switches and routers and security devices
- basic server network configuration
    - IPv4 address(es) for each network interface
    - IPv6, too
    - DHCP
    - DNS
    - Multihoming (do you have connections to multiple networks? if so, what is the routing config?)
    - VLAN/VXLAN switch port configuration
- advanced server network configuration
    - host-based firewalling
    - do you need to run a routing protocol on your servers? (ospf/isis/bgp)
    - overlay networks if needed
    - VLAN trunking to each server, and adding/removing VLANs
    - VXLAN
- storage configuration and setup
    - configuration and mounting of directly-connected disks (RAID, zfs, etc.)
    - mounting any network-attached storage that needs to be mounted
    - Distributed storage solutions…glusterfs, iscsi, drbd, etc etc etc
- hypervisor setup per server (kvm, xen, vmware, etc… this potentially changes other things in this list)
- basic container runtime setup per server (docker, rkt, etc.)
- advanced container runtime setup/container runtime support
    - overlay network for containers (flannel/weave/etc)
    - dependencies for state-sharing across cluster nodes (usually a KV store)
- container manager/scheduler/orchestration setup (kubernetes, mesos, docker swarm, etc.)

    - might be containerized, or might be running directly on the OS instances
    - have to pick a "master" or multiple "masters" at this point in time.. everybody who starts up points at the master IP
    - usually also needs a KV store, possibly other dependencies as well
    - what do you do if you have too many nodes? have to break it into multiple instances of the container manager.
- container image repository
    - serves up images that the container runtime and container manager can pull onto the cluster rnodes
    - provide a way to load images into this
    - make sure the container runtime and container manager can see this

9.1.5. Events we care about which might be abstracted away by the underverse

- Server reboots
- Servers disappearing or becoming degraded
- Network partitions
- Planned maintenance
    - The ability to influence how maintenance is performed (e.g. rolling upgrades, staged upgrades)

9.2. Design questions for flock/troop cluster formation

Question:
What features do we enable by bootstrapping in this flock-to-troop manner, compared to a simpler method such as the one used by Kubernetes? (Kubernetes method: start a KV store, then start a leader or set of leaders and point them at the KV store, then start followers and point them at the leaders.)

- We don't have to decide upon a flock leader or troop manager in advance. (But we do have to have a complete set of the actual or potential IP addresses we want to scan for discovery purposes.)
- When a leader or manager dies, a new one takes over. (See below for questions about the specifics of this process.)

Response:
Flocking/trooping is parameterless (given the way networking is done in practice, everything will need some way of denoting where the IP addresses are) explain this better (and even allowing kub style registering as a means). It is adaptive in response to large-scale changes (like a network split). need to specify what the adaptation is. and its advantages over, say, k8s.

Question:

Why is "flocking" kept separate from "trooping"? I know that the general answer is "the troop provides more stability & resiliency than a flock", but what are the types of events that a troop hopes to smooth over?

Response:

the notion is that the troop is more stable and longer lasting than the underlying flock. for example, the troop might have a stable, user-supplied name (which flocks certainly don't have). There is troop-specific policy covering:

- how to handle slow small changes in the flock
- how to handle large scale changes in the flock
- how much is automated or deferred to an operator
- how to handle service and resource registration (for example, kv store versus central server)
- troops may use different transport techniques for heartbeats etc than the underlying flock (to reduce bandwidth constraints)

Question:

What information does a troop keep track of which a flock does not? What features does a troop provide to the operator which a flock does not?

Response:

The troop looks after supporting service registrations and publishing of resources. This might mean setting up a KV store and publishing its address, or setting up a registration server, or even having a hierarchical distribution process via a local protocol. The troop will have a more stable name (maybe even user-supplied). As per troop-policy, it might suspend service as well as emit warnings/advisories upon certain classes of changes.

Question:

Under what circumstances does a troop choose to cease functioning? Under what circumstances does a troop continue functioning, even though a flock would not?

Response:

Generally, there is always a troop as long as there is a flock, but this depends on troop policy.

Question:

How does a rnode leave a troop? How does a node leave a flock?

Response:

Generally, a rnode leaves a flock by timing out. A rnode leaves a troop by a similar process, but the troop timeout might have a different value. Consider the case of wanting to support two distinct MESOS's (MESOS is used here as a generic resource manager) on a single flock. There are two ways to do this:

1. add a filter to MESOS to make it ignore all but a certain set of rnodes
2. use the namespace parameter to split the flock up into two sets of rnodes, and then go back to a single MESOS per troop.

Question:

What is the earliest point in the bootstrapping process where a persistent store service is expected to exist and be accessible by others? What entity is responsible for starting and maintaining it? What happens to the store when that entity goes away?

