fog05 - deep dive

Angelo Corsaro [1] Gabriele Baldoni [2]

January 31, 2018

[1]Chief Technology Officer, ADLINK Technology Inc.

[2]Technologist, ADLINK Technology Inc.

Plugin Manifest

The first thing we need to manage the various entity type is a plugin, so this is the schema for a plugin definition:

```
    {"$schema": "http://json-schema.org/draft-06/schema#",
    "title": "Plugin",
    "description": "Information for a plugin",
    "type":"object",
      "properties":{
            "status":{"type":"string"},
            "version":{"type":"integer"},
            "description":{"type":"string"},
            "uuid":{"type":"string"},
            "name":{"type":"string"},
            "url":{"type":"string" }
    }
}
```

Plugin Manifest Example

For instance let's see the manifest for the KVM Plugin

```
{
  "name": "KVMLibvirt",
  "version": 1,
  "uuid": "8fb33188-846c-45f8-83df-2a25e6b78049",
  "type": "runtime"
}
```

If *url* is specified means that the plugin is already avaiable in the plugin directory

Entity Manifest

Now it's turn of the generic entity manifest

```
{"$schema": "http://json-schema.org/draft-06/schema#",
"title": "Generic Definition",
"description": "entity definition schema",
"type": "object",
"properties": {
  "status": {"type": "string","description": "describe the action, so the desired lifecycle state"},
  "name": {"type": "string"},
  "uuid": {"type": "string"},
  "type": {"type": "string","description": "describe the type, can be an entity (vm, uk, be, us, ros....) or an a
  "version": {"type": "integer"},
  "entity_data": {"type": "object","description": "depends on entity"},
  "constraints": {"type": "object","properties": {
      "i/o": {"type": "array","items": {"type": "object","properties": {
              "type": {"type": "string", "description": "specific i/o name (gpio/com/....)"},
              "number":{"type":"integer"}}}},
        "networks":{"type":"array","items":{"type":"object","properties":{
              "type":{"type":"string", "description":"necessary interface type (wlan, eth, filbus, tsn)"},
              "number":{"type":"integer"}}}},
        "accelerators":{"type":"array","items":{"type":"object","properties":{
              "type":{"type":"string","description":"hw accelerator needed (netfpga, fpga, cuda, opencl...."},
              "number":{"type":"integer"}}}},
        "arch":{"type":"string","description":"required cpu architecture"},
        "os":{"type":"string","description":"required os"}}},
  "dst": {"optional": "true", "type": "string","description": "in case of taking-off/landing is the destination r
```

The entity data field contains information specific for each type of deployable entity

Example entity Manifest

This is the manifest of a simple cirros vm image

```
    {
    "name": "cirros_example",
    "version": 1,
    "type":"vm",
    "uuid": "9fa75e6a-d9c3-11e7-b769-5f3db30f0c2e",
    "entity_data":{
        "name": "cirros034",
        "uuid": "9fa75e6a-d9c3-11e7-b769-5f3db30f0c2e",
        "cpu": 1,
        "memory": 512,
        "disk_size": 5,
        "base_image":"http://192.168.1.142/virt-cirros-0.3.4-x86_64-disk.img",
        "networks": [{"mac": "d2:e3:ed:6f:e3:ef", "br_name": "virbr0"}],
        "user-data": "",
        "ssh-key": ""
    }
}
```

The status is added directly by the cli interface before sending the manifest to the node

Network Manifest

Next one is the manifest for a virtual network

```
    {
  "$schema": "http://json-schema.org/draft-06/schema#",
  "title": "Define virtual network",
  "description": "Information for creating a virtual network",
  "type":"object","properties":{
      "status":{ "type":"string"},
      "name":{"type":"string"},
      "uuid":{"type":"string"},
     "ip_range":{"type":"string"},
      "has_dhcp":{"type":"string"},
      "gateway":{"type":"string"},
      "vxlan_id":{ "type":"integer","description":"The vxlan id"},
      "multicast_address":{"type":"string","description":"the vxlan multicast address"}
    }
}
```

