StraaS 1.0 Design

Edit New Page

Jump to bottom

Liu, Lubin(lubliu) edited this page on May 6, 2017 · 8 revisions

Welcome to Straas 1.0 wiki page. This wiki introduces why we need Straas 1.0 and the design details.

1. Motivation

We need to redesign a system if and only if the old system has non-negligible design defect and can not support must-have features. You can refer to this link to see the details of the old version.

The details of all the pain points to the old StraaS can be find at this link

2. Challenge

Redesign a system is never a trivial thing. We list the challenges here and they are also our aims.

2.1 The change to the code should be as small as possible

It costed over two years of comet team to develop the old version. Every small part of this version works fine now. We need not and should not to discard all of them. Reuse the code and tools can save much time. So, the new design should keep the excellent part of the old version and promote the defective parts.

2.2 Stable to changes

A non-ignorable disadvantage of the old version is that we have to involve a lot of efforts when we need do some changes. Changes include adding new features or doing some changes to existing component. A new feature means adding a new version of some existing component, or adding a complete new components and so on.

The new design should be stable to changes. This is the strongest reason to do the reconstitution.

2.3 Stable to infrastructure failure and well monitored

A very important feature of the cloud is un-stable. VMs may be unreachable anytime. The application on the VM may be crashed anytime also. As a base service for the whole Rheos, the StraaS components should be stable to the infrastructure failures.

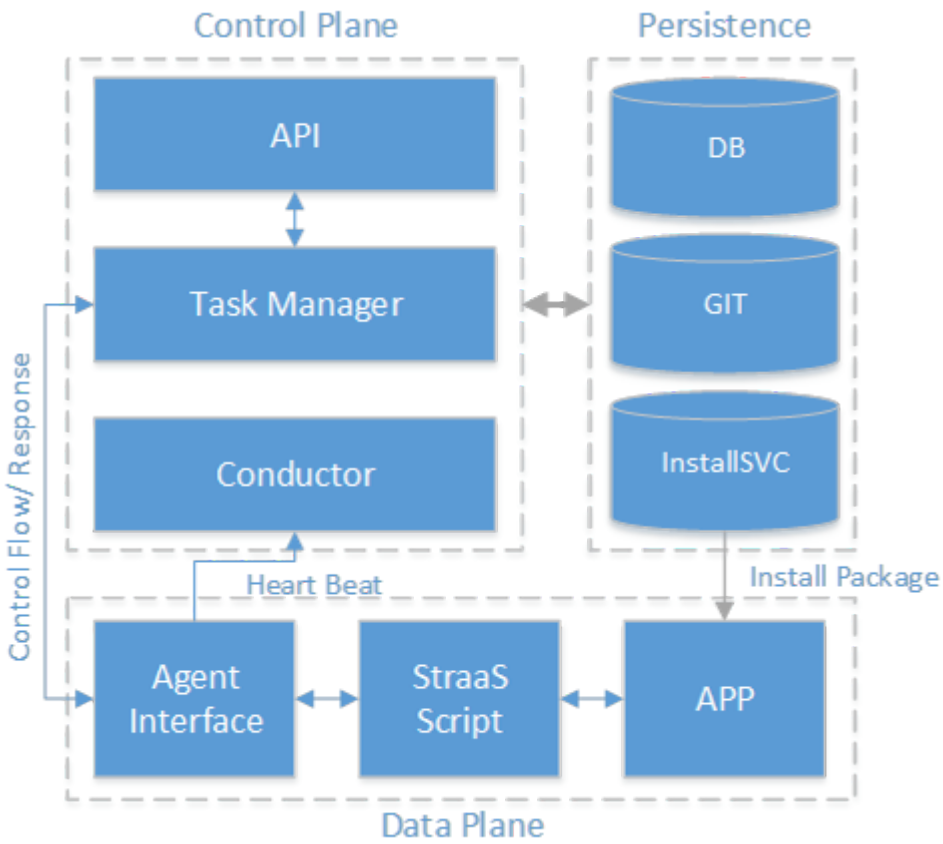
The stability doesn't mean "never failure". Many problems need human efforts and can't be handled by automatic tools. To solve the problems as soon as possible, we need a good monitor tool to do notification as earlier as possible.

2.4 Aim to opensource

When add a dependency, we should always keep one thing in mind: that dependency should be opensourced and won't disturb our license. The functions in straas should be as generic as possible. The ebay specific function should be plugined.

3. Architecture

The following picture shows the architecture of our new design, i.e., Straas 1.0. The overall architecture is much similar with the original one but small changes change everything.



3.1 API

The API receives user requests and returns the response. The response handler should validate the request first, like the user, tenant and token, then handle the DB. The validator should implement the auth & auth.

3.2 Task Manager

The task manager create a workflow for some requests, like creation and deletion, or redirect the request from the API to corresponding agent. All the components share one piece of code of all the workflows.

3.3 Conductor

The conductor receives the status from agent and update the status in DB. The checker also check the DB status and send out the alert email. The DB proxy receives status update request from the nodes and update the database.

3.4 Agent Interface

The entry point to handle the service on the VM. The interface is generic and the implementation is moved to the scripts.

3.5 Straas Script

This repo contains all the implementation of component specific logic. These scripts will be invoked by the agent interface.

3.6 App

The "App" represents the software installed on the VM. It could be kafka, zookeeper, storm..

3.7 Persistence

There are three persistence in our system. The DB stores the instance info just like the previous version. The git server stores the settings and the scripts for our system. The nexus server stores the debain packages.

3.8 Rabbitmq Server

We still use the rabbitmq server to decouple the requests between our services.

4. Design Details

According to the drawbacks of the old version and the aims we set, we design the new system mainly on the following aspects.

4.1 A new way to handle request

The design of the RESTful service part has two main drawbacks. One is that the paths in StraaS are not consistent and the other is that lacks of auth&auth.

4.1.1 New paths in StraaS

There is one item in the StraaS backlog which needs to be considered now: REST API routes and endpoints are not consistent and RESTy.

The paths and the returning code should be standard in the next version.

[NOTICE]

All the paths starts with '/straas'.

Only 4 API from the old version still work now: get, get_all, run_storm_jar, guestcmd

4.1.1.1 Existing API

1. **resource**: /v1.0/topologies

   action: Create a topology

   method: POST

   body sample: create topo api request body sample

2. **resource**: /v1.0/topologies

   action: Returns all topologies in a user's tenant

   method: GET

   response sample: list all topo api sample

3. **resource**: /v1.0/topologies/{id}

   action: Returns the information about a topology as JSON

   method: GET

   response sample: list one topo api sample

4. **resource**: /v1.0/topologies/{id}

   action: Deletes a topology

   method: DELETE

5. **resource**: /v1.0/components/{name}/nova_flavors

   action: Return the supported flavors for a particular component type. Component type is one of 'kafka'/'storm'/'zookeeper'

   method: GET

6. **resource**: /v1.0/components/{id} (flexup)

   action: Add a node to an existing component/cluster

   method: POST

   body:

```
 {
   "count": 2,                (optional. defaults to 1)
   "role": "zookeeper/kafka/nimbus/supervisor"      (optional. If not
specified, the new node will use the component level settings, else, it will
use the node level settings. For example, enable_cname)
 }
```

7. **resource**: /v1.0/nodes/{node-id}

   action: Removes a node from a component/cluster

   method: DELETE

8. **resource**: /v1.0/nodes/{id}/services

   action: Start the component service(s) running on the specified node

   method: POST

9. **resource**: /v1.0/nodes/{id}/services

   action: Stop the component service(s) running on the specified node

   method: DELETE

10. **resource**: /v1.0/nodes/{id}/services

action: Restart the component service(s) running on the specified node

method: PUT

11. **resource**: /v1.0/nodes/{id}/commands

    action: Run the command on the Guest Agent node

    method: POST

    body:

```
{
     "command":  "{command}",
```

```
    "args":  "{args}"   (optional)
}
```

4.1.1.2 Configuration Management API

Rules of the parameters

- branch: In our system, the branch has two kind of values. One is 'master', 'develop' style, which is controlled by our admin. The other one is 'topology_name@AZ' style, which is created in the creation workflow for users.

