$ go get Kraken

A stateful approach to cluster management

Time to rethink cluster management

- Modern language and software architecture
- Hardware and distribution agnostic
- Complete modularity
- Microservices architecture
- Community supported project
- *It needs to be smarter!*
What do we mean by “Smarter?”

We need our cluster manager to:

- **Boot** to a desired state
- **Maintain** that state, even in the event of soft failures
- **Manage** automated change of state, like rolling updates
- **Adjust** to changes in configuration management
- **Provide** administrative feedback on system health and state

https://xkcd.com/1319/
Ad hoc vs. Stateful Automation

**Ad hoc Automation**

Create scripts and hooks to handle known failure modes

- Lower initial cost of development
- No verification that failure was actually handled
- High “entropy” / difficult to maintain, or even track reliably
- Inflexible: generally designed to maintain only a single defined state
Ad hoc vs. Stateful Automation

**Ad hoc Automation**

Create scripts and hooks to handle known failure modes

- Lower initial cost of development
- No verification that failure was actually handled
- High “entropy” / difficult to maintain, or even track reliably
- Inflexible: generally designed to maintain only a single defined state

**Stateful Automation**

1. Keep track of desired state vs. current state
2. Know how to evolve the system from current state to desired state

- Verifies that the desired state is actually reached
- More maintainable, centralized source of automation
- Can be used to attain/maintain any state we know how to evolve to
Kraken is a State Engine

- **Kraken** is a state engine
  1. Tell kraken the state you want (via APIs)
  2. State changes trigger events
  3. Event listeners make things happen
- State store, event engines, services, etc. are modules
- The state store actively tracks the state of system components
Kraken is Distributed

- **Kraken** is a *distributed* state engine

1. State can propagate up/down a tree of nodes.

2. *(Future)* State can replicate sideways.

3. Every node tracks at least its own state and propagates changes up.

4. Any node can be given a portion of state, including the “running this service” state, temporarily or permanently providing services to all or a portion of nodes.

This node has been given state for the nodes below it, and had services enabled.

Here changes propagate both up and down the tree.

Admins can request & query state through the APIs

Admin changes through API

Sideways replication is future work.

State replicates between these nodes.

Nodes track their own state and propagate changes up the tree.
Kraken is Modular

- The Core of Kraken:
  - Distributed State Engine (and query language)
  - Event Engine/State Evolution
  - Plugin Management
- Services are modules, and are controlled through State
- Image distribution and loading is modular
- Different classes of modules can be added as needed (Scheduler integration, BMC interfaces, etc.)
- Modules can be Go interfaces, or outside processes communicating via RPC or ReST API
State of the Kraken

- Basic set of functional microservices
- Can boot multiple architectures
- Uses layered container images in reference implementation
  - ...but much still to be done.
- Code just released on GitHub:
  - http://github.com/hpc/kraken
We need smarter cluster management for the future of HPC

References

Auxiliary slides
Consistency & Availability of state

- It’s more important that we keep nodes running
- It’s less important if nodes occasionally do something wrong
  - As long as it fixes itself
  - And doesn’t interrupt jobs
- We should be consistent when we can, but available always
- We need to scale HUGE
  - Paxos, Raft, 2-phase commit... are out.
  - Look to Azure, AWS, etc... who all use Eventual Consistency for this kind of service.

CAP Theorem

We need two kinds of state

1. “Configuration state”
   - What we want the system to look like
   - Is specified by the administrators (or configuration management)

2. “Discoverable state”
   - What the system actually looks like
   - State is automatically discovered on the node(s)
Four rules of Kraken state

1. There will always be a well-defined *source of truth*

2. We will *never guarantee* synchronicity of state

3. State can be wrong as long as it *eventually converges*

4. State will be *small*
Rule 1: Source of Truth

1. The source of truth for Configuration state is the Full State Node (FSN)
   - Configuration state is set through the FSN
   - In event of failure, the nearest available node to the FSN is trusted

2. The source of truth for Discoverable state is the Single State Node (SSN)
   - SSN has modules to read actual state, e.g. hardware health
   - If the SSN is not initialized, or declared dead, the parent is trusted
Rule 2: Asynchronous

- We never guarantee that either configuration or discoverable state is the same throughout the tree
- There is no equivalent to a “barrier” or “sync” operation
- Any microservice that requires such an operation must provide its own mechanism for sync
- This *greatly* improves scalability
- Example: multicast image deployment would need to have its own mechanism to say all nodes are ready to receive

Figure [1]: Incremental replication between CouchDB nodes.

Rule 3: Eventual consistency

- Different parts of the tree may be out of sync at any given time, but...

  ➤ In absence of changes, sync will converge

  ➤ And the max time to converge is well-defined

- Hence, we may make mistakes

  ➤ But we will correct them “quickly”

Rule 4: State is small

- State synchronization is accomplished with 1-way “hello” messages containing the state of a node.
- State needs to be contained with a single network packet (ideally single node).
- We should build trees to load balance microservices, not the state engine itself.
- This allows recovery from partial state node failure to simply skip to the parent.

Kraken: Why?

- Current open provisioning systems are old, poorly maintained and too restrictive
- Need a flexible platform that can easily test new techniques
- Need a cluster manager that is designed to scale to meet demands of NextGen supercomputers
- Needs to be written in a modern language with modern design patterns
- Needs to support diverse hardware and architectures