The status is added directly by the cli interface before sending the manifest to the node

Example Network Manifest

This is an example of a very simple virtual VxLAN network

```
    {
    "name": "example_vxlan",
    "uuid": "fdeb9506-d9c1-11e7-8387-f768dc1e4002",
    "vxlan_id":5,
    "multicast_address":"239.0.0.5"
}
```

Application Manifest

An application is a connected graph of entities or nested application, this is the manifest schema for the definition

```
    {
    "$schema": "http://json-schema.org/draft-06/schema#",
    "title": "Generic Application Definition",
    "description": "application definition schema",
    "type": "object","properties": {
        "status": {"type": "string","desciption": "describe the action, so the desired lifecycle state"},
        "name": {"type": "string"},
        "uuid": {"type": "string"},
        "description": {"type": "string"},
        "networks":{"type":"array","items":{
            "type":"string", "description":"manifest for each network"}},
        "components": {"type": "array","items": {"type": "object",
            "properties": {"name": {"type": "string"},
            "need": {"type": "array","items": {"type": "string"}},
            "proximity": {"type": "object"},
            "manifest": {"type": "string"}}
        }
        }
    }
}
```

Thanks to the need and proximity arrays it is possible to deploy components in the right order and in the right place to avoid stress of networks and nodes.

Example Application Manifest

This is an example of an application composed by two components, that provide a wordpress blog

```
{
  "name":"wp_blog"
  "description":"simple wordpress blog",
  "uuid":"application uuid",
  "components":[
      {
          "name":"wordpress",
          "need":["mysql"],
          "proximity":{"mysql":3}
          "manifest":"afos://sys-id/node-id/onboard/appuuid/component_name"
      },
      {
          "name":"mysql",
          "need":[],
          "proximity":{},
          "manifest":"afos://sys-id/node-id/onboard/appuuid/component_name"
      }
  ]
}
```

Getting information about nodes

We already saw how to start a node, but how we cam get information about nodes around us? We can use the *fos* cli interface to do that

```
$ fos node list
```

Shows basic information about node inside this system

If you want detailed information about a specific node

```
$ fos node -u <node_uuid> --info
```

This will provide you a lot of information about the hardware and software configuration of that node

Node interaction - Plugins

If we want to add a plugin to a node? Still use the *fos* cli interface

```
$ fos node -u <node_uuid> -p -a -m <path_to_plugin_manifest>
```

This will make the node load the plugin described in the manifest file

If you want information about plugins in a node

```
$ fos node -u <node_uuid> -p
```

Node interaction - Networks

If you want to create a virtual network for your service, you just need to write down your manifest and then to send it to the nodes

```
$ fos network -u <node_uuid> -a -m <path_to_network_manifest>
```

This will create the virtual bridge and the VxLAN interface inside the node

When you want to remove a network it's also very simple

```
$ fos network -u <node_uuid> -r -nu <network_uuid>
```

Node interaction - Atomic Entities

Ok, so you added a plugin, defined a virtual network, but there is something still missing... The entities!!! Managing the life-cycle of an entity is very simple:

- Define the atomic entity
  ```
  $ fos entity -u <node_uuid> -m <path_to_entity_manifest> --define
  ```
- Configure the atomic entity
  ```
  $ fos entity -u <node_uuid> -eu <enitity_uuid> --configure -iu <instance_uuid>
  ```
- Start the atomic entity
  ```
  $ fos entity -u <node_uuid> -eu <enitity_uuid> --run -iu <instance_uuid>
  ```
- Stop the atomic entity
  ```
  $ fos entity -u <node_uuid> -eu <enitity_uuid> --stop -iu <instance_uuid>
  ```
- Clean the atomic entity
  ```
  $ fos entity -u <node_uuid> -eu <enitity_uuid> --clean -iu <instance_uuid>
  ```
- Undefine atomic the entity
  ```
  $ fos entity -u <node_uuid> -eu <enitity_uuid> --undefine
  ```

Node interaction - atomic Entities

Some types of entities can be migrated, a migration requires that:

- plugin for that type of atomic entity is loaded in the two nodes
- the virtual network is created in the two nodes
- the nodes are configured correctly (eg. ssh keys, users....)

