ceph

Ceph中国社区

IT大咖说 知识共享平台

CEPHALOCON APAC 2018

THE FUTURE OF STORAGE

22-23 March 2018 | BEIJING

Deployment and Management of Ceph with Salt

(DeepSea)

Joshua Schmid
Software Engineer
SUSE Enterprise Storage

Agenda

I. **Introduction**

II. What’s Salt

III. What’s DeepSea

IV. How does it work

   I. Preparation

   II. Validation

   III. Deployment

   IV. Management

V. Features

Introduction

- Germany based
- Software Engineer at SUSE (5y)
- Full time on Deployment and Management Framework (DeepSea)

Agenda

I. Introduction

II. <u>What's Deepsea</u>

III. What's Salt

IV. How does it work

    I. Preparation

    II. Validation

    III. Deployment

    IV. Management

V. Features

What’s DeepSea

Accumulation of custom Python modules, Salt states and Salt orchestrations that enable you to deploy and manage Ceph at scale.

- Started at SUSE
- Easy to configure
- Highly scalable
- Customizable
- Simplify Management tasks

Agenda

I. Introduction

II. What’s DeepSea

III. <u>What’s Salt</u>

IV. How does it work

    I. Preparation

    II. Validation

    III. Deployment

    IV. Management

V. Features

What’s Salt

Salt is a Python-based open-source configuration management software and remote execution engine. Supporting the "Infrastructure as Code" approach to deployment and cloud management, it competes primarily with Puppet, Chef, and Ansible.

- 6 Years old
- Master – Minion architecture (concurrency)
- Based on ZeroMQ
- Highly scalable
- Extensible

Basic Assertions & Tasks

- A cluster consists of nodes
- A node should have a/multiple role/s
- - OSD, MGR, MON, RGW, MDS, IGW, NFS-G, openATTIC, Client-roles
- A role has certain requirements & restrictions
- A node needs to be configured (Deployment)
- Post deployment tasks (Management)