  [EXAMPLE] 'develop', 'test-1@phx01'.

- tag: The tag in the path or in the body always means 'user input string'. This tag could be related to the tag on our config repo and the mapping rule is like this: tag_on_git='{role}&{branch}&{tag}'. The role has two values, 'user' and 'admin', which is inserted by our service. The branch is which branch this tag related to and it should be specified when creates a tag. The 'tag' is the 'user input string'.

  [EXAMPLE] 'test-v1.0.0', 'admin&develop&test-v1.0.0'(git tag), 'user&test-1@phx01&test-v1.0.0(git tag)'.

- ref: A ref could be a 'brach' or a '{branch}&{tag}'.

  [EXAMPLE] 'master', 'test-1@phx01', 'test-1@phx01&test-v1.0.0'.

1. **resource**: /v1.0/configurations/tree/:ref/:path

   action: List all the configuration files under that folder or get the content of some file

   method: GET

   parameters:

Name	Type	Discription
ref	string	The name of the commit/branch/tag. Default: the repository's default branch (usually master)
path	string	The path could in two formats. One is "service{/:file}" and the other one is "app/:type/:version{/:file}". The first one is to list the files for the straas service and the second one is used to list the files of some component under some verison. The service side will return corresponding results based on "folder" or "file".

2. **resource**: /v1.0/configurations/tree/:branch/:path

   action: update the content of some file

method: PUT

parameters:

Name	Type	Discription
branch	string	The branch of this file. It can't be a tag, must be a branch.
path	string	Must be a path of some file.

body:

```
{
    "content": "the new content of this file",
    "message": "the commit message"
}
```

3. **resource**: /v1.0/configurations/tree/:branch/tags

   action: List all the tags under some branch

   method: GET

   parameters:

Name	Type	Discription
branch	string	The name of the branch. It could be "master", "develop" when the path equals to "service" and be the same as the topology name when the path equals to "app".

4. **resource**: /v1.0/configurations/tree/:branch/tags

   action: Create a new tags under some branch

   method: POST

   parameters:

Name	Type	Discription
branch	string	The name of the branch. It could be "master", "develop" when the path equals to "service" and be the same as the topology name when the path equals to "app".
tag	string	The tag name. User input this field.

   body:

```
{
    "tag": "the user input tag",
    "message": "the commit message"
}
```

5. **resource**: /v1.0/configurations/tree/:branch/tags/:tag

   action: Delete a tag under some branch

   method: DELETE

   parameters:

Name	Type	Discription
branch	string	The name of the branch. It could be "master", "develop" when the path equals to "service" and be the same as the topology name when the path equals to "app".
tag	string	The tag name. User input this field.

4.1.1.3 New API

1. **resource**: /v1.0/components/{component-id}/tree/:ref (configuration)

   action: Change the configuration for an existing component/cluster

   method: PUT

   parameters:

Name	Type	Discription
ref	string	The branch or the tag wants to trigger the config.

2. **resource**: /v1.0/components/{component-id}/dependencies (won't be enabled in the first version of 1.0)

   action: Get the dependency Info for a component

   method: POST

3. **resource**: /v1.0/nodes/scripts

   action: Sync the script repo to corresponding nodes

   method: PUT

   body:

```
{
    "scope": "Could be three values, i.e., topology, component or node",
    "items": [
        "the id for corresponding scope",
        "topology_id_1",
        "topology_id_2",
        "when the scope = topology"
    ]
}
```

4. **reousrces**: /v1.0/nodes/:node_id (replace api)

   action: re-image this node but keep all the settings

   method: PUT

5. **resources**: /v1.0/storage/:storage_type/regions/:dc/:path

   action: upload a binary file to the corresponding storage_type and path

   method: POST

   body: (Content-Type: application/octet-stream) The whole body will be regard as the file

6. **resources**: /v1.0/storage/:storage_type/regions/:dc/:path

   action: delete a binary file from the storage

   method: DELETE

7. **resources**: /v1.0/storage/:storage_type/regions/:dc/:path

   action: list all the file under this path

   method: GET

8. **resources**: /v1.0/components/:component_id/status

   action: for health check service to write the status back

   method: POST

   body:

   ```
   {"$node_name": "ACTIVE/INACTIVE/SKIPPED/UNREACHABLE"}
   ```

9. **resource**: /v1.0/nodes/files

   action: upload files to corresponding nodes

method: POST

body:

```
{
    "scope": "Could be three values, i.e., topology, component or node",
    "items": [
        "the id for corresponding scope",
        "topology_id_1",
        "topology_id_2",
        "when the scope = topology"
    ],
    "container": "the container name of this file",
    "filename": "the name of this file on swift",
    "target_path": "the target path on the local machine"
}
```

4.1.1.4 Old API (should be removed in the futrue)

1. **resource**: /stormjar/{storm_component_id} (old api, just for backward compatible)

   action: Uploads a storm topology jar to the Storm cluster (equivalent of executing storm jar command)

   method: POST

   body:

```
{
     "storage_url": "Swift storage URL (preauthurl)",
     "container": "Swift container name",
     "filename": "Name of file stored in Swift container",
     "main_class": "some.foo.bar.ClassName",
     "topology_name": "Storm topology name"
     "args":  [ arg1, arg2, arg3 ]   (optional)
}
```

2. **resource**: /

   action: Returns all topologies in a user's tenant

   method: GET

3. **resource**: /{topology_name}

   action: Returns the information about a topology as JSON

   method: GET

4. **resource**: /guestcmd/{component_id}/{node_name}

action: Run the command on the Guest Agent node

method: POST

body:

```
{
     "command":  "{command}",
     "args":  "{args}"   (optional)
}
```

4.1.1.5 CMS related API

1. **resource**: /v1.0/typologies/{topology_id}/cms

   action: create the CMS model based on the model in the straas db

   method: POST

   body: EMPTY

2. **resource**: /v1.0/typologies/{topology_id}/cms

   action: get the CMS model based on a topology id

   method: GET

3. **resource**: /v1.0/typologies/{topology_id}/cms

   action: update the CMS model based on a topology id. It will try to sync the model based on the topology info in StraaS db.

   method: PUT

   body: EMPTY

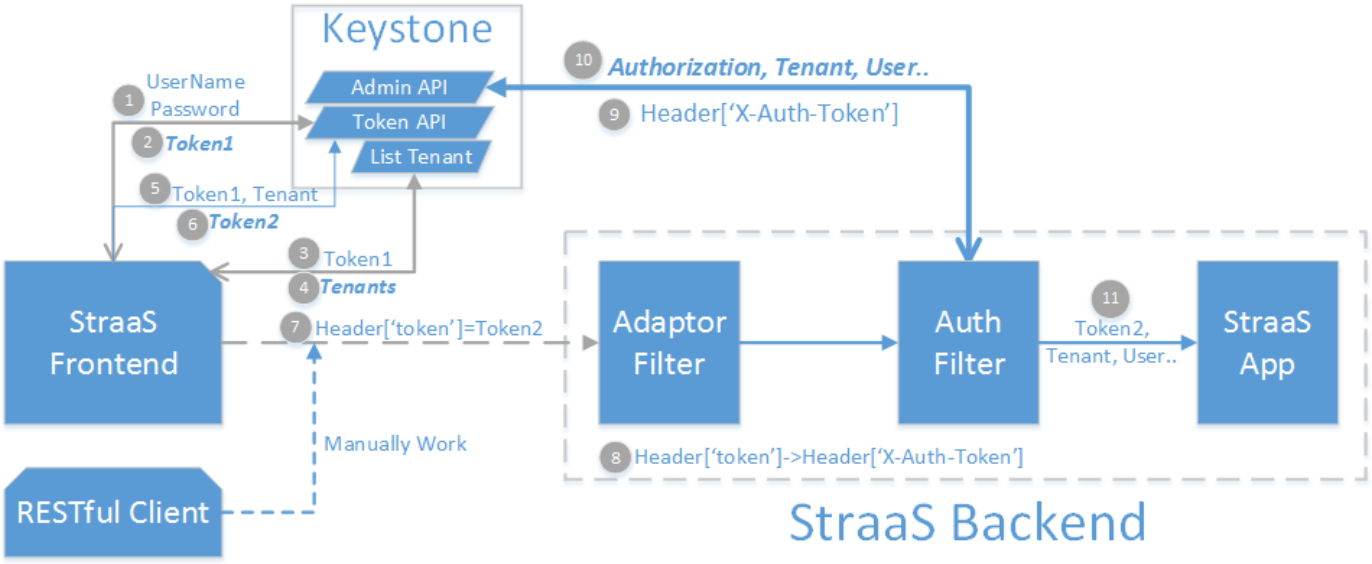
4. **resource**: /v1.0/typologies/{topology_id}/cms

   action: remove the CMS model based on the topology id

   method: DELETE

4.1.2 Auth & Auth

The process of the authentication and authorization in StraaS is as follows.