Issue the migration it is also very simple

```
$ fos entity -u <node_uuid> -eu <enitity_uuid> -du <destination_node_uuid> --migrate -
```

fos CLI interface - Manifest Check

When you write a manifest you may check if the syntax is correct, the *fos* cli interface can help you in this step

```
$ fos manifest [-n | -e | -a] <path_to manifest>
```

So you can check if the manifest of your entity, network or application is correct.

fos-get CLI interface - Deep inside the Distributed Store

You may want to check directly in the store, so we provided a *fos-get* command to help you do this

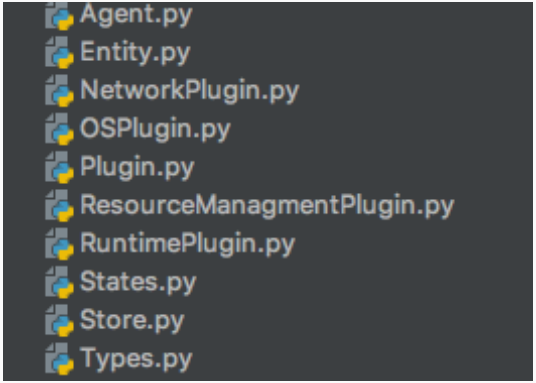
```
$ fos-get -u <uri>
```

This will simply print what it get from the distributed store. You should follow these rules when looking for an URI

- Every uri starts with '[a or d]fos://'. Where a stands for the actual store and d for the desired store.
- You can use 2 type of willdcards
  - * Will get information that are directly under the scope (eg. like an ls in bash)
  - ** Will get all information under the scope (eg. like a tree in bash)

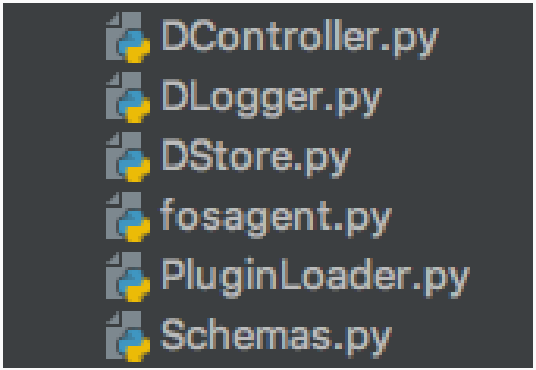
The fog05 python SDK

If you don't want to interact with fog05 using the cli interface you can write your own python script that interact with the store, or you can write a new shiny plugin. Or if you want you can also write down your own Distributed Store using the underlying protocol you want.



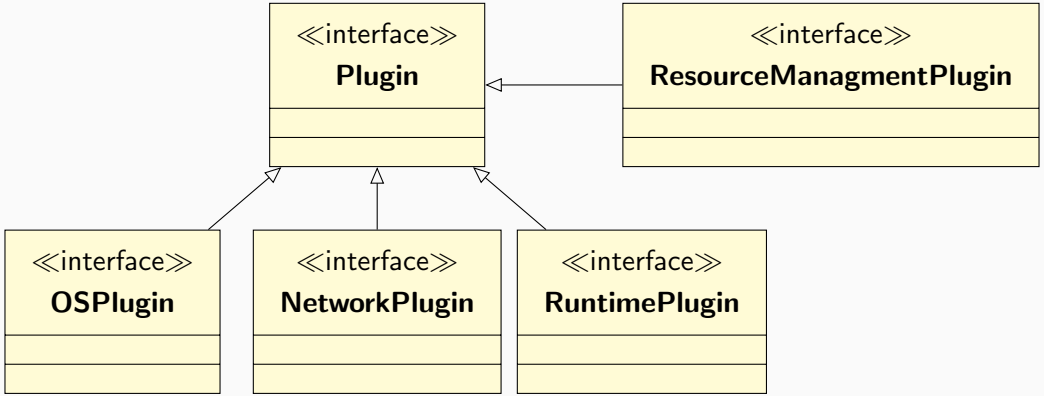
The fog05 python SDK

Or you can use the store we provide and simply interact with it.