Response:

We assume the fulcrum can recognise at least a "modest amount" of persistent local storage (where we hope it will find checkpoints etc). The resource discovery process (needed to figure out what the rnode will advertise to the troop) will know how to find, mount (or whatever) any other locally based persistent storage, and then as it registers services by which this can be accessed, it would start any necessary server processes. (There might not be any.)

Question:

How does an operator interact with a cluster which was bootstrapped in this way, and what does this interaction look like? How does the operator discover which rnode to query in order to get "dashboard" information?

Response:

The easiest way is to ask any rnode who its flock and flock leader is. That is two steps, but will get the answer. Of course, if there is a network partition, then you'd need to ask a rnode in each partition, but there is no way around that problem.

Question:

At what times will we decide to elect a new flock leader and a new troop manager? What is the process needed to elect a new flock leader or troop manager? (Are these one and the same rnode in all cases?) How does the new leader/manager get access to the former leader/manager's data? How do we keep troop services from shutting down and ensure that they maintain their presence in the service catalog (if applicable)?

Response:

The easiest method is for the troop leader (manager) to be the same rnode as the flock leader. The flock leader jumps around during the early stages of flock aggregation, but after a while, flock membership is stable and its just leader election. Flock leadership can always be upset by a new rnode coming online with a very high leader vote, but that can be mitigated by having the leader set its vote to a higher value than any other vote if it has been leader for a while. As all the flock and troop information is computable from per-rnode information, there is nothing to be transferred per se. If the troop leader does change, then some troop services might become wedged until the new service point publishes.

Rnodes will start the flocking process over if they haven’t heard from the flock leader in the heartbeat time (e.g. 5 seconds). Flock leaders are programmed to send 2 heartbeats during this time, so as long as one gets through, we’re good.

Question:

Is there any time where the cluster becomes more rigid and is less likely to change without operator intervention?

Response:

This could be a troop policy, but it would be a bad idea. The point is that flocks and troops should represent the truth on the ground.

Question:
Where do we imagine the bottlenecks will be in this system? How will we scale the leader functions horizontally when we need to (multiple flock leaders/troop managers)?
Response:
Bandwidth for heartbeats. But this can be avoided by using the Gupta,Liskov multi-stage ring architecture for heartbeating. Obviously, the bandwidth needed for resources can grow arbitrarily depending on how verbose those descriptions are.

Question:
How do we hope to deal with situations where some of our information is incorrect? (Example: resource contention on our flock leader causes it to believe that much of its flock has gone out to lunch, but the converse is the case: the leader has partially gone out to lunch.) How do we avoid leader flapping?
Response:
This is hard:

- in extremis, the leader will start missing heartbeating, in which case, the issue will resolve itself (or not, we might oscillate).
- a rnode could also monitor its own health (or ability to service its flock), and voluntarily elect to demote itself. (without changing the protocol, it could just lower its vote, and tell everyone; some other rnode will elect itself leader.)
- the leader would be sending dashboard info which could be extended to include, say, member rnode timeouts over the last minute (say). This would allow the operator to notice that a rnode can't seem to support a large flock and prevent that rnode from being leader.

Question:
Where exactly do we send the dashboard information?
Response:
Ya gotta do something. There are multiple answers, depending on what you are happy with:

- use an external registration service that maps queries into a URL (like Consul)
- run the initial fulcrum with a hardcoded answer
- Whatever floats your boat.

Question:
How do we do anything beyond starting up a troop?
Response:
Too many different answers possible here, but to give a specific answer, one could imagine imagine specially-named checkpoint files that the troop agent would advertise. The troop leader could then then run *begat* (or somesuch) process on those files, which would in turn cause files

to be copied and processes to be executed. Once the troop executed begat, subsequent activities are managed by the Radix infrastructure, and no longer are troop business.

This answer presumes that an operator starts the Radix service for the first time, or somehow seeds the rnodes with appropriate checkpoints. But after that, this should be an automatable no-touch process.

-

9.3. I heard you wanted all your containers to be able to talk to each other

9.3.1. Option 1: docker swarm (new style as of mid-2016)

https://docs.docker.com/engine/swarm/networking/
https://docs.docker.com/engine/swarm/

What are the bringup steps?

1. run "swarm init" (from any rnode) & point to a single manager IP
2. run "swarm join" from every worker
3. (optional) run "rnode promote" to designate failover managers
4. run "docker network create --driver overlay" to create a network
5. run "docker service create" and point the service to the network you made in step 4
   (Creates a Virtual IP and service name in swarm's DNS)

Benefits:

- supported directly by docker
- you can create multiple overlay networks
- you also get some orchestration-type features
- supposedly fast and scalable
- you get TLS security by default
- relatively easy to set up

Drawbacks:

- docker-specific (no rkt)
- configuration requires deciding on masters in advance (likely you'd want to do this even if it weren't required)
- setting up requires starting managers differently from workers
- cluster quorum has to be maintained (it uses raft consensus, and there's a database on disk that you have to stop the cluster in order to safely back up)

- services get "scheduled" onto the swarm and you have to use "constraints" in order to place them on specific rnodes... this may or may not be a drawback, depending on what you expect
- you have to run your own docker registry if you have custom images

9.3.2. Option 2: docker multi-host networking (old style swarm)

What are the bringup steps?