The whole process is as follows:

1. The straas front end sends the 'username' and 'password' to the keystone token api.
2. The keystone service returns a generic token, named as 'token1'.
3. The straas front end sends the 'token1' to the 'list tenant' api and get the tenant list belongs to 'token1'.
4. The 'list tenant' api returns the list of tenants.
5. The user picks a tenant and the straas front end send the 'token1' as well as the 'tenant' to the keystone token api.
6. The keystone service returns the tenant related token, named as 'token2'.
7. The straas front end put the 'token2' into the header, the key is 'token', which is hardcoded in the C3 horizon. On the other hand, user can use the restful client to send this request as well.
8. The adapter filter renames the 'token' key to the 'X-Auth-Token'. The implementation of 'auth filter' is from the third party and it only takes the 'X-Auth-Token' in the header as the token.
9. The auth filter sends the token to the keystone admin api and the service will validate this token. If it is invalide, the request wil be rejected. If it is valide, the auth filter will add the user info and tenant info to the header.
10. The auth filter passes all the header info to the straas app.

The authorization is path specific and we implement it in the straas app side. The rules are as follows:

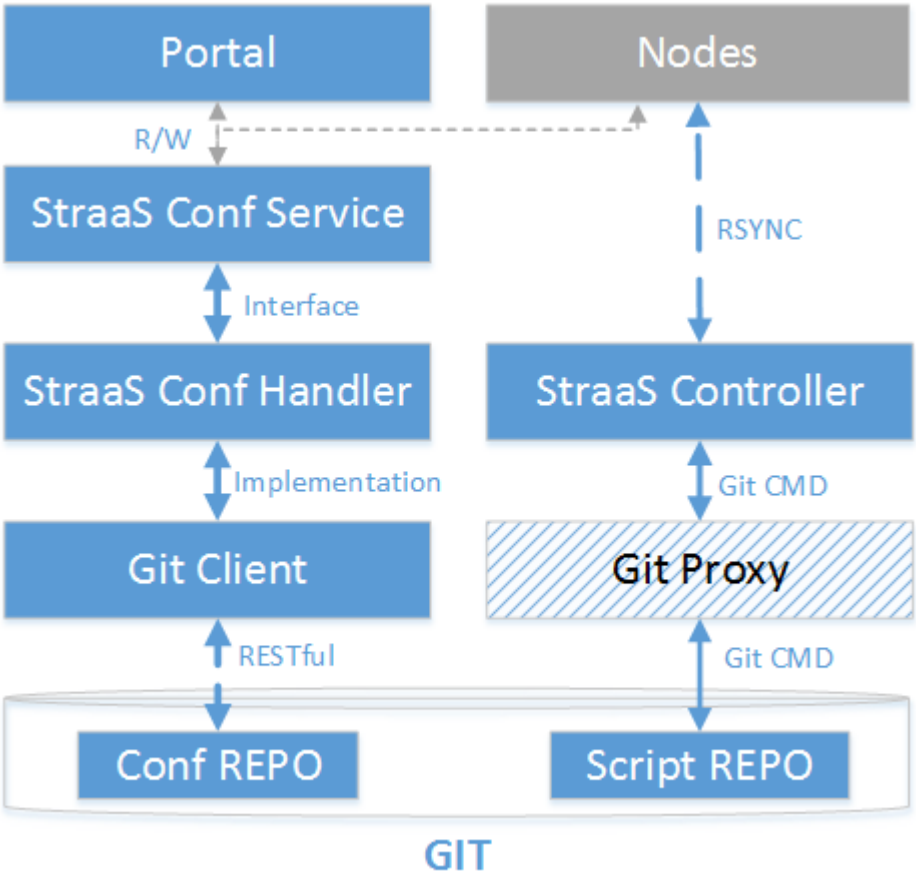
1. Admin could handle everything. The rule to judge an admin account is by the tenant id.
2. Normal user could only handle the topologies created by himself.
3. For configuration management api, the user could only read the default settings and change the values in the user.json file and apply the changes on the clusters created by

himself.

4.2 Centralized git service

4.2.1 Overview

Git is a good tool for trace the changes of files. In the new version, we stored two kinds of files on Git. "Conf" is a centralized repo to store the configurations for the straas service and for the topology. "Script" is a another repo to store the scripts for applications. The overall design for the git service is as follows:



For config repo:

- Both the portal and the nodes could read and write the files on GIT through configuration management api.
- The git service exposes the read and write API with RESTful.
- The git handler exposes the read and write API with interfaces. The implementation details are hidden behind these interfaces.
- The git client is a RESTful proxy to handle the files on GIT.

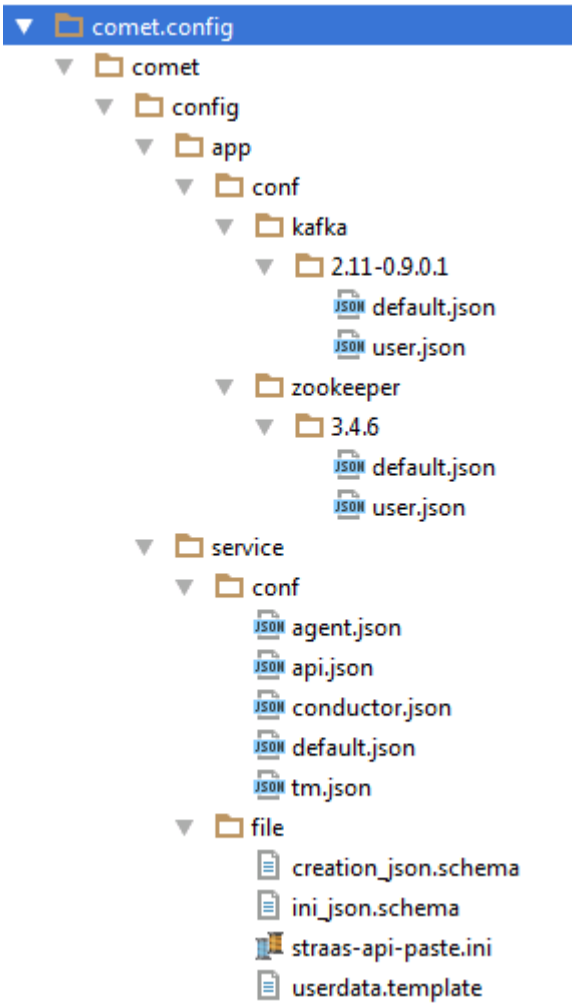
For script repo:

- In production, the git proxy syncs the data between git server and straas controller.
- The straas controller uses Linux build in command, rsync, to sync scripts to each nodes.

4.2.2 Centralized configuration repo

4.2.2.1 Introduction

Configuration files are distributed in too many folders and it is so hard to share them. We classify the configurations into three kinds: metadata, configuration for the service and configuration for the application. The catalog is as follows:



Here are the main design decisions for how to use this repo:

1. Two categories in this repo. The files under 'service' folder are used by straas service. The files under 'app' folder are component specific.
2. All the files under 'service' folder, the 'default.json' under 'app' folder and the keys in 'user.json' could only be modified by admin user.
3. All the admin user specific files should be maintained on the 'master' branch.
4. When use create a topology, a corresponding branch will be created. He can then modify the settings in the 'user.json' file and applys the changes to the corresponding clusters.
5. User can create TAG to mark some change. The rules for naming a TAG is as follows: {:CATEGORY}_{:BRANCH}_{:USER_ILLUSTRATE}. The first two items are added by the system automatically to do the permission control. The third item is defined by user themselves. "CATEGORY" is an enumeration variable and it should have 2 values: "service"

and "app". For admin specific files, the "BRANCH" should be "master", and for "user.json", the "BRANCH" should be as the same as the "topology" name.