The fog05 python SDK - Interfaces

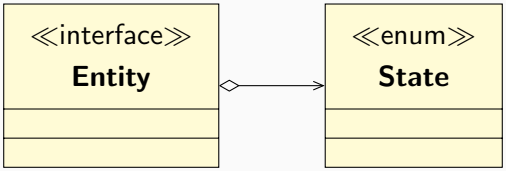
Let's see the UML of the interfaces for the plugins.



So you can simply implement one of these interface to manage entities, operating systems, network and resource management

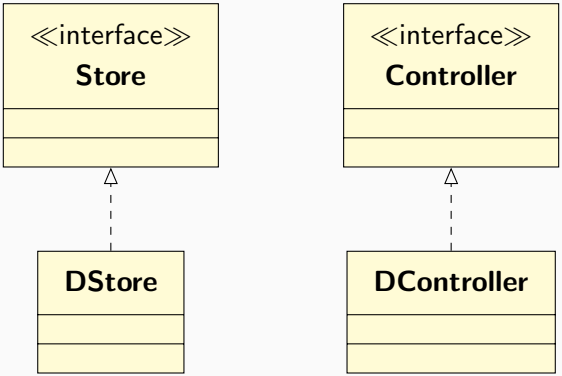
The fog05 python SDK - Interfaces

Interfaces for Entity and State



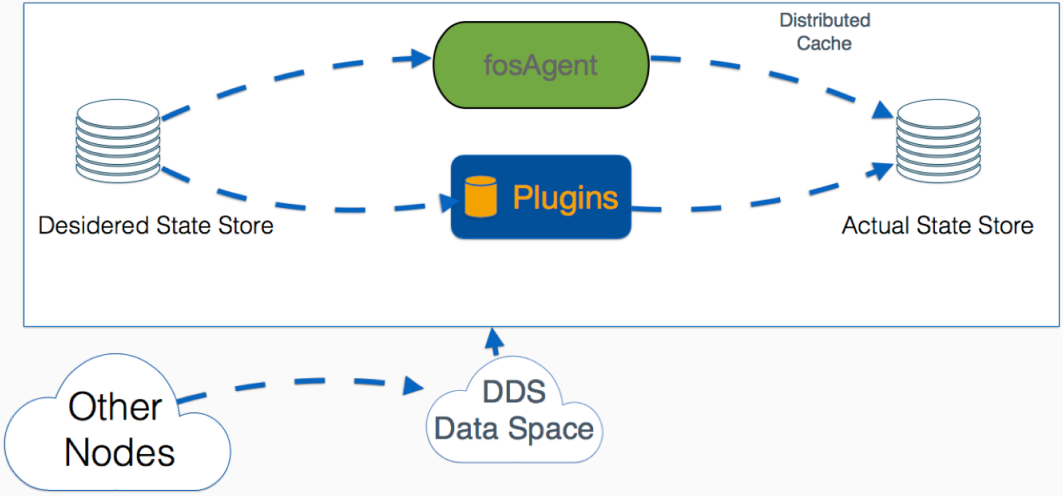
The fog05 python SDK - Interfaces

Let's see the UML of the interfaces for the store



DStore and DController implement the Distributed Store using DDS as underlying protocol

Distributed Store Architecture



Each plugin and the fosAgent observing the actual store their job is to make the actual store match the desired store

Demo Time!!

fog05 in action!