1. launch consul on at least one rnode
2. (optional) create more consul rnodes for failover
3. (re)start docker engine on each rnode and point it to the consul url
4. create network (same as above)
5. run containers that specify the network you created as --net

Benefits:

- supported directly by docker
- you can create multiple overlay networks

Drawbacks:

-

- docker-specific (no rkt)
- have to perform a docker start or restart on every rnode
- have to run a key-value store and keep it up
    - cluster quorum has to be maintained
    - have to decide in advance which rnodes should be part of this
- if you want TLS security, it gets way more complicated to start up
- have to specify the network you want every time you start a container
- this isn't the hot newness anymore, so docker might drop support for it

9.3.3. Option 3: rkt -> flannel -> docker

What are the bringup steps?

1. start etcd somewhere (either use a discovery url to make configuration the same across all rnodes, or manually configure each rnode)
2. on each rnode, run flannel as a rkt container, pointing it to etcd
3. on each rnode, run docker, consuming the flannel output file on disk

Benefits:

- flannel works with rkt as well as docker, so you aren't tied to docker
- default mode in coreos's container linux

Drawbacks:

- messiest bringup process... need to be able to run both rkt and docker, and have to start processes on every rnode

- have to run a key-value store and keep it up
- you only get one overlay network (but this is what kubernetes needs, so it's ok for that purpose)
- security can make this more complicated

9.4. Istio Service Mesh & Alembic

(DMB: I'm putting this here to kickstart a conversation. This overview should be extended or moved to a "things we didn't use" section.)

At least part of Istio is network proxies that seem to overlap with **Alembic's** purpose. Can we fuse these ideas? Especially the monitoring aspects.

> [[andrew:]] *I don't perceive any overlap with Alembic per se (insert apologies for being unclear enough about Alembic so that this confusion could arise). But Istio addresses a nearby issue that has been floating around which is how do you implement the plumbing associated with using micro-services? It is worthwhile walking through these issues so that folks can see for themselves what the problem terrain is.*
>
> *One area of "overlap" is that potentially, a number of the properties and features addresses by Istio could annotate a vertex-edge graph description of an application, and thus could be part of an Alembic description.*

With a service mesh, like Istio, there's another layer, and the added complexity. However, it does look like it helps with complicated cluster issues like secure connections, monitoring, etc. And can do so for older applications, not just bespoke microservice apps.

9.4.1. Istio

Google and IBM are pushing the Istio project as a standard for a "Service mesh". The mesh is composed of proxies for platform services. At least some features are provided silently by a given proxy (security, monitoring) and don't require code changes to apps. They list this as selling point, but it seems that at least some services would be more like REST calls to the proxy. More like sidecar wrappers. Needs more investigation.

The end result is less distributed system code in applications & servers and move it to service mesh layer. Also, operational tasks and coding are more separated from application logic.

I'm curious how much it actually helps reduce the certificate management burden.

https://github.com/istio/istio

> *Visit istio.io for in-depth information about using Istio.*

Istio is composed of three main components:

***Envoy** - Sidecar proxies per microservice to handle ingress/egress traffic between services in the cluster and from a service to external services. The proxies form a secure microservice mesh providing a rich set of functions like discovery, rich layer-7 routing, circuit breakers, policy enforcement and telemetry recording/reporting functions.*

Note: The service mesh is not an overlay network. It simplifies and enhances how microservices in an application talk to each other over the network provided by the underlying platform.

***Mixer** - Central component that is leveraged by the proxies and microservices to enforce policies such as ACLs, rate limits, quotas, authentication, request tracing and telemetry collection.*

***Manager** - A component responsible for configuring the Envoy and Mixer at runtime.*

Istio currently only supports the Kubernetes platform, although we plan support for additional platforms such as Cloud Foundry, and Mesos in the near future.