6. The 'default.json' offers the common settings and the default values. The values in specific files, like 'api.json', 'user.json' will overried the values in the 'default.json'.
7. Manage the files always need to add a "mark" as a parameter. A mark could be tag or branch. The default is "master".

For the content of each file, the format is as follows:

```
{
  "type":"ini",
  "content":[
    {
      "section":"string",
      "doc":"string",
      "options":[
        {
          "key":"string",
          "value":"string",
          "type":"string",
          "exclusive":"boolean",
          "doc":"string"
        }
      ]
    }
  ]
}
```

The format is just like a json format 'ini' file. The meaning of 'section' and 'option' is as the same as the 'ini' file. The 'exclusive' marked this key is specific in straas if it is 'true'.

4.2.2.2 service side

All the services in StraaS, including "API", "TM", "Conductor" and "Agent", could share one configuration file. The shared <key, value> pairs can be maintained in the 'default.json'. All the service specific settings are maintained in the 'XXX.json' files. The duplicated key in the specific json file will override the the corresponding keys in the 'default.json'.

For example, in the '[DEFAULT]' section of the 'default.json', there is an 'option' called 'log_file' and the value is 'test.log'. This option in the '[DEFAULT]' section of 'api.json' is 'straas-api.log' and this value will override the 'test.log' when we need the 'CONF.log_file' value.

The function of the files under 'file' folder are as follows:

creation_json.schema: The json schema used to validate the creation body in the 'create topology request'. ini_json.shcema: The json schema used to validate the configuration files in the config repo. straas-api-paste.ini: The paste file for the straas api service. The way to handle a request is defined in this file. userdata.template: The template for userdata file. The sentence in this file will be excuted when the VM is first booted.

4.2.2.3 app side

The users can manage the configuration of their clusters. There are two kinds of settings in the conf file, default values and user speicifed values. The user can only change the value in a pre-defined set. The admin defined this set. This is used for permission control.

The duplicated keys in the user specific file will override the value in the default file.

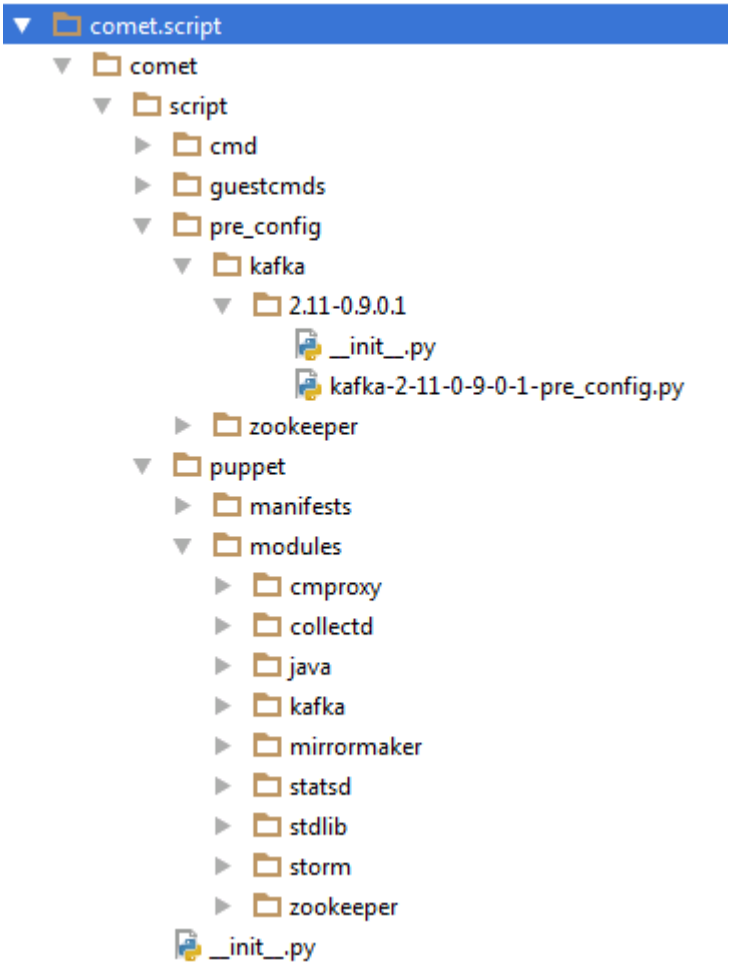
Each application, each version has an independent folder.

After modify the settings file, the user can trigger the configuration changed workflow and the restart service workflow.

4.2.3 Centralized script repo

4.2.3.1 Introduction

In this version, we extract the scripts out of the comet.agent. The centralized git service can also support the changes to the scripts. On the straas controller, it always maintains the newest version of the script repo. It syncs this repo to each node with 'rsync'. The catalog is as follows:



4.2.3.2 cmd

This folder contains the component specific scripts to call the puppet code and do the validation.

- runpuppet.sh: takes the script path as the only input param and excute the 'puppet apply' command to run puppet. This script is used in the 'configuration' related operation.
- validate.sh: validate whether the service is on. It always checks several ports in the good status. This script is used in the 'validation' related operation.

4.2.3.3 guestcmds

This folder contains all the guestcmd supported by our system. Previously, it is packaged in the comet.agent.

For servered for the generic workflow and the 'zookeeper as a service' in the future, we add more subfolder called 'zkcmd' in this part. For example, previously, the operation of creating a namespace in the zk is hardcoded in the zk manager. Now it is changed into a guestcmd.

To manage our system better, guestcmd is a wanderful tool.

4.2.3.4 pre_config

This folder contains the script of one stage in the generic workflow. Each component always has application specific code and we don't want to hard code it. So we create this path and in the generic workflow, the corresponding script will be added to the class path dynamically.

The script in under this path is written in python.

4.2.3.4 puppet

This folder contains all the component specific code which are used to do configurations. Previously, they are package in a package called comet.puppet and built in the images.

We also do some modifications to support versioning.

