Network and Service Operations Project OpenStack with Ansible

SHIVAKUMAR YADAV JABBU , ROHAN ALUGURI
Electrical Engineering with a focus on Telecommunication Systems
BLEKINGE INSTITUTE OF TECHNOLOGY
Karlskrona, Sweden
SHJA20@student.bth.se,ROAL22@student.bth.se

***Abstract*—OpenStack is a distributed cloud platform known for its complexity during the setup process. To address this, an Ansible playbook can be constructed as an automatic deployment script for deploying an OpenStack infrastructure. Additionally, a bash script can be utilized to execute the Ansible files. This project aims to streamline the deployment of an OpenStack infrastructure using automation techniques.**

***Index Terms*—OpenStack,Ansible**

I. Introduction

The objective of this project is to develop an installation script using Ansible playbooks that facilitates the deployment of simple OpenStack setups, while also allowing flexibility for more complex scenarios through the reuse of separate script components. The initial focus of the project was on establishing networking components such as networks, routers, subnets, security groups, and creating a security key. Furthermore, the project involved creating the necessary instances, including a Bastion host, and configuring Haproxy for load balancing. Python scripts were utilized to generate hosts and configuration files. The ultimate goal is to deploy Flask applications to backend servers and employ Haproxy as the frontend server for load balancing. Additionally, the backend servers are expected to respond to UDP service requests, while the frontend server will utilize Nginx for load balancing the backend servers [1].

II.

A. Design of project

OpenStack is a highly complex cloud platform that can be challenging to set up. To simplify the process, a shell script can be utilized alongside an OpenStack RC (Resource Configuration) file. The RC file contains essential information about the cloud environment and can be obtained from the OpenStack user's cloud portal or API. Here is a sample OpenStack RC file:

```
export OS_USERNAME=****
export OS_PASSWORD=***
export OS_AUTH_URL=https://*****:****
export OS_USER_DOMAIN_NAME=**
export OS_PROJECT_DOMAIN_NAME=***
export OS_REGION_NAME=****
export OS_PROJECT_NAME="****"
export OS_TENANT_NAME="****"
export OS_AUTH_VERSION=3
export OS_IDENTITY_API_VERSION=3
```

The project is organized into three stages: installation, operation, and cleanup. Each stage is controlled by a specific command: install, operate, and cleanup. These commands can be extended with additional parameters such as the OpenStack RC file, a tag name, and an SSH key file. Here is an example of how the commands could be structured:

```
install <openrc> <tag> <ssh_key>
operate <openrc> <tag> <ssh_key>
cleanup <openrc> <tag> <ssh_key>
```

1) Install: During the installation step, a temporary folder is created within the same directory to store essential files for the project. This folder and its contents are later removed as part of the cleanup process. The hosts file is initially empty, as it will be updated dynamically with instances in the playbook and used to perform tasks on those instances.

In the first step of the installation process, the OpenStack RC file is exported, providing the necessary credentials and configuration for accessing the OpenStack environment. This ensures proper authentication and authorization for subsequent steps.

Next, the Ansible playbook is executed using the ansible-playbook command. This playbook generates a keyname, network, router,security group and subnet, which are named based on the provided tagname in the command. The security group is configured to allow TCP port 5000, UDP port 6000, SSH port 22, and TCP port 9090, while enabling unrestricted intra-security group communication. These networking components are crucial for establishing the infrastructure.

The playbook then checks the servers.conf file to determine the required number of instances (backend servers) to be built. The network ID and key name generated in the previous step are used for all the backend servers.

```
- name : Create server nodes
    os_server:
      state: present
      name: dev{{ item }}
      image: Ubuntu 20.04 Focal Fossa 20200423
```

```
    flavor: flavorname
    key_name: ansible_key
    auto_floating_ip: no
    security_groups: default
    nics:
      - net-id: "{{ testnet.id }}"
  with_sequence: start=1 end =3
```

Within the previously established network, a Bastion host and Haproxy server have been constructed, each assigned a floating IP address. The Bastion host's floating IP is utilized for SSH connectivity to all servers, including the Haproxy server. The Haproxy server's floating IP is employed to validate HTTP and UDP requests. Additionally, two Python scripts have been executed to generate the hosts file and configuration file, both located within the same folder. Example of hosts file

```
[haproxy]
net_haproxy

[webservers]
dev3
dev2
dev1

[all:vars]
ansible_user=ubuntu
```

example of config file

```
host dev1
port 22
user ubuntu
IdentityFile /.ssh/id_rsa
hostname 192.168.0.29

host net_haproxy
port 22
user ubuntu
IdentityFile /.ssh/id_rsa
hostname 192.168.0.102

host net_bastion
port 22
user ubuntu
IdentityFile /.ssh/id_rsa
hostname 91.106.195.140
```