Agenda

I. Introduction

II. What's Salt

III. What's DeepSea

IV. <u>How does it work</u>

  I. Preparation

  II. Validation

  III. Deployment

  IV. Management

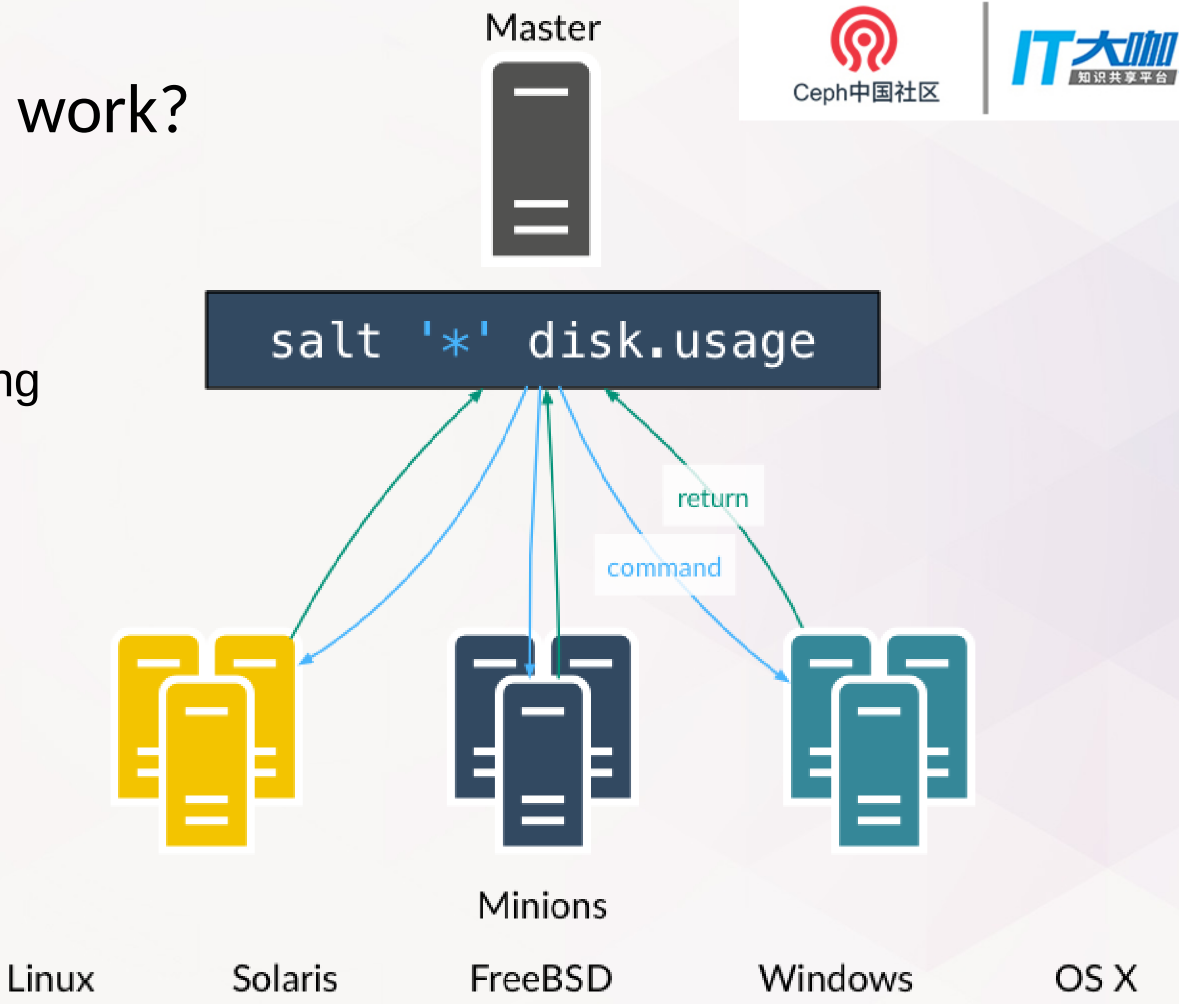
V. Features

How does Salt work?

Master advises **Minions** regardless of their underlying OS to execute commands.

‘disk.usage’ is a **module**.

A module can either be built-in or self-provided

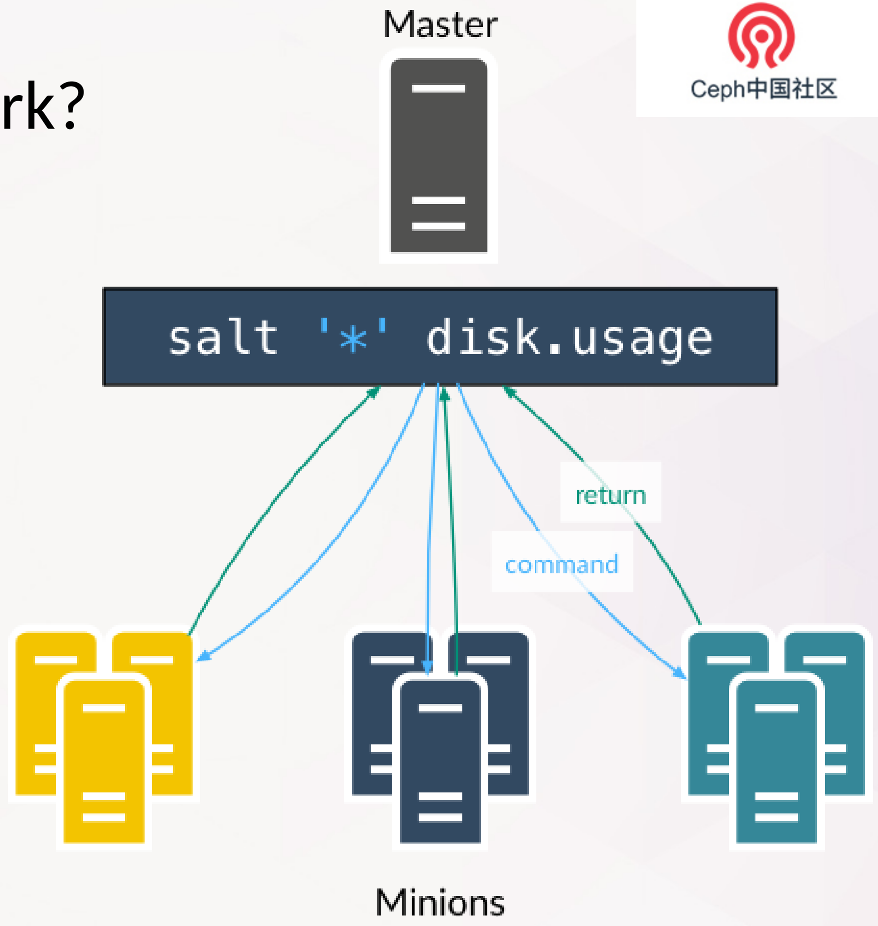


```
salt '*' disk.usage
```

How does Salt work?

