Challenges in Management Services for Distributed Storage

and how Tendrl addresses them

Mrugesh Karnik, Red Hat
22nd March 2017
Tendrl was originally conceived to allow administrators to provision, monitor and manage multiple software defined distributed storage systems (currently Ceph and Gluster) under the same modern web interface.
Major areas for implementation

- Storage system state monitoring
- Management operations
- Provisioning on an existing platform
- Comprehensive logging and notifications
- Flexible, unified API
- Modern web interface
Storage system state

- Data modeling: RBDMS, NoSQL?

Tendrl doesn't need to understand the storage system state, merely interface with it.

This is where the abstractions come in. But they don't scale because they're not generic enough. Tendrl goes all the way with it's abstractions: objects and their interfaces.
Tendrl's Object Model

- The most abstract we could go: objects.
- Represent everything as an object. Every object has a state, attributes and actions.
- It isn't necessary to 'understand' an object's implementation, just it's interface.
- Storage entities are all represented as objects.
- Some entities are defined as part of Tendrl's 'standard library', such as 'host' and 'cluster'.
- Storage system specific entities are defined by the storage system's Tendrl integration and added dynamically to Tendrl, such as 'ceph osd', 'gluster volume'.
The object model allows Tendrl to treat every entity the same and doesn't require hardcoded support in Tendrl itself. Storage systems' integration modules are free to defined their own objects and their interfaces.
Object example: Ceph pools

```
objects:
  Pool:
    atoms:
      create:
        enabled: true
        inputs:
          mandatory:
            - Pool.poolname
            - Pool.pg_num
            - Pool.min_size
          optional:
            - Pool.max_objects
            - Pool.max_bytes
            - Pool.ec_profile
        name: "Create Pool"
        run: tendrl.ceph_integration.objects.pool.atoms.create.Create
        type: Create
        uuid: bd0155a8-ff15-42ff-9c76-5176f53c13e0

delete:
  enabled: true
  inputs:
    mandatory:
      - Pool.pool_id
  name: "Delete Pool"
  run: tendrl.ceph_integration.objects.pool.atoms.delete.Delete
  type: Delete
  uuid: 9a2df258-9b24-4fd3-a66f-ee346e2e3720
```
Tying objects together: Flows

- Every object can have **atoms**. Atoms are idempotent actions that can be performed on the object itself.
- In the future, we would be able to associate atoms with object state, so that, eg. a 'stop' atom can be executed only if the object is in the state 'on'.
- Multiple atoms are tied together in **flows**.
- Flows are operations that can be exposed to the end user via the API.
Example flows: Ceph pools

```yaml
namespace.tendrl.ceph_integration:
  flows:
    CreatePool:
      atoms:
        - tendrl.ceph_integration.objects.pool.atoms.create
description: "Create Ceph Pool"
enabled: true
inputs:
  mandatory:
    - Pool.poolname
    - Pool.pg_num
    - Pool.min_size
    - TendrlContext.sds_name
    - TendrlContext.sds_version
    - TendrlContext.integration_id
run: tendrl.ceph_integration.flows.create_pool.CreatePool
type: Create
uuid: faeab231-69e9-4c9d-b5ef-a67ed057f98b
version: 1
DeletePool:
  atoms:
    - tendrl.ceph_integration.objects.pool.atoms.delete
description: "Delete Ceph Pool"
enabled: true
inputs:
  mandatory:
    - Pool.pool_id
```
Flow to API endpoint

- The API parses the flows made available in a 'well-known location' and makes endpoints from them at: /<cluster_id>/<flow>
- To discover what flow endpoints are available, GET the /<cluster_id>/Flows endpoint, which is dynamically generated as well.

```ruby
get '/:cluster_id/Flows' do
  cluster = cluster(params[:cluster_id])
  flows = Tendrl::Flow.find_all
  flows.to_json
end

post '/:cluster_id/:flow' do
  cluster = cluster(params[:cluster_id])
  flow = Tendrl::Flow.find_by_external_name_and_type( 
    params[:flow], 'cluster'
  )
  halt 404 if flow.nil?
  body = JSON.parse(request.body.read)
  job = Tendrl::Job.new( 
    current_user
  )
```
Jobs

- Via the API endpoint, flows are invoked as jobs.
- Jobs have a TendrlContext object (which is also defined as an object with attributes).
- The TendrlContext object provides details for job routing, such as:
  - A specific node or a list of nodes
  - A specific cluster
- All Tendrl operations are asynchronous and are handled as jobs.

More on job routing later.
Definitions

- The object model and flows are together called **definitions**.
- Definitions are defined in **YAML** files, which are called **definition files**.
- **This is the core abstraction in Tendrl. The definitions and the 'language' to parse and work with the definitions is the glue that ties the whole of Tendrl together.**
- The 'business logic' for Tendrl resides in the definitions. Other components, such as the API, are either dumb or implement specific objects for their own domain.
Object Model's developer impact

- Tendrl 'core' itself is storage system agnostic.
- The storage system specific codebase is aggregated in individual 'integrations'.
- Both Tendrl core and integrations make their objects available for management using the definition files.
- The codebase is extremely modular, easy to develop and test.
- Integrations themselves can be written in any language, because to Tendrl, the only things that directly matter are the definition files.
Object model mapped to code

tendrl/ceph_integration/objects/
  ├── config
  │    ├── __init__.py
  │    └── definition
  │         ├── ceph.yaml
  │         │    └── __init__.py
  │         └── ecprofile
  │                ├── atoms
  │                │        ├── create
  │                │        │            └── __init__.py
  │                │        │    └── delete
  │                │        │        └── __init__.py
  │                │        └── __init__.py
  │                └── flows
  │                                ├── delete_ec_profile
  │                                │        └── __init__.py
  │                                └── __init__.py
  │    └── event
  │         └── __init__.py
  └── global_details
      └── __init__.py
  └── node_context
      └── __init__.py
  └── pool
      └── atoms
Tendrl Components

