Intent and purpose of this document

This document wants to answer two main questions:

1. What features do we want this product to have? (What is "in scope" for this product?)

2. What limitations of this product are "by design", or in other words, which lack of functionality have we agreed upon? (What is "out of scope" for this product?)

Along the way, we may also identify limitations of the code or design as they currently exist, which we think would keep us from implementing the desired features and functionality.

At the bottom, we include some activities we anticipate crux being used for, and the motivating "whys" underlying that.

Potentially illuminating questions

These questions are intended to act as “unit tests” for this document. If the document answers these questions, the tests pass.

- How does a user interact with a radix cluster in order to get work done? What is a user able to do, and what are they not allowed to do? How does this look from an API or user experience perspective? How do these abilities and interactions change over time with changes in the cluster state or lifecycle?
- What does the multi-tenancy model look like? Do we intend to support multiple users (i.e. administrative domains, i.e. “hordes”) per computational entity? If so, how are multiple hordes on the same CE separated from each other?
- Which resources that support a user’s application are allowed to be outside the control of radix, and which must be kept under radix control? What kinds of policy are we hoping to enforce, and how does this change what needs to be under our control?
- Which radix functionality will require data persistence? In what form should this persistence exist, and how will it be managed?
- Which resources does radix absolutely rely on, and in contrast, which resources should a radix cluster be able to do without for some period of time?

Terminology

- Computational entity - We consider our domain to be a set of *computational entities (CE)* networked together in a certain way. A computational entity is a unique set of resources (cpu, memory, local storage, networking) where we can execute our code. For example, a CE might be a bare-metal server, a container, or some type of virtual machine. We set out our requirements for each of these CEs below.

The functional expectations and limitations for radix:

- User application requirements and scope:
    - Functional requirements (what's in scope for this product?):
        - The ability to run a user application in various forms including as a local process, as a container, and as a VM. Do we want/need to support Go plugins as a user mechanism? (This requires more specific discussion about what is allowed and not allowed, and the methods we intend to provide in order to accomplish this.)
        - Application control of its own size, i.e. application expansion and contraction, i.e. autoscaling. (This may or may not be equivalent to cluster autoscaling, which is mentioned below. Whether or not it's equivalent is based on whether the cluster needs to change size in order to change the application's size, which is again the multi-tenancy issue.)
        - Some ability to request and change the resources that are assigned and available to the application, including resources that are provided to the application from someplace outside the cluster, i.e. "the underverse". (This requires more specificity regarding the nature of the resources and how they may be mutated. But this may well just be the number of computational units -- probably the units themselves are unmodifiable.)
        - Some ability to get events from the platform and to respond to those events. (The details of this, along with its limitations, should be discussed.)
        - The ability to manage application lifecycle events in an automated fashion, e.g. application bootstrap, fault recovery, and no-touch upgrades.
    - Intended limitations (what's outside the scope of this product?):
        - We do not intend to provide an abstracted API for all possible "underverse" resources. (What exactly we do intend to provide as a method of resource control requires further discussion.)
        - Others TBD…?
    - Unintended current limitations (based on how things work today, are there any limitations that could prevent us from providing some of the in-scope features?):
        - Any service run on radix must be in the form of a Go plugin, which means that starting a user application would need to be done using some sort of plugin-sidecar. (Note: if this is the intended final form of application management, move this to the "intended limitations" section above.)
            - If users are given the ability to run containers or VMs using a plugin-sidecar, the resulting processes would potentially end up outside the knowledge and control of the radix cluster. This has the potential to cause application manageability and security/policy limitations.

            - if a new service runs (for example) as a new process, what is crux's API to that process? (or is there one?).
- Cluster behavior:
    - Functional requirements:
        - a CE is a place where we can
            - run some code (*fulcrum*) supplied by Radix
                - (Potentially multiple? Or only one?)
            - update the fulcrum code
            - has symmetric networking access to other CEs in the cluster (outlined below)