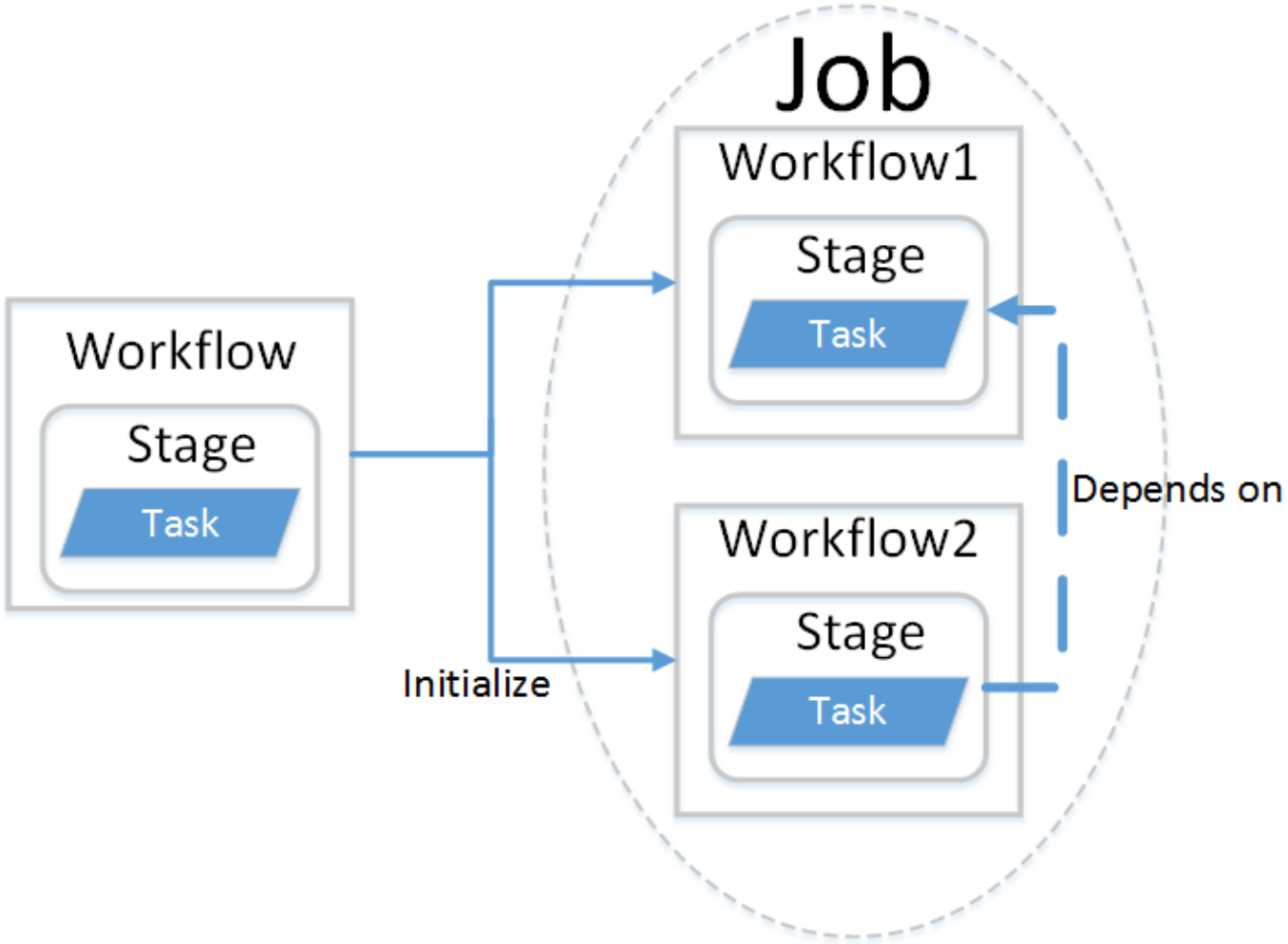
4.3 Abstract components and generic workflow engine

As we said in the "Motivation" part, the old design has a concept of "component", but doesn't regard it as an abstraction. After extracting the common fields from components and creating the abstraction of "component" metadata, we can then regard all the components as the same.

A workflow is always a DAG. We follow the implementation of the old version on how to schedule the tasks in a DAG, i.e., we also use the "taskflow" package. The difference between old and new version are what's in a workflow and how we generate the DAG.

4.3.1 Concepts

We would like to introduce several concepts in the generic workflow. There relationship is as follows:



- Workflow: A group of steps to handle one kind of request. For example, the creation topology request is corresponding to the 'creation' workflow, and the deleting request is corresponding to the 'deletion' workflow. A workflow is composited by several ordered stages.
- Workflow instance: A truly workflow, instantiate from the workflow. For example, we could initiallize a zk creation workflow to create a zk cluster.
- Stage: One step in some workflow. There is order info between stages. A stage could contain serveral tasks. We take the 'creation' workflow as an example, the 'spawn' stage and the 'verify spawn' stage is the first two stages for it.
- Task: Some small operations could be extracted as tasks. For example, the 'installation' stage in the 'creation' workflow contains two tasks, one is 'install the software on each node' and the other one is 'run rsync to put the comet.script to each node'.
- Job: All the operations to handle one specific request, and it is a group of workflow instances. For example, some user send a creation request and wants to create a zk+kafka clusters in this topo. The straas server will start a job, which creates two 'creation' workflow, one for zk and the other one for kafka.
- Dependency: In StraaS, all the dependencies are in stage level. In a job, the stages in two workflow may have dependency relationship. For example, for such a creation zk+kafka clusters job, the 'pre_config' stage in kafka creation workflow has to wait for the 'validate configuration' stage in zk creation workflow to be finished.

- Generic: The 'generic' in the 'generic workflow' means the primary code for some workflow is shared by all the components. For example, for most part of the creation workflow, the zk cluster and kafka cluster are all the same, and only the implementation of the 'pre config' stages are different. For every component, we don't need to maintain one piece of code for each.

4.3.2 The workflows in StraaS 1.0

In this section, we would like to introduce all the workflows in StraaS in detail.

4.3.2.1 The creation workflow

A new creation request body

In the previous version, there are a lot of hardcoded keys, which can't be used in a generic creation workflow. So we redesigned the whole request body. An example is as follows:

```
{
  "name": "test-1",
  "availability_zone": "phx01",
  "components": [
        {
            "name": "zk",
            "type": "zookeeper",
            "version": "3.4.6",
            "size": 1,
            "enable_fault_domain": true,
            "min_num_of_fault_domain": 4,
            "anti_affinity_group": "zk-test-group",
            "flavor": "tiny"
        },{
            "name": "strm",
            "type": "storm",
            "version": "0.10",
            "size": 3,
            "enable_cname": false,
            "depends_on": "zk",
            "submodules": [
                    {
                        "role": "nimbus",
                        "name": "nmbus",
                        "size": 1,
                        "enable_cname": true,
                        "flavor": "large"
                    },{
                        "role": "supervisor",
                        "name": "supervsr",
                        "size": 2
                    }
                ]
        }
```

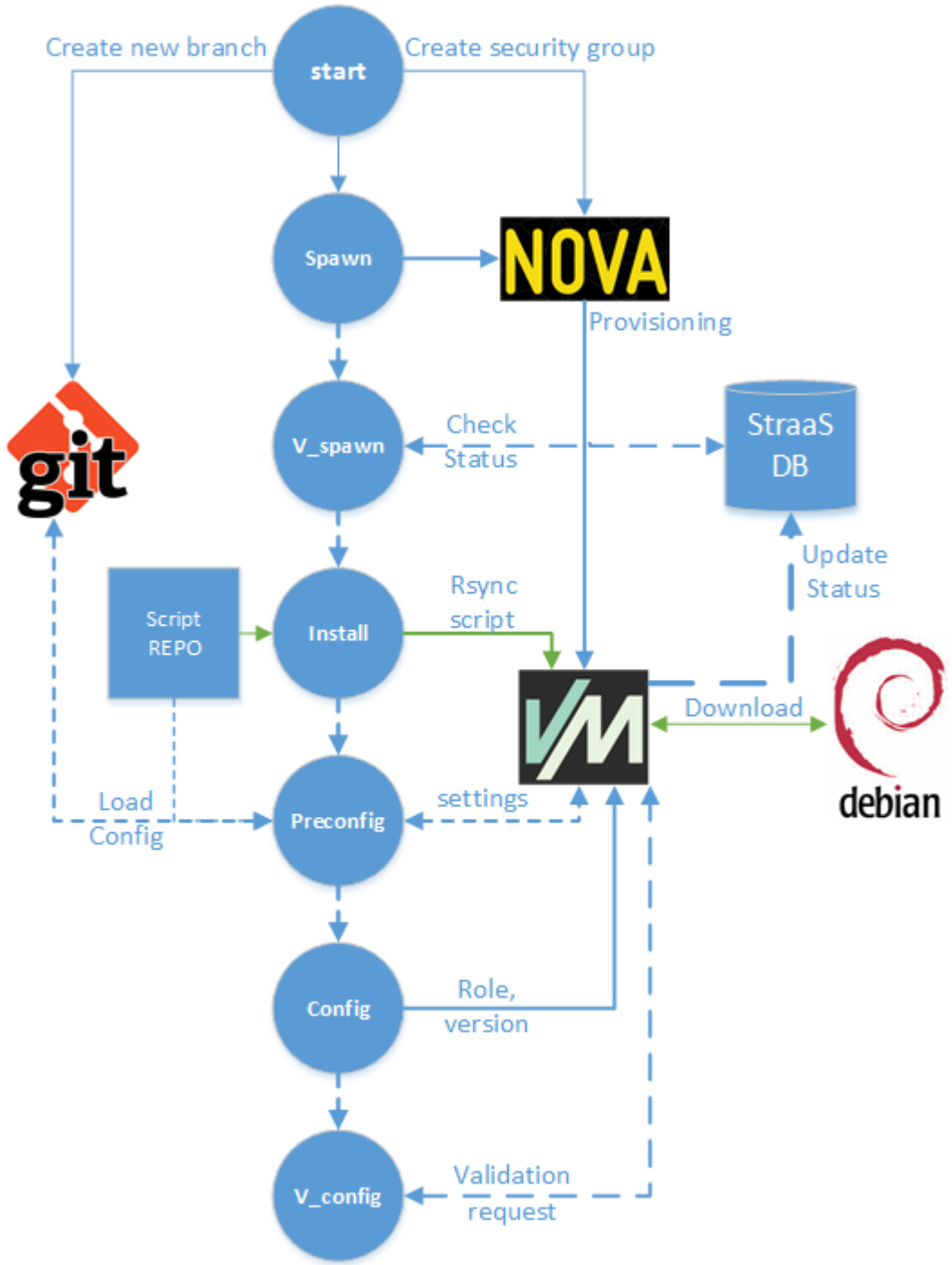
```
        ]
}
```

This new body is much powerful that before and the new features are as follows:

- Topology should pecify the az info. In this version, only the az+name in a topology table is unique. This feature is designed to support gloabl DB in the future.
- zookeeper, kafka, storm are regard as component in this system and need to be specified with the 'type' field.
- The user could decide whether to enable fault domain, cname and choose the anti-affinity group.
- The 'depends_on' field specify the name of some component in the same topology. This info will lead to the order in this job.
- The nodes in some components are not equal. For example, the storm has 'nimbus' node and 'supervisor' node. Submodules could override the setting in the component level.

The details of the stages in this workflow

The stages for the creation workflow are as follows:



The details of each stage are as follows:

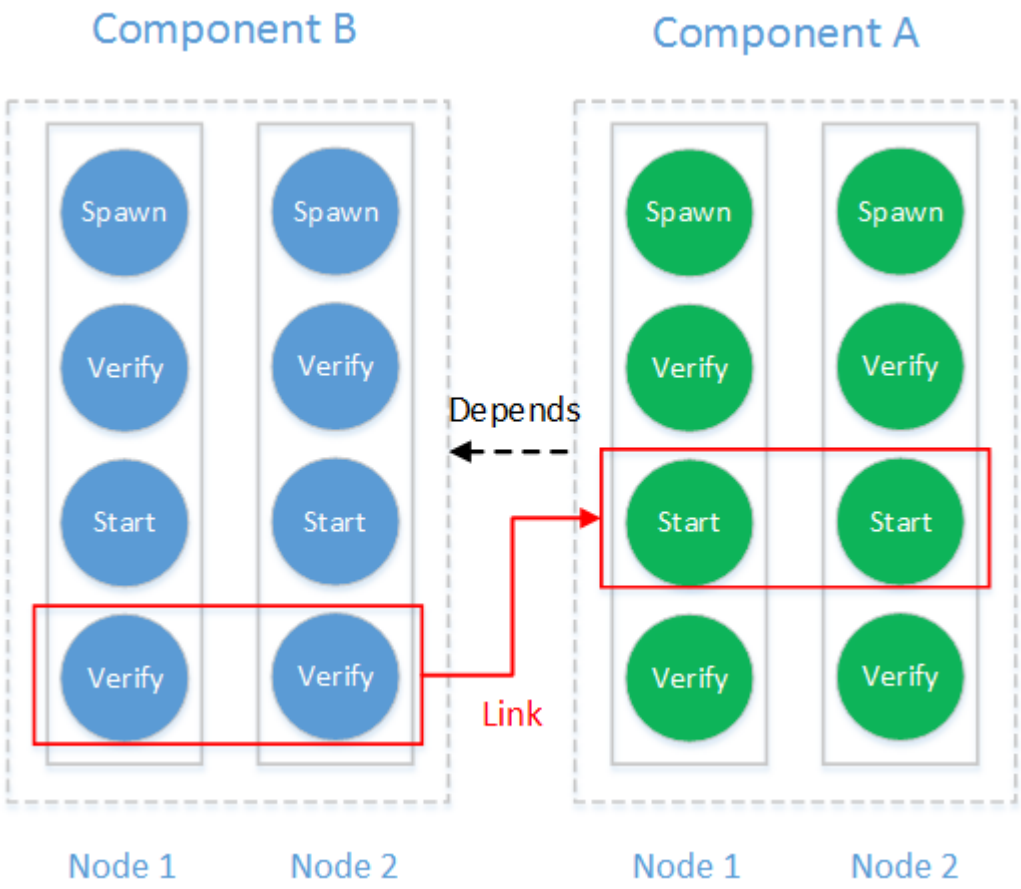
- start stage: This stage has two tasks, one is to create the security group for this topology and the other one is to create a branch for it. In one job, this stage should only be called once. The security group is equal to the tenant id of this topo.
- spawn stage: Send a node spawn request to the NOVA. This reqeust should specify the image id, the flavor, the userdata and so on. After we integrated with PE, the spawned node should pre-installed all the softwares on the base image.
- verify spawn stage: We start the comet.agent process in the userdata template. So we the VM is up, the comet.agent will send back a hello message, which contains the hostname and ip address of that node, and the straas service will update the state of that node to 'VM_BOOTED', which indicates that that node is spawned susccessfully. This stage is to check the state of nodes from DB periodly and wait for the nodes to be spawned.

- install stage: Install the corresponding components under corresponding version on each node and sync the comet.script to them. Previously, the component softwares are pre-installed on each image and we need to build a lot of image before the release. Installing them dynamically saves a lot of efforts.
- cname creation stage: This stage is optional and we don't draw it on the pic. Some node needs creation cName to used for the replace workflow. For each cname, the service will try to ping it until it is pingable.
- pre-config stage: This stage prepares all the configuration for that component needs in the config stage. These configurations are in map format and we called it 'sections'. The result of this stage is to generate a 'site.pp' file under some path and this file will be used next stage. This stage is component specific and the implementation of them are in the script repo. The code in the generic workflow will load the corresponding implementation at runtime. There are three tasks in this stage. 'feed_component_level_param' inserts the component level parameters into the sections object. 'feed_node_level_param' inserts the node level parameters into the sections object. 'do_other_operation' does other necessary operations. For example, for kafka, the first task will insert the 'zk_list' into the sections object, and the second task inserts the 'broker_id' into the sections obj. In the third stage, it create a namespace on the zk cluster it depends on.
- config stage: Send the config request to each node and call the 'runpuppet' script. This stage also will start the application service, which is wrotten in the puppet code.
- validate config stage: Send a validate reqeust to each node and the comet.agent will call the 'validate' script to check the process.

How do we handle the denpency in one job

The user could set the dependency info in the request. The 'depends_on' info is component level. In the DB this field is the component id of the component depends on.

To make it clear, we suppose there are two components and each node just need four stages. These two components will create two workflow instance. Component A is depends on component B, then the engine should build a link between the "verifyStart" stage of component B to the "start" stage of component A. This means that the "start" stage of component A must be blocked until the "verifyStart" stage of component B finishes, as shown in the following picture.



4.3.2 Deletion workflow

The deletion workflow should support the deleting the whole topology in the first version and later, it could also support deleting some component.

The steps for deleting the whole topology are as follows:

1. Delete the node one by one.
2. Delete the node info, component info, topology info from the DB.
3. Delete the tags and branches in the configuration REPO.

4.3.3 Flexup workflow

The steps for flexup workflow are as follows:

1. Provisioning new nodes.
2. Install the application.
3. Sometimes the new node need to do configuration changes, which means it will change the settings on the config repo. (Like zk flexup)
4. If there are configuration changes, the flow needs to trigger a configuration changes workflow.
5. If not, just need to config the new nodes. (Like supervisor flexup)

4.3.4 Flexdown workflow

The steps for flexdown workflow are as follows:

1. Delete the node in NOVA.
2. Delete the node in DB.
3. If there are configuration changes, the flow needs to trigger a configuration changes workflow.

