Project Mercury

To summarize (tl'dr), I've been working on a replacement for our primary applications: kserver, kclient, and warden. The replacement will be built with a few core tenants:

- Simplicity
- Ease of Administration
- Control
- Forget Nothing
- Log Everything

I've dubbed the application namespace mercury. Mercury as in the element; having the properties of liquid metal. I thought about T1000, but James Cameron would sue me for not promoting the next three Avatar movies.

I am intrinsically motivated by challenge and deadlines, so I have positioned myself to deliver this rewrite for the new OSIC (openstack innovation center / Andy) deployment. The full scope of my vision cannot be realized so quickly, but I fully intend to provide a subset of the services provided by mercury for consumption by RPC and Intel. My self imposed due date is September 2nd; when the first cabs roll into dfw3. As I am tired of using the word 'phase', each logical segment of the application stack will be described as halo's (ref: http://www.nin.wiki/Halo_numbers).

Halo 1: Liquid Metal

Before getting into the meat of the application architecture, I will cover a few core concepts. The first concept is that of <buzzword>micro services</buzzword>.

As part of the Simplicity and Ease of administration tenants, deploying a monolith web framework running on apache/wsgi is out of the question. There are 'micro frameworks' such as flask and golang which can be plopped behind load balancers, but this merely transfers the complexity to client code. Also, lets be realist, the web, despite what the average REST evangelist would like you to believe, was built around human interaction and is ill-suited for efficient machine to machine communication.

Conversely, building a socket level communication protocol into a deployment application certainly goes against the Simplicity tenant. As such, I've decided not to do that.

Enter ZeroMQ (required reading: http://zguide.zeromq.org/py:all). ZeroMQ, despite the MQ (and the

reason for the name), is actually a high-level socket library. It takes the complex bits of every distributed application (fault tolerance, queuing, async processing, etc) and hides them, exposing only a high-level interface for doing things you actually want to do. On the back end, mercury uses only ZeroMQ for inter-process/node communication. This greatly simplifies the implementation of client/server, pub/sub, and router/dealer constructs within the core application stack.

ZeroMQ does not provide a protocol for passing messages, however. Thus, mercury will use a standard serialization mechanism for all messages: msgpack (http://msgpack.org/).

Now for the fun stuff: architecture.

I've attached some very rough drafts of some of the design documents that I have created. Unlike, past endeavors, I am spending MUCH more time on design. So, these documents will change, but they should give a general idea of where things are going.

One more thing to note, all devices throughout all services and clients will have a single atomic reference. At present, we rely on MAC address or inventory ObjectID to relate devices in distributed systems. This is no longer the case. Mercury will provide a construct to generate a unique identifier for a device based on the following criteria:

- If the firmware for a hardware device presents an asset tag or serial number, we will create a unique hash based on these two identifiers.

- If the device, such as a VM (though mercury shouldn't me provisioning these things anyway) or non OEM chassis – we will generate the hash from all onboard MAC addresses.

The unique device ID will travel with the server throughout it's life cycle, wherever it goes. While it may not be apparent yet, this ID will allow us to do some pretty amazing things without having to lug around an entire inventory object to do it.

Lets get started:

Halo-1 will provide the following services:

- inventory
- rpc
- boot
- agent
- log
- cli

Inventory:

This service will provide an API for storing and querying data that has been collected by an agent running in a specialized in-memory linux environment. The inventory will contain hardware information for each device, such as, but not limited to, cpu, memory, pci, usb, hardware raid, obm capabilities, dmi, network interfaces, and location data (LLDP, CDP). As a departure from the current inventory implementation in kserver, the inventory will not contain any asset management or configuration data. It will simply be the source of truth for hardware inventory, nothing else.

RPC:

This service is illustrated in the mercury-rpc pdf attached to this email. It provides the central component for the Control tenant and is a complete departure from the kclient functional process. As illustrated, the service is dual-headed. On the back end, agents will register with the service. Once registered, a thread will be generated to 'ping' the registered agent to ensure it is alive. On the front end, the service will accept an inventory query, command, and arguments. It will use the inventory query to route the command and argument structure to active agents. The matching agents will perform the task in parallel and return any result to the rpc service. Some example commands: configure_raid, reboot, press_entry.

Boot:

Unfortunately, we can't get away from HTTP entirely. We still need the ability to control dumb agents such as iPXE. When a device boots, it will get a DHCP ip, bootloader, and script. This process will be vastly more simple than the kserver boot controller, providing only three options; boot from local storage, boot into the mercury agent OS, or chain another script from another location (boot.rackspace.com for example). This service will most be built using the flask micro framework, and at present, is an afterthought (been there, done that)

Agent:

This is the code that actually runs on the server. It is grouped here as the agent, but is actually two programs: mercury-agent and mercury-inspector. The agent will perform the following actions: Register with the RPC service, start the pong service, and the rpc agent service. The rpc agent service will route commands to discrete library interfaces to perform the actual work. The service itself will use a ZeroMQ REP socket and will block until a unit of work is completed.

The inspector code is responsible for querying the hardware, listening for CDP/LLDP broadcasts, and sending this information to the inventory service.

The inspector actually represents the bulk of the code that needs to be produced. Fortunately for us,

we've already written most of it and we really only have some tidying up to do in this respect.

Log:

I said log everything and I meant it. Ever transaction that occurs within the stack will be logged and sent to a centralized logging service. Each log message will be accompanied by a mercury Inventory ID, rpc Job ID, internal namespace, and release version when appropriate. This will allow third party tools such as elastic search to index based on transaction to an extreme degree of precision. With this concept, reports can be generated using any arbitrary metric, take this query for instance: Show me all PowerEdge R710's deployed with X app version in M14-.*.dfw3.

CLI:

Possibly written in Go. This will provide the first user interface to mercury. It will be a command driven tool that will interact with the rpc and inventory services. The intended users are RPC (rackspace private cloud) and intel devops. Some example usages:

mcli inventory group=M12 status
or
mcli inventory hardware.raid.adapter.0.disk>=5 status –active-only

mcli rpc group=OSIC-dev press_init –backend=RAX-CORE-IPCOM –template=press-ubuntu-15.04-dev-andy ← This is how you rekick servers with mercury!