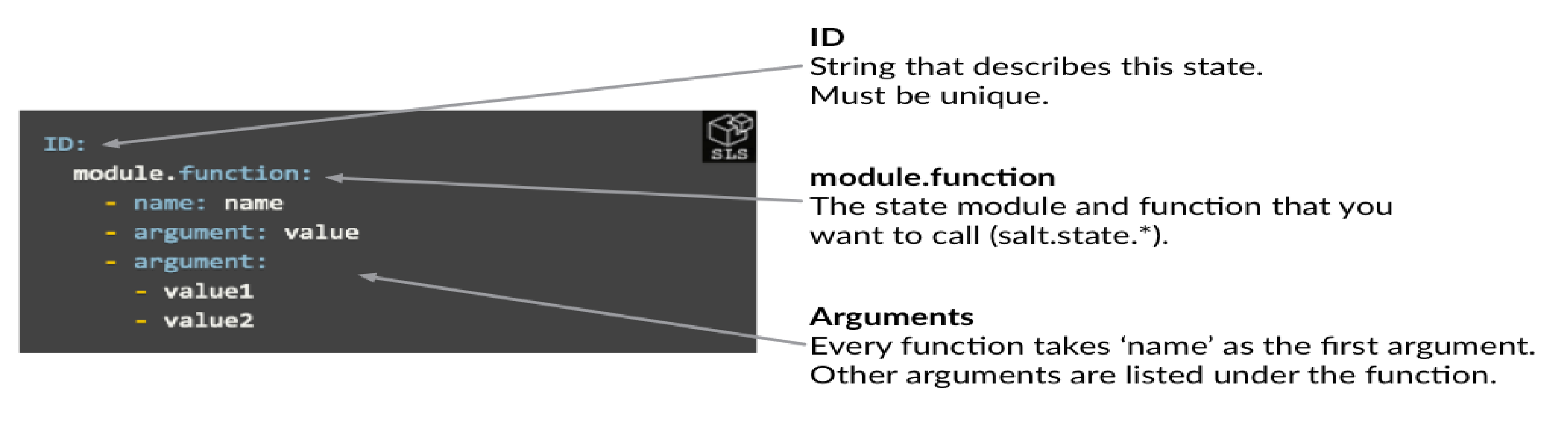
Control your Infrastructure with Code?

```
if (salt ‘*’ disk.usage > 100G ) {
    do_smth;
}
```



```
salt '*' disk.usage
```

How does Salt work?



```
ID:
  module.function:
    - name: name
    - argument: value
    - argument:
      - value1
      - value2
```

ID
String that describes this state.
Must be unique.

module.function
The state module and function that you want to call (salt.state.*).

Arguments
Every function takes ‘name’ as the first argument.
Other arguments are listed under the function.

Lives in a so called Salt State File **(SLS)**. Can be extended with **Jinja.**

States allow to customize and condense operations (e.g modules, functions)

salt ‘*’ state.apply your.state

How does Salt work?

“Lives in a so called Salt State File **(SLS)**. Can be extended with **Jinja.**”

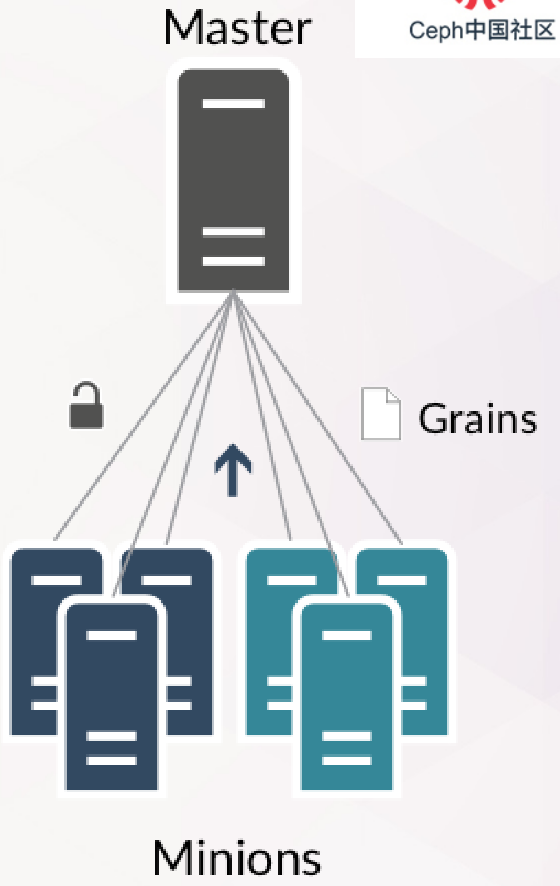
- Where do theses ‘values’ come from?
- How do we store data?
- How do we get information about the nodes?