Following the completion of the installation process, the next step involves pinging the newly created hosts to verify their accessibility. This ensures that the infrastructure has been successfully set up and the hosts are reachable within the network. By performing the ping operation, you can confirm the connectivity of the hosts and validate the successful installation of the OpenStack infrastructure.In addition to the previous steps, Prometheus is utilized on the Bastion host to monitor the availability and health of all servers within the OpenStack infrastructure

2) Operate: n the operational phase of the project, the script runs in an infinite loop with a minimum interval of 30 seconds. If the previous loop finishes before the 30-second mark, the script will sleep until the remaining time is reached. During each loop iteration, the script checks the required number of servers specified in the "server.conf" file.

Based on the required number of servers, the script dynamically adjusts the infrastructure by scaling up or down the number of servers accordingly. It updates the hosts file and config file to reflect the changes. Additionally, it performs a ping operation to verify the reachability of all hosts within the infrastructure.

In the first loop, Flask is installed on the backend webservers regardless of the number of servers. From the second loop onwards, if any new servers are deployed, the script updates the config files and hosts file accordingly. It installs Flask on the newly built servers and updates the backend servers in the Haproxy server configuration. If any servers go down during the process, only the Haproxy server is updated.

Once the required number of servers is present according to the "server.conf" file, the script sleeps for the remaining time within the 30-second interval before starting the next loop.

Furthermore, uWSGI application server is installed, Flask applications are launched, and Nginx is configured to serve the applications. Haproxy is configured in the frontend server to listen to HTTP requests and perform round-robin load balancing to the backend servers.

To check the HTTP request in the Haproxy server, you can use the curl command on the command line and print the output. The specific example of the output would depend on the specific details of the application and the expected response from the server.

In order to ensure high availability and eliminate single points of failure, two Haproxy servers are deployed for the Frontend server. Keepalived is employed to establish a high-availability cluster between the two Haproxy nodes. If one node becomes unavailable, the other node seamlessly takes over the service, ensuring continuous availability of the Frontend service. By using Keepalived in conjunction with the dual Haproxy setup, the infrastructure achieves redundancy and fault tolerance, preventing service disruptions in the event of a node failure. This configuration enhances the overall reliability and availability of the backend servers' load balancing capabilities.

```
TASK [test-1] ******
ok: [haproxy]

TASK [debug] ************************
ok: [haproxy] => {"msg":
"21:24:17 Serving from dev5 (*.*.*.*)\n"
}

TASK [test-2] ***
ok: [haproxy]
```

```
TASK [debug] ******************
ok: [jyoo_haproxy] => {"msg":
"21:24:19 Serving from dev4(*.*.*.*)\n"
}
```

SNMP (Simple Network Management Protocol) is configured on the backend servers to respond to UDP requests, the frontend servers are set up with Nginx to listen for UDP requests and forward them to the backend servers. The output of the SNMP walk command will display the results retrieved from the backend servers, providing information about the network devices and their corresponding SNMP-enabled metrics. The specific output will vary depending on the SNMP configuration and the devices being monitored, but it typically includes information such as device names, system statuses, interface statistics, and other SNMP-manageable entities.

```
TASK [checking snmpwalk -4] ***
changed: [jyoo_haproxy]

TASK [debug] *****************
ok: [jyoo_haproxy] => {
    "output.stdout_lines": [
        "iso.3.6.1.2.1.1.1.0 = STRING:
        \"Linux dev2 5.4.0-26-generic
        #30-Ubuntu SMP Mon Apr 20 16:58:30
        UTC 2020 x86_64\""
    ]
}

TASK [checking snmpwalk -5] *************
changed: [jyoo_haproxy]

TASK [debug] ******************
ok: [jyoo_haproxy] => {
    "output.stdout_lines": [
        "iso.3.6.1.2.1.1.1.0 = STRING:
        \"Linux dev1 5.4.0-26-generic
        #30-Ubuntu SMP Mon Apr 20 16:58:30
        UTC 2020 x86_64\""
    ]
}
```

3) Cleanup: In the final stage, Ansible is utilized to perform various tasks, including the removal of allocated floating IPs, deletion of servers created in stages one and two, elimination of the subnet, router, and network created in the initial stage, as well as the deletion of the key-name used for SSH access. These actions effectively dismantle the OpenStack infrastructure, ensuring the complete cleanup of all resources associated with the project.

