MiniProject

Aim: Building a Virtualization Orchestration Layer

Introduction - In today's world, Hypervisor is the new OS and Virtual Machines are the new processes. Many system programmers are familiar with the low level APIs that are exposed by the operating systems like Linux and Microsoft Windows. These APIs can be used to take control of the OS programmatically and help in developing management tools. Similar to the OS, Hypervisors expose APIs that can be invoked to manage the virtualized environments. Typical APIs include provisioning, de-provisioning, changing the state of VMs, configuring the running VMs and so on. While it may be easy to deal with one Hypervisor running on a physical server, it is complex to concurrently deal with a set of Hypervisors running across the datacenter. In a dynamic environment, this is a critical requirement to manage the resources and optimize the infrastructure. This is the problem that we try to solve in this project.

Problem Definition – Build a fabric that can coordinate the provisioning of compute and storage resources by negotiating with a set of Hypervisors running across physical servers in the datacenter.

Phase 1 Expected Outcome –

1. Resource Discovery:

Recourse Discovery refers to the component which search/discovers the available hardware recourses in the given setup. For the scope of this mini project you will be using a static file that gives you location of the hosts in the setup in the form of list of IP addresses. Using this information, you need to determine the available resources (CPU, RAM, HDD) on the hosts. This information will be used in the next step.

2. Resource Allocation:

Resource Allocation refers to the component, which decides what resources to allocate to fulfill the given request. For the scope of mini-project, resource allocation is done only when the request for creating a new Virtual Machine (VM) is received.

The important design aspect of resource allocation is that it should be loosely coupled in the sense that mechanism and implementation should be separate. What this means is it should be a possible to change the "Algorithm" for allocation with minimal code changes or still better with change in configuration only. For example, the Round Robin and Bin packing are to different policy/algorithms that decide VM should be created on which Physical Machine (PM). So it should be easy to change from Round Robin to Bin Packing. You would be using libvirt [4] to execute the actual request on the PM.

3. A REST API Server:

A server which provides the services for management and monitoring the virtual and physical infrastructure. The REpresentational State Transfer (REST) is a style of distributed architecture. For the mini project is it in enough to know that in REST, all recourse or concepts are identified/represented by URL. All the output is to be returned in the Json format [3].

4. One or More Clients:

At least a command line client to demonstrate the functionality that consumes the REST services. You can build more clients like android client [6] or browser plugin.

5. Installation Script

You are required to submit a script for setting up your system. The testing will be done on Ubuntu 12.04 64 bit. The server will have libvirt, python and java pre-installed. This image (meaning entire OS environment with all this packages) will be shared so that you can test your script before submission. If you require any other packages, write the installation commands in the script itself.

Syntax for the script:

./script pm_file image_file

pm_file : Contains a list of IP addresses separated by new-line. These addresses the Physical machines to be used for hosting VMs. A unique ID is to be assigned by you.

image_file : Contains a list of Images(full path) to be used for spawning VMs. The name of the image is to be extracted from the path itself. A unique ID is to be assigned by you.

After running the script, the rest server should be up and running.

API Specification:

VM APIs:

■VM_Creation:

- Argument: name, instance_type.
- Return: vmid(+ if successfully created, 0 if failed)
  {
  vmid:38201
  }
- URL: http://server/vm/create?name=test_vm&instance_type=type

■VM_Query

- Argument: vmid
- Return: instance_type, name, id, pmid
  {
  "vmid":38201,
  "name":"test_vm",
  "instance_type":3,
  "pmid": 2
  }
- URL: http://server/vm/query?vmid=vmid

■VM_Destroy

- Argument: vmid

- Return: 1 for success and 0 for failure.
  {
  “status”:1
  }
- URL: http://server/vm/destroy?vmid=vmid

■VM_Type

- Argument: NA
- Return: tid, cpu, ram, disk

```
{
   "types": [
      {
         "tid": 1,
         "cpu": 1,
         "ram": 512,
         "disk": 1
      },
      {
         "tid": 2,
         "cpu": 2,
         "ram": 1024,
         "disk": 2
      },
      {
         "tid": 3,
         "cpu": 4,
         "ram": 2048,
         "disk": 3
      }
   ]
}
```

- URL: http://server/vm/types

Block Storage APIs:

■VM_Creation:

- Argument: name, instance_type.
- Return: vmid(+ if successfully created, 0 if failed)
  {
  vmid:38201
  }
- URL: http://server/vm/create?name=test_vm&instance_type=type

■VM_Query

- Argument: vmid
- Return: instance_type, name, id, pmid
  {
  "vmid":38201,
  "name":"test_vm",
  "instance_type":3,
  "pmid": 2
  }
- URL: http://server/vm/query?vmid=vmid

■VM_Destroy

- Argument: vmid
- Return: 1 for success and 0 for failure.

```
{
“status”:1
}
```

- URL: http://server/vm/destroy?vmid=vmid

■VM_Type

- Argument: NA
- Return: tid, cpu, ram, disk

```
{
   "types": [
      {
         "tid": 1,
         "cpu": 1,
         "ram": 512,
         "disk": 1
      },
      {
         "tid": 2,
         "cpu": 2,
         "ram": 1024,
         "disk": 2
      },
      {
         "tid": 3,
         "cpu": 4,
         "ram": 2048,
         "disk": 3
      }
   ]
}
```

- URL: http://server/vm/types

Image Service APIs:

■List_Images

- Argument: NA
- Return: id, name

```
{
“images”:[
        {
        “id”: 100,
        “name”: “Ubuntu-12.04-amd64”
        },
        {
        “id”:101,
        “name”: “Fedora-17-x86_64”
        }
```

]
}

- URL: http://server/image/list

Suggested Steps:

1. Select the right tools by studying various Hypervisor/Cloud controller like CloudStack, OpenStack and Eucalyptus, libvirt. See [5] to get more details of the system like this.
2. Design the algorithm to coordinate the provisioning spanning multiple resources.
3. Design the REST interface that abstracts the fabric functionality.
4. Build clients that demonstrate the end-to-end use cases.

Submission format:

1. Put your executable (Jar/Exe/sh/py/*) into a folder named “bin”.
2. Put all your source code in a folder named “src”.
3. Put these two folders in a folder and Archive it with a your rollno as a name E.g. 201107616.tgz

Evaluation Method:

Fully automated. All the APIs will be called and the results will be checked against the actual. All the pass/fail calls will be reported. So be sure to follow API correctly.

All the code will also be checked for the duplication. Anybody found copying the code from others or directly from the internet/open-source projects will get no marks for this project.

References:

1. EC2 APIs
2. OpenStack APIs and Architecture.
3. http://www.json.org/
4. http://libvirt.org/
5. Similar Open Source Project: http://archipelproject.org/
6. Android Client example: VM Manager
7. Json Validation: http://jsonlint.com/
8. You can use curl to see your server’s request-response during testing phase.