How does Salt work?

Minions send data to **Master**

Dynamic information
about minions → **Grains**

Static and Custom information
about minions → **Pillar** Data



How does Salt work? - Recap

Recap:

We need to apply **Roles** to **Nodes** which need to meet certain requirements and want to be configured.

Grains provide information about a **Node(Minion)**. This allows to check for certain **Requirements** and **Restrictions**.

The **Pillar** provides information from the user.

Modules can be used to execute commands on the **Node**.

States described in **SLS** Files allow a consolidation of **Modules**, enabling us to logically group tasks that are needed to deploy **Ceph**.

Targets can be defined to match certain hosts that are assigned to a certain role.

Agenda

I. Introduction

II. What’s Salt

III. What’s DeepSea

IV. How does it work

    I. **Preparation**

    II. Validation

    III. Deployment

    IV. Management

V. Features

Preparation

You are in a Data-Center with 2500 Nodes. How do you identify each machine and assign a role to it?

Preparation

Short answer: The Pillar

In order to map minions to roles DeepSea uses a files called the **policy.cfg**

```
# Role assignment
role-master/cluster/node1*.sls
role-admin/cluster/node1*.sls
role-igw/cluster/node2*.sls
role-mon/cluster/node[1,2,3]*.sls
role-mds/cluster/node[:-1]*.sls
role-mgr/cluster/node[1,2,3]*.sls
```

Or even

```
role-mon/cluster/mon*.sls re=.*1[135]\.subdomainX\.sls$
```

This allows to tag minions with specific roles. The files that are being matched also contain extra information about that minion like it's public IP address.

Preparation

After the user applied the changes he made, pillar data can be verified by querying for it.

So we expect node1 to have the role-admin, role-master, role-mgr and role-mon

salt 'node1' pillar.get roles*

Will return a python structure that salt interprets and prints nicely.

```
node1:
  - roles:
    - mon
    ….
```

Preparation

That also means that we abstracted one layer. We don’t have to do:

salt ‘node1’ state.apply our.custom.state

We are able to do:

salt -I roles:mon state.apply our.custom.state

That allows us to not think about hostnames anymore. → More scalable

Preparation

Being able to target that way, we can call different commands in order to deploy Ceph.

‘salt -I roles:mon state.apply our.state.for.mon.validation’
‘salt -I roles:mon state.apply our.state.for.mon.configuring’
‘salt -I roles:mon state.apply our.state.for.mon.deployment’
‘salt -I roles:mon state.apply our.state.for….’

The same for every other Role?

That doesn’t scale...

Orchestrations

Like **States** allow to combine modules, **Orchestrations** allow grouping of **States.**

‘salt-run state.orch a.custom.orchestration‘

Has multiple **States** in it that do everything from

- Validation
- Configuration
- Deployment
- Maintenance

Orchestrations

Stage 0:

- *Pre-deployment, Patching, Syncing*

Stage 1:

- *Gathering information about cluster*

Stage 2:

- *Write to Pillar, Get user input, Configuration*

Stage 3:

- *Deploy Ceph-core services*

Stage 4:

- *Deploy Non-core services (mds, rgw, openATTIC)*

Stage 5:

- *Remove unwanted roles from nodes*

Orchestrations – Stage 0

‘salt-run state.orch ceph.stage.0‘

- *Activate salt-api (for remote control, openATTIC uses it)*
- *Sync modules*
- *Apply updates*
  - → *Call packagemanager.py (method configurable)*
  - → *Calls either Apt or Zypper (depending on the **grain**)*
- *Conditional restarts*

Features

Filestore → Bluestore migration

- *per OSD or per Node*

Baseline benchmarking(pre-deployment)

- *for rbd, cephfs, baseline, blockdev, fs*

Support for SLES, openSUSE, CentOS, Ubuntu(wip)

- *Contributions are welcome*

Automated restarts after update or config change

- *Detecting via lsof and checksums*

Non-disruptive updates and upgrades

- *Rolling updates that stop when a failure is detected*

Demo

Thank you