4) Motivation of design: OpenStack provides infrastructure blueprints, API endpoints, and Heat to enable infrastructure-as-a-service delivery. Ansible, in conjunction with OpenStack Heat, offers an agentless framework for configuring, deploying, and automating complex application software stacks in the cloud. The integration of Ansible and OpenStack enables the seamless deployment of comprehensive application stacks. Furthermore, Ansible's capabilities extend beyond OpenStack deployment, allowing for the provisioning, configuration, and deployment of cloud applications and services using its powerful modules. With Ansible's module-based approach, every layer of the infrastructure can be efficiently managed, and the simplicity and flexibility of Ansible make it an ideal tool for cloud operators, developers, and users. Additionally, Ansible utilizes SSH for system management, eliminating the need for deploying remote daemons or agents. Its design allows for creating and managing any number of servers using loops, leveraging playbooks for efficient management of multiple servers, and organizing tasks into reusable roles. Ansible's human-readable and intuitive syntax simplifies the creation and maintenance of deployment scripts, while its comprehensive module library provides seamless integration with complex OpenStack services.

5) Alternatives: As alternatives to Ansible for the IaaS project, Python and shell scripting can be employed. Python enables the creation of instances and networks, leveraging its extensive libraries and frameworks for programmatic control of cloud resources. Shell scripting provides a powerful approach to automate tasks and execute OpenStack command-line tools for operations such as ping tests and serving Flask applications. Both options offer flexibility and control over infrastructure deployment, catering to specific requirements and allowing customization in the process.

B. Performance

To evaluate the performance, the "apache2-utils" package was installed on an Ubuntu system. A load test was conducted using the "ab" (ApacheBench) command with 10,000 requests and a concurrency of 100 requests at a time. The test was performed on the floating IP of the Haproxy server, and the mean values of response times were observed to be similar with slight fluctuations. Notably, as the number of nodes increased from 1 to 4 and 5, the standard deviation decreased, indicating improved stability and consistency in response times. The results are visually represented in figures labeled as fig1, fig2, fig3, fig4, fig5 corresponding to 1 node, 2 nodes, 3 nodes, 4 nodes and 5 nodes, respectively.

As the number of backend servers increases, there is a consistent decrease observed in the processing time, waiting time, and connection time as indicated by the benchmarking tool results. This improvement can be attributed to the distribution of workload across multiple servers, enabling parallel processing and reducing the overall load on individual servers.

C. Management of solution

Based on the project specifications, which involve servers with 2 cores, 6 GB RAM, and a download speed of 340 Mbps, it has been observed that a single backend server can handle 10,000 requests with a concurrency of 100 without any failures. However, from the above figures, it can be inferred that when simulating 10,000 requests with 100 requests with three backend servers using a benchmarking tool, the average

```
Concurrency Level:      100
Time taken for tests:   83.692 seconds
Complete requests:      10000
Failed requests:        13
   (Connect: 0, Receive: 0, Length: 13, Exceptions: 0)
Non-2xx responses:      15
Total transferred:      1821129 bytes
HTML transferred:       481026 bytes
Requests per second:    119.49 [#/sec] (mean)
Time per request:       836.917 [ms] (mean)
Time per request:       8.369 [ms] (mean, across all concurrent requests)
Transfer rate:          21.25 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  483 320.7    407    2881
Processing:    14  261 303.3    204    6661
Waiting:       13  206 300.5    148    6661
Total:         33  744 491.1    670    8218

Percentage of the requests served within a certain time (ms)
  50%    670
  66%    797
  75%    859
  80%    896
  90%   1081
  95%   1258
  98%   2637
  99%   2891
 100%   8218 (longest request)
```

Fig. 1. number of nodes is 1

```
Server Software:        gunicorn
Server Hostname:        188.240.222.15
Server Port:            5000

Document Path:          /
Document Length:        48 bytes

Concurrency Level:      50
Time taken for tests:   70.885 seconds
Complete requests:      10000
Failed requests:        12
   (Connect: 0, Receive: 0, Length: 12, Exceptions: 0)
Non-2xx responses:      12
Total transferred:      1821886 bytes
HTML transferred:       481080 bytes
Requests per second:    141.07 [#/sec] (mean)
Time per request:       354.424 [ms] (mean)
Time per request:       7.088 [ms] (mean, across all concurrent requests)
Transfer rate:          25.10 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  246 173.8    228    1643
Processing:    12   69 133.6     40    5326
Waiting:       12   56 134.3     29    5449
Total:         30  316 216.8    301    5326
```

Fig. 2. number of nodes is 2

```
Server Software:        gunicorn
Server Hostname:        188.240.222.15
Server Port:            5000

Document Path:          /
Document Length:        48 bytes

Concurrency Level:      100
Time taken for tests:   6.559 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      1820000 bytes
HTML transferred:       480000 bytes
Requests per second:    1524.71 [#/sec] (mean)
Time per request:       65.586 [ms] (mean)
Time per request:       0.656 [ms] (mean, across all concurrent requests)
Transfer rate:          270.99 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       11   39 105.1     19    1267
Processing:    12   25  23.2     18     188
Waiting:       11   22  19.2     17     188
Total:         24   63 112.5     36    1351
```