          In our minds, CEs are bare metal servers, VMs or containers.
        - Multiple CEs must be able to form a single cluster with each other with zero human intervention.
        - Multiple fulcrums running in the same CE should be able to form a single cluster with each other. In other words, a minimum amount of resource separation between fulcrums should be required. (How much required resource separation would be acceptable?)
        - A CE should not depend on any specific containerization or virtualization feature. In other words, it should be possible to run radix without using containerization or virtualization at all. (Some discussion of the expected limitations when not using containers or VMs is in order.)
        - We anticipate multiple CEs will form together into a *group* (commonly, a cluster) in order to perform some (group) function.
        - We need to dynamically modify group properties (only the number of CEs in the group).
        - Some amount of fault tolerance, e.g. in the case of various types of network outage events. (Requires more exploration; the acceptable limitations to this are unknown.)
    - Intended limitations:
        - To be discussed.
    - Unintended current limitations:
        - Multiple fulcrums cannot run in the same execution environment without some level of network separation. Further details are in the “network” section below.
- Networking:
    - Functional requirements:
        - The ability to function normally (without human intervention), especially after power cycling and (most) networking failures/repairs.
        - The ability to survive common networking failures/repairs such as network splits.
        - The ability to access data stored outside the cluster; this includes remote storage (such as Amazon S3). Any local storage would be accessed through normal inter-node networking.

  - Intended limitations:
    - All instances of radix (fulcrums) must be able to contact each other as a full symmetric mesh; clusters cannot use border connectivity to form across a network boundary. (driven by forward secrecy.)
    - All fulcrums must be able to advertise to other radix instances a unique address at which they can be reached---invisible network address translation inside the cluster is not allowed.
  - Unintended current limitations:
    - Each fulcrum must have a unique IP address; fulcrums cannot form clusters with other fulcrums that use the same IP.
    - Each cluster must use the same UDP port for cluster formation purposes. (This is potentially true for other fulcrum-level services as well.)
- Storage and persistence:
  - Functional requirements:
    - Services need the ability to save and recall application state across reboots and/or restarts.
  - Intended limitations:
- Security requirements and features:
  - Functional requirements:
    - Encrypted communication between CEs.
    - Perfect forward secrecy for the session keys for the above transmissions.
    - Service access policy attributes for the GRPC interfaces.
    - intra-application security? does crux supply? need more thoughts here.
    - security aspect to applications implemented as (especially) Go plugins (which cannot be terminated or controlled).
  - Intended limitations:
    - To be discussed.
  - Unintended current limitations:
    - Communication with the beacon is a fly in the otherwise PKI-based security ointment.
    - Others, no doubt.

and the answer is (what)

One way to approach the above questions is to state the intended (high-level) answers in the form of high-level features. If we can agree on these, then that would force answers to the above.

Users would like to develop and run:

- distributed applications made up of parts running on different "nodes", with networking between these parts.

- application code should be able to refer to services in a generic way (independent of where the service is running) that survives restarts. (FIXME: naming that allows for connection to *specific* instances).
- users would like a mechanism to access their services from outside the cluster (e.g. as API requests or http/raw protocol access).
- all intra-application and application-service traffic should be encrypted.
- TODO: service configuration/parameters/secrets
- applications need to access persistent data in the form of S3 services, local filesystems, and network filesystems.
- the set of instances that an application runs on needs to be dynamically alterable through a crux API (changing the number of instances of the user's application).
- the parts of an application might be Go routines (plugins), discrete processes, containers, VMs. We're currently not sure what to do here.
- CEs have a fixed amount of resources; that is, if you want more resources, you need to procure more CEs. (Whether accepting and fulfilling these types of requests is something that would be in the scope of Crux in the future is still being debated.)
- a new application part/instance can be added and then stopped; it cannot be altered in situ.

Administrators would like:
- The ability to support multiple users on the same set of CEs (multi-tenancy).
- Some level of administrative control over the resources that a user is allowed to consume (needs further elaboration).
  - The ability to constrain the number of application parts running within a CE.
- The ability to run these clusters in various environments including bare-metal, containerized on kubernetes, using VMs on AWS, etc.

Different classes of users of this system:
- Administrators of crux
- The users of crux (“developers”)
- The users of the developers’ applications (“end customers”)

more to be done here.

Notes:

- how do we access a broken horde/cluster? logs etc.
- Is “the beacon” part of the long term design, or are we planning to design it out of existence? If so, what would replace it?

why-1

- Example from Martin: Cluster membership is a hard problem and it needs to be solved...Why and for whom?
- we'd like to be able to build crux on crux itself. which means running begat.