4.3.5 Replacement workflow

The steps for the replacement workflow are as follows:

1. Delete the old node in NOVA.
2. Provisioning a new node in NOVA.
3. Config the new node with old configuration.
4. Restart nodes following some rules.

4.3.6 Configuration changing workflow

The steps for the configuration changing workflow are as follows:

1. Do the changes through the front end.
2. Select a group of nodes that should be influenced.
3. Trigger the configuration changes on each node.

4.3.7 Sync the script to all the nodes

The steps for the script sync workflow are as follows:

1. Upload the newest version of comet.script to the controllers.
2. Trigger a configuration change workflow to some topologies, components or nodes.

4.4 Generic agent and agent proxy

4.4.1 new agent design

"Altus" is a powerful system and we do learn a lot from it. The design of the "cronus" agent is especially match our demands. The cronus agent is a pre-installed component on the VMs of "Altus". As long as we write the scripts following the rules of "Altus", then the cronus agent can handle all the installing, configuring, starting, verifying, removing, restarting affairs.

In the old version, the logic to install the application is in the astro script, and the logic to do configuration and do verification are bound to the agent. In logic, these customized demands could all be handled by the application itself.

'Agent proxy' is part of the script repo. We extract the customized demands into the "Agent proxy" part. The path of the "Agent proxy" on the VM is predefined. All the demands are implemented as scripts. The naming rule of these scripts are also predefined. The agent manages and monitors the application by invoking the corresponding scripts in "Agent proxy".

The decoupling of agent and application makes sure that we need not modify the agent when new features come. All the application related logic are moved to the "Agent Proxy".

4.4.2 Agent interface

The interface on the agent side is part of the generic workflow. Some stages need to send a request to the agent side to trigger an operation. Most of these operations are implemented with script.

4.4.2.1 Install stage

The responsibility of this stage is to install the corresponding application with the right version on to this node. The interface is as follows:

```
def install(self, context, role, version)
```

Now the script is really simple, just one command, as follows:

```
apt-get install -y --force-yes {name}={version}
```

The interface on the generic agent side is a blocking call. After installing the software, it also do the verification with this command:

```
dpkg -l | grep {name} | grep {version}
```

[script path]:

None

[return value]:

success: GUEST_AGENT_INSTALL_SUCCESS

failure: GUEST_AGENT_INSTALL_FAILURE

4.4.2.1 Pre-config stage

The responsibility of this stage is to generate the 'site.pp' file for this node, which contains the full set of the settings. The interface is as follows:

```
def pre_configure(self, context, role, version, sections)
```

The 'sections' object contains all the settings for this node. These settings has two sources. One is from the git settings under 'app' folder. The other one is setted in the 'pre_config' stage on the server side, like the 'zk_list'. The data structure of this obj is {section, {option_key, option_value}}. A parser will translate the map to the 'site.pp' file. The rules are as follows:

- The section in 'section' is corresponding the class name in the 'site.pp' file.
- The 'option_key' and 'option_value' are corresponding to the key value paire in the 'site.pp' file.
- The string value on the git is arounded with '' and other types don't need to. This design is for the restrictions of site.pp key value.

[script path]:

:script_base_path/puppet/manifests/:role/site.pp

[return value]:

success: GUEST_AGENT_PRE_CONFIG_SUCCESS

failure: GUEST_AGENT_PRE_CONFIG_FAILURE

4.4.2.3 config stage

The responsibility of this stage is to trigger the puppet script to do the configuration operation. The interface is as follows:

```
def configure_server(self, context, role, version)
```

All the operations related configuration stage, like modify the setting files, move some jars and libs and start the service, are wroten with pupppet.

[script path]:

entry: :script_base_path/cmd/:role/:version/runpuppet.sh

impl: :script_base_path/puppet/modules/:role

[return value]:

success: GUEST_AGENT_CONFIG_SUCCESS

failure: GUEST_AGENT_CONFIG_FAILURE

4.4.2.3 validate stage

The responsibility of this stage is to varify the status of the service. It always checks several ports. The interface is as follows:

```
def validate_server(self, context, role, version)
```

The script is wroten with shell.

[script path]:

:script_base_path/cmd/:role/:version/validate.sh

[return value]:

success: GUEST_AGENT_VALIDATE_SUCCESS

failure: GUEST_AGENT_VALIDATE_FAILURE

4.4.3 "guestcmd" interface

"guestcmd" is a very useful interface provided by the old version. It provides a convenient way to manage the nodes. We will keep this interface and use more than before.

Previously, the scripts used for the "guestcmd" is packaged in the comet.agent and pre-installed on the VM. Now we could save them on the centralized script repo.

The guestcmds should be clasified to several groups:

- zk-cmd: The guest command related to the zk cluster, like create the namepace.
- kafka-cmd: The guest command related to the kafka cluster, like list the topic.
- storm-cmd: The guest command related to the storm cluster, like upload a jar.
- admin-cmd: The guest command used by the admin, like enable a new puppet script.
- assist-cmd: The guest command to assist our system, like list the guest command.

4.4.4 More powerful heartbeat

To coordinate with "Conductor", the heartbeat should contains more info, including service status and some metrics on that machine. We need more discussion on the monitor and alerting.

4.5 A global straas service

A global straas service has several advantages:

1. Provisioning the clusters in other AZ. For example, the straas service on the PHX01 could also provisioning the clusters in PHX02. This function is useful for tie1 cluster.

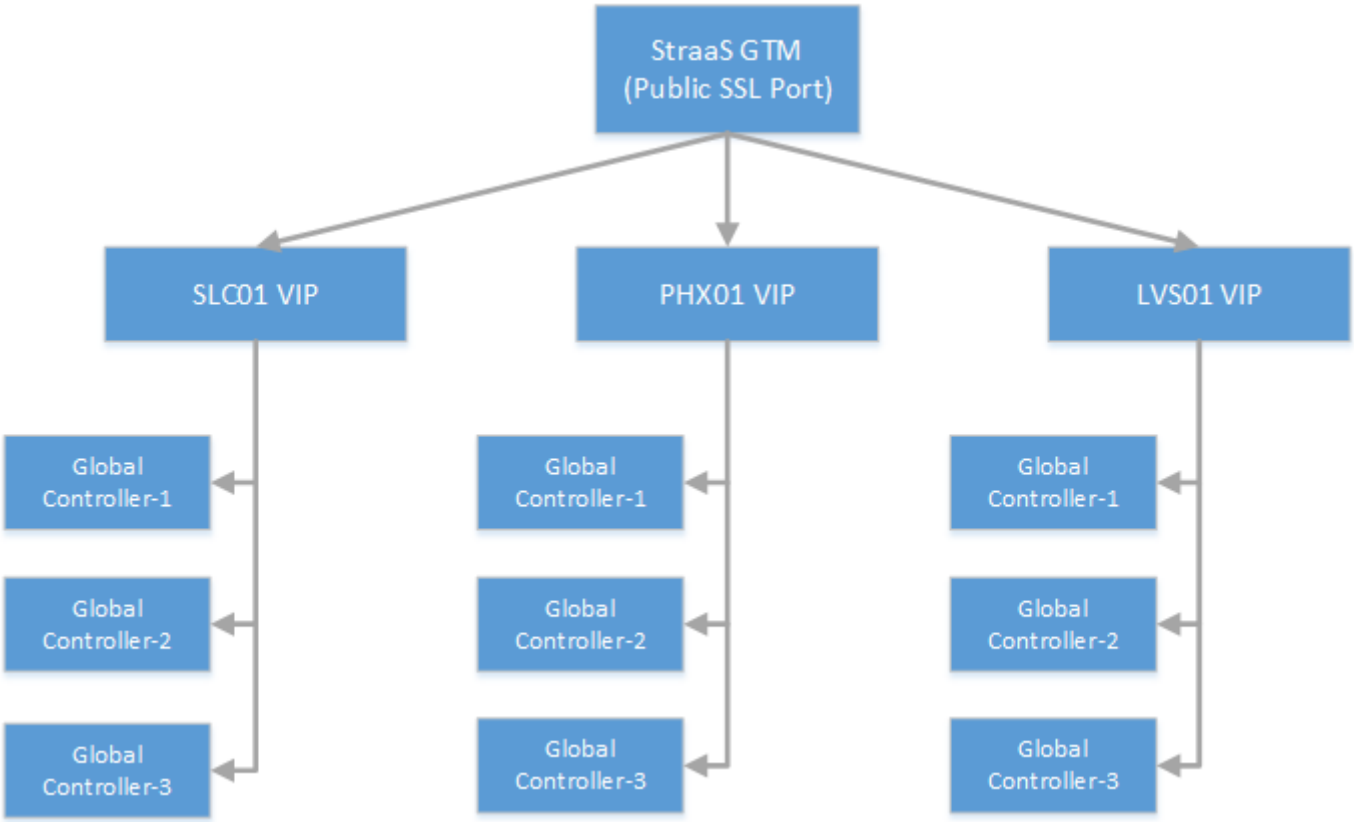
2. Simplify the release process. Now the settings for straas are different in each AZ. A global service guarantee that we don't need to change the settings like NOVA compute url for each AZ.
3. HA. Each controller is all-round. Even if the controllers in a whole AZ is down, we can still provisioning the clusters for that AZ.