9.4.2. Misc

https://istio.io/
https://github.com/istio/istio

https://istio.io/talks/istio_talk_gluecon_2017.pdf
https://istio.io/docs/reference/release-roadmap.html
https://spiffe.io/

9.4.3. Sept 2016 Envoy

https://lyft.github.io/envoy/docs/index.html
https://lyft.github.io/envoy/docs/intro/comparison.html
https://lyft.github.io/envoy/docs/intro/deployment_types/deployment_types.html

https://eng.lyft.com/announcing-envoy-c-l7-proxy-and-communication-bus-92520b6c8191
(Abridged & edited list of features from link)

- *L3/L4 filter architecture: Core of Envoy connection handling. Simple API to add your own.*
- *HTTP L7 filter architecture*
- *HTTP L7 routing:*
- *GRPC support: -Envoy supports all of the HTTP/2 features required to be used as the routing and load balancing substrate for GRPC requests and responses.*
- *Service discovery:*

- *Health checking:*
- *Advanced load balancing:*
- *Front/edge proxy support*
- *Best in class observability:*

9.4.4. May 2017 Istio

https://istio.io/blog/istio-service-mesh-for-microservices.html

> *Istio adds traffic management to microservices and creates a basis for value-add capabilities like security, monitoring, routing, connectivity management and policy. The software is built using the battle-tested Envoy proxy from Lyft,*
>
> *….*
>
> *Developers are freed from having to bake solutions to distributed systems problems into their code. Istio further improves productivity by providing common functionality supporting A/B testing, canarying, and fault injection.*
>
> *...*
>
> *Istio enables operators to authenticate and secure all communication between services using a mutual TLS connection, without burdening the developer or the operator with cumbersome certificate management tasks. Our security framework is aligned with the emerging SPIFFE specification, and is based on similar systems that have been tested extensively inside Google.*

9.5. Netflix & Sidecars for Microservices

Overlaps with proxy based approach. Some insight on cases where sidecars (or proxies) can be helpful.

Netflix mainly uses their Prana sidecar to allow non-JVM services to use their java based infrastructure. Wrapping their JVM library calls in a REST interface. They still prefer to use native libraries for their JVM apps, so it's a targeted solution, not a religion at Netflix.

https://medium.com/netflix-techblog/prana-a-sidecar-for-your-netflix-paas-based-applications-and-services-258a5790a015

> *We use Prana with non JVM applications and also with any application which is not developed on the Netflix stack. For example, we run Prana alongside Memcached, Spark, Mesos servers to make them discoverable*

https://www.voxxed.com/blog/2015/01/use-container-sidecar-microservices/

Mentions various sidecar implementations

- Netflix's Prana
- Docker Ambassador Cross-container Linking

- AirBnB's SmartStack

9.6. Prior work

Software and Protocols with some overlap with Flock concepts.

9.6.1. MESOS

A scheduler / resource allocator.

9.6.2. Nomad (hashicorp)

For deploying applications. Has scheduler in addition to cluster management. But how does cluster management part compare to flock+troop?

- Single binary & simple config.
- Membership done with Serf library implementation of SWIM gossip protocol.
    - https://www.serf.io/docs/internals/gossip.html
    - https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf
- Distributed and fault tolerant via Leader election & state replication (Raft protocol)
- Task drivers work with containers, vm's, standalone applications

https://www.nomadproject.io/intro/index.html

9.6.3. Serf

https://www.serf.io/
CAP: Serf is AP vs Consul's CP

Hashicorp's go library for "Decentralized Cluster Membership, Failure Detection, and Orchestration." Used by consul.

Based on SWIM Gossip protocol with some speed improvements for propagation and convergence rates. Optional snapshots to local storage for faster re-joins on failure.
https://www.serf.io/docs/internals/gossip.html

All traffic is protected using AES-128 Symmetric key encryption. Initial key is distributed how?

Serf is rnode level granularity, while Consul (built on top of serf) has rnode and service concepts.

Custom Events can be used to implement, app specific processing.

It supports mDNS discovery if multicast is enabled. Else you supply at least one peer to the -join flag.

9.6.4. Apache YARN

https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
Provides resource management & job scheduling + monitoring. How does resource management compare to troop's resource management?

The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the RnodeManager(s) to execute and monitor the tasks. (Macropod / Envoy similar to ApplicationMaster?)

9.6.5. Terraform (hashicorp)

Infrastructure provisioning & management. Provision images.

- Config file used to define components needed for a single application in a datacenter.
- Infrastructure as code / abstracted infrastructure components
- Build servers, DNS entries, etc. Would then hand off to Configuration Management tools like ansible/salt/puppet for application level config
- HCL files define target state / resources to build.

https://www.nomadproject.io/intro/vs/terraform.html

For small infrastructures with only a handful of servers or applications, the complexity of Nomad may not outweigh simply using Terraform to statically assign applications to machines. At larger scales, Terraform should be used to provision capacity for Nomad, and Nomad used to manage scheduling applications to machines dynamically.

9.6.6. BOSH

??

9.7. Logging

Erixlog discussion and current implementation state:
https://docs.google.com/a/erixzone.net/document/d/134oUoPNT4a5oLee0B9oJEWRzfjXeOu77H8UexujrRco