Fig. 3. number of nodes is 3

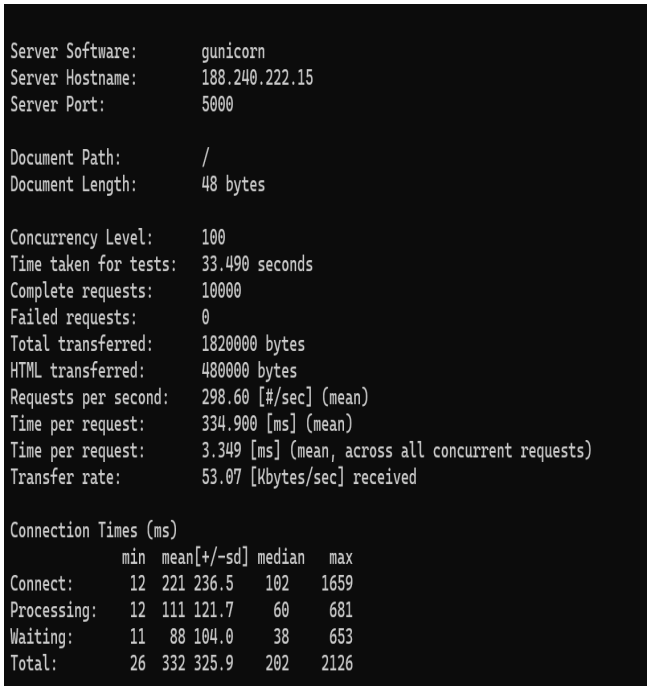
```
Server Software:        gunicorn
Server Hostname:        188.240.222.15
Server Port:            5000

Document Path:          /
Document Length:        48 bytes

Concurrency Level:      100
Time taken for tests:   24.870 seconds
Complete requests:      10000
Failed requests:        7
   (Connect: 0, Receive: 0, Length: 7, Exceptions: 0)
Non-2xx responses:      7
Total transferred:      1821085 bytes
HTML transferred:       480626 bytes
Requests per second:    402.09 [#/sec] (mean)
Time per request:       248.702 [ms] (mean)
Time per request:       2.487 [ms] (mean, across all concurrent requests)
Transfer rate:          71.51 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  137 185.9     30    1439
Processing:    12   62 120.5     23    5543
Waiting:       11   52 117.5     21    5671
Total:         26  199 246.9     52    5543
```

Fig. 4. number of nodes is 4



```
Server Software:        gunicorn
Server Hostname:        188.240.222.15
Server Port:            5000

Document Path:          /
Document Length:        48 bytes

Concurrency Level:      100
Time taken for tests:   33.490 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      1820000 bytes
HTML transferred:       480000 bytes
Requests per second:    298.60 [#/sec] (mean)
Time per request:       334.900 [ms] (mean)
Time per request:       3.349 [ms] (mean, across all concurrent requests)
Transfer rate:          53.07 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       12  221 236.5    102    1659
Processing:    12  111 121.7     60     681
Waiting:       11   88 104.0     38     653
Total:         26  332 325.9    202    2126
```

Fig. 5. number of nodes is 5

time taken for processing these requests is less , connection, rocessing, waiting time and standard deviation values are good comapred to other experiments with differnet backend servers.

Based on this information, it can be concluded that to handle 100 users, three backend servers would be required. Similarly, to handle 1,000 users, 30 backend servers would be necessary, and for 10,000 users, approximately 300 backend servers would be needed. These calculations are based on the observed performance metrics and the requirement of maintaining acceptable response times for the given number of users

D. Solution Operation

Operating a service from multiple locations using OpenStack with Ansible can pose challenges and potential problems regarding networking and service performance. These challenges include network latency, limited network bandwidth, data synchronization complexities, network security concerns, and the need for high availability and redundancy. Among these challenges, network latency and bandwidth limitations have a significant impact on service performance. Ensuring synchronized data across locations, maintaining network security, and implementing robust failover mechanisms are crucial. The impact of these challenges on service performance varies depending on factors like the service nature, data synchronization requirements, network infrastructure, and distance between locations. Mitigation strategies involve careful network architecture design, geographic resource distribution, effective load balancing, and failover mechanisms. Optimization techniques such as performance monitoring and capacity planning, are essential for maintaining service performance in a multi-location setup.

REFERENCES

[1] Factors affecting OpenStack deployment — operations-guide 2013.2.1.dev1189 documentation.