- **API**: A standalone, stateless application that exposes the Tendrl interface.
- **Central Store**: etcd, as a clustered, distributed key-value store. Home to all the definition files, jobs, notifications, state cache etc.
- **Node Agent**: Tendrl's core per-node workhorse.
- **Integration**: Storage system specific component for Tendrl.
**Tendrl Core**

- Trio of Node Agent, Central Store and API.
- Node Agent runs on every node Tendrl manages. This includes the Tendrl Central Store and API nodes as well.
- Central Store is connected to by everyone: Node Agent, API, Integration.
- The only inbound connections are to the Central Store.
- Provisioning flows are implemented via the Node Agent. These include the flows to provision:
  - Storage system specific integration modules for existing deployments
  - Storage system specific provisioning systems for creating new clusters from scratch.
- The framework already supports being able to deploy Tendrl components themselves, but the flows are yet to be written.
Storage System Integrations

- The Integrations do the following jobs:
  - Gather and monitor the storage system state and keep Tendrl's cache updated.
  - Supply the storage system specific definitions for Tendrl to interpret the state.
  - Supply the storage system specific definitions with the operational flows.
  - Implement the storage system specific objects and flows.
Tendrl wants to dip directly into the Source of Truth: the storage system itself.
For the versions currently supported by Tendrl, neither Ceph nor Gluster provide a suitable source of state information.
Tendrl thus uses a rados based integration with Ceph and accesses the maps directly. The maps are interpreted using the definitions.

For Gluster, the integrations run on each of the node and gather information from each of the nodes into the Central Store. This combined state representation, like for Ceph, is interpreted via definition files.
The primary reasons for state caching are:
- Monitoring and notifications
- Transactional operations
A side-note on REST APIs

- REST APIs are akin to searching for information when you know specifically what you want to ask for.
- Tendrl relies on a more 'browsing' approach where it has all the information, which is then indexed via definition files. Any information that is in the index, Tendrl can access.
- In a deployment, updating configuration is cheaper than updating code.
- Since Tendrl gathers all the information that there is to be known, making it aware of the parts of that information that aren't currently indexes, is a matter of updating the definition files.
- Tendrl also shies away from doing differential updates to the state, wherever possible, preferring to replace chunks of the information, depending upon the source.
Let's take a step back..

Does the integration between Tendrl and the storage systems need to be completely dynamic?

Yes, but there are certain corners that we can cut.

- Tendrl must always know a few things:
  - It needs to understand hosts and their storage hardware.
  - It needs to understand how to interact with provisioning systems specific to storage systems, such as gdeploy and ceph-ansible.
  - It needs to understand how to detect an already running storage system cluster to be able to 'import' it.
Tendrl's built-ins

- Tendrl currently officially supports Ceph and Gluster.
- It ships with detection modules to be able to detect both of these systems' processes, versions, clusters and the roles of the nodes within the clusters.
- It ships with platform detection modules to detect the operating system.
- It ships with inventory modules that currently support gathering the local storage and network inventory.
- It ships with wrapper modules that enable it to talk to provisioning systems.

The 'cut corners' are the storage system specific modules that ship with Tendrl itself. But even then, they're modules and are known to Tendrl via definitions.
Roles

- Tendrl has a concept of **roles**, which are implemented via **tags**.
- Tags are hierarchical. Some of them are hardcoded, some supplied dynamically via definition files. Examples:
  - tendrl/node
  - tendrl/api
  - tendrl/central-store
  - ceph/mon
  - ceph/osd
  - gluster/peer
  - provisioner/ceph
  - provisioner/<cluster_id>
- For practical purposes, roles and tags are synonyms.
Roles, continued

- Tags are used to assign roles to a specific Node Agent or Integration process.
- In addition to the node or cluster based routing for jobs, the TendrlContext object can also indicate jobs to be routed based on:
  - A specific role which is known to Tendrl as advertised by the entity playing that role.
  - A specific role among a list of nodes.

Roles and tags allow the deployment topology to be dynamically updated. The implementation allows for a role to be migrated to a different node should the node assigned with the role encounter a problem.
The challenges addressed

- Tendrl Core is not directly tied to any storage system integration.
- Tendrl does not try to understand and model the information from the source of truth. It simply uses it.
- Apart from the Central Store, which can be clustered, there is no single point of failure.
- Even in cases of failure, it is possible to monitor each and every component of the deployment, including Tendrl's own components.
- The role framework enables dynamic topology updates in case of failures.

Tendrl focuses on providing a solid framework that enables robust state access, job handling and modular integrations.
Sysadmin stuff

- Tendrl's logging infrastructure is able to provide tagged, machine parseable logs that can be put through an external log aggregation system for the administrator to debug.
- The same logging framework also enables notifications.
- Tendrl uses systemd for service management. The service support itself is modular, so support for additional services such as NTP or DNS configuration etc. can be added easily.
- Tendrl's modular provisioning capabilities allow the use of not just ansible based, but also external provisioning systems, such as puppet or chef.
- It is possible to configure specific systems for specific roles via configuration files.
- Tiered configuration (deployment-wide, per node etc.) is work in progress.
Contributions welcome!

- Github presence: https://github.com/Tendrl
- Architectural guidelines, components overview etc.: https://github.com/Tendrl/documentation
- Propose changes, features by filing an issue at https://github.com/Tendrl/specifications/issues
That's all for now.
Questions?
Thank you.

mrugesh@brainfunked.org
mkarnik@redhat.com