Building such a global service is not a trivial thing. The challenges are as follows:

1. How to build a global DB?
2. How to build a global Rabbitmq cluster?
3. How to manage the AZ specific settings?

4.5.1 Architecture

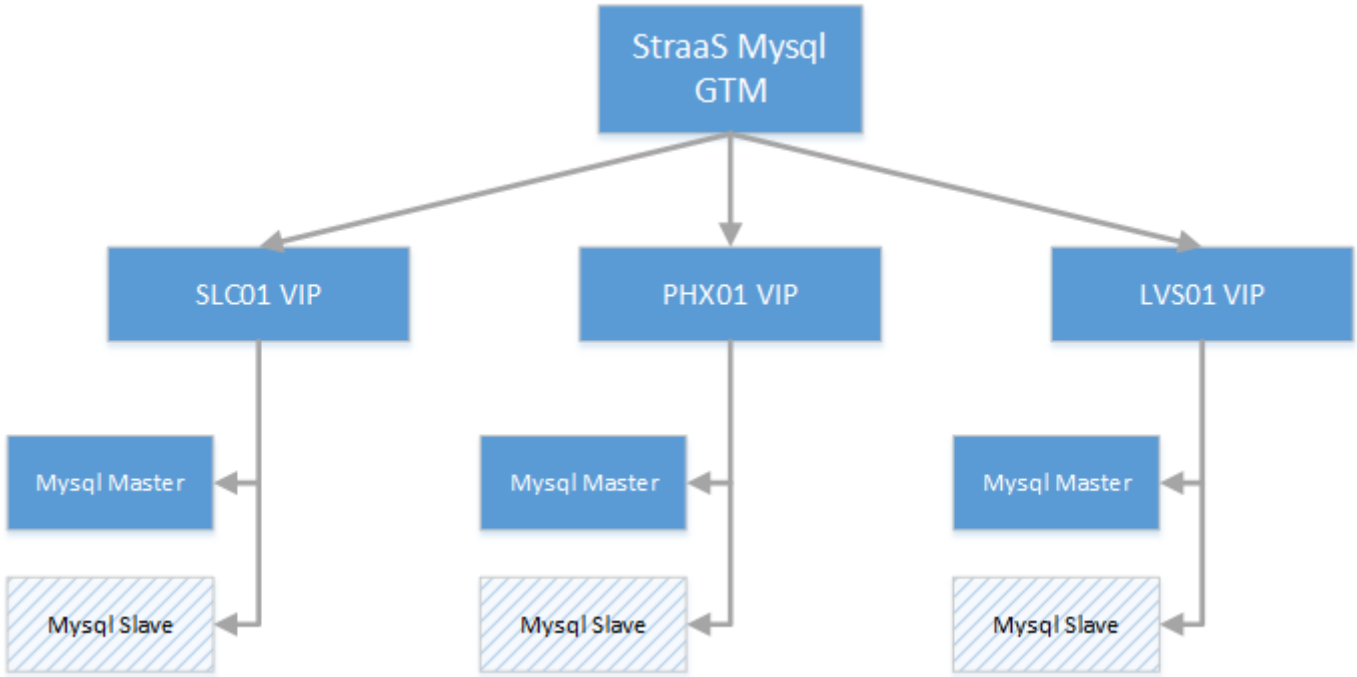
The architecture of the global straas service is as follows:



- All the request will go through StraaS GTM(Global traffic manager) first.
- The GTM will redirect the request to the corresponding VIP.
- In each AZ, we have 3 straas controllers and all the controllers are the same.
- In each controller, we start up a group of straas service, i.e., API, TM and Conductor.

4.5.2 Global DB

Build a global database is the first step to build a global straas service. The architecture of the global DB is as follows:



- The client side could config with the GTM.
- The GTM redirect the requests to the VIP in each AZ.
- In each AZ, there are one master node and one slave node.
- The master nodes are synced among all the AZs.
- When the master node is down, we can switch to the slave node.

4.5.3 Global rabbitmq cluster

We need to discuss more on this part. The basic idea is that we need to decouple the sender and receiver, which means we don't need to guarantee the sender receives the response from the receiver. We should use DB and git to link them.

4.6 A new way to deploy

- Build physical machines as straas controller.
- Add these controllers into the "Hostgroup" in Rainbow.
- Modify the configurations through straas front end.
- Build the new tar ball and upload to the installsvc server.
- Do the release on rainbow one AZ by one AZ.

4.6.1 Fabric based agent upgrade script

Basic rules

- avoid long and useless return log, like unzip and download file
- hit log when time consuming command start and end to execute
- redirect the error log, which returns from the fabric command instead of stderr
- always consider rollback and cleanup

- separate target hosts to batch
- start with small group but cover multiple dimension, like az/vpc/tenant
- verify automatically instead of manually check
- think about concurrence

Preparation

input: az, vpc

output: ip address list, grouped by clusters, consider to split to several batch

- [ATTENTION]: For the output, consider we have 100 ips totally, it could be divided into 4 batches, 1%, 5%, 50%, 100%. The ips int the same cluster must be in the same batch.

Upgrade

input: ip address list

output: success ip list, failed ip list

- [ATTENTION]: 1) clean up the files! 2) keep essential files for rollback

Rollback

input: ip address list

output: success ip list, failed ip list

- [ATTENTION]: 1) clean up the files!

Verification

input: ip address list, verification type(upgrade, rollback)

output: success ip list, failed ip list

5. Conclusion

In this section we conclude our new design. Straas 1.0 has a lot of shining points and of course, it is not perfect.

5.1 Advantages

1. Reuse most of the code. We keep most of the components in the old version and the overall system follows the structure of the previous version. Upgrading is much easier than building from the beginning.

2. Support versioning. The design is born to be extended. So, it is definitely support versioning.
3. Much easier to manage the configuration. Most of the configurations will be moved to the centrollized DB and we can provide a service to exposed an admin website to manage them conveniently. Everybody is saved from managing tens of configuration files under tens of folders.
4. Easy to extend. Three steps to add new version or components: First, you need to build a new debain package; Second, write app proxy scripts and upload to the git repo; Third, we need to do some configuration changes on the git service.

5.2 Disadvantages

1. We add one more dependency in StraaS. Git is a new dependency and the service will load the configurations on it when starts up. Nodes also need to load the scripts from it.
2. Because we don't want to write any application specific code in the service side, the script on the client side will be complicated.
3. We need to maintain ENV by ourselves. Move our service out of C3 controllers and C3 DBs, then we need to maintain by ourselves. We also need to maintain DEV, PRE-PROD and PROD.

5.3 The Comparison between new and old system

Benefits	Features	Past	Now
Save Operation Efforts	Zookeeper Replacement	300 mins	5 mins
	Kafka Replacement	60 mins	5 mins
	Configuration Change	Depends on cluster size	2 mins
Save Development Efforts	Add a New Version	30 days	3 days
	Add a New Component	35 days	3 days
	cName for All Components	10 days	3 days
Security & Normalize	Auth & Auth	Not Supported	Supported
	Customized Images	24	0
	Private Repo	2	0