University of Zurich UZH

VERTIEFUNGSARBEIT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Prototypical Implementation of a Cloud Experimentation Environment by an OpenStack Fake Driver

Stephan Mannhart
Steinach, Switzerland
Student ID: 11-917-515

Supervisor: Patrick G. Poullie, Dr. Thomas Bocek
Date of Submission: July 1, 2015

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Vertiefungsarbeit
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Contents

Abstract

OpenStack is a powerful software toolkit to manage production clouds but when it comes to simulating and experimenting with different cloud setups, it doesn't provide a ready to use solution. However, trying out different hypervisor and virtual machine ratios as well as different resource consumptions can lead to interesting insights about how to increase the effectiveness of a cloud. This paper shows the installation and configuration of an OpenStack based experimentation environment with virtualised compute hosts and faked virtual machines.

Keywords: *OpenStack, LXC, fake driver, cloud experimentation environment*

I. Introduction

This paper is structured into three main parts: First I will provide insight into the cloud experimentation environment that has been set up to serve as an interface for the simulations. After that I will focus on performance limitations in regards to the used approach and finally, I will discuss the chosen concept of implementing a cloud runtime resource allocation (CRRA) simulation into the cloud experimentation environment.

II. Cloud Experimentation Environment

The cloud experimentation environment (CEE) consists of the OpenStack "set of software tools for building and managing cloud computing platforms" [1], LXC containers for the various OpenStack compute nodes and the fake driver handling all VM operations. Following, I discuss all mentioned components of the environment in more detail to provide a thorough overview.

II.1. OpenStack Architecture

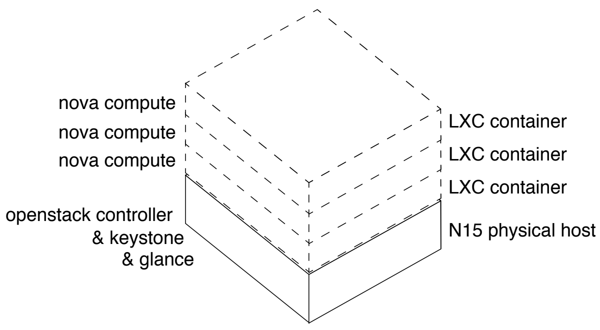


Figure 1: CEE OpenStack Configuration Overview

In Figure 1 you can see the general OpenStack configuration that I chose for the physical host N15 where I set up the CEE. OpenStack consists of a number of services that need to be installed on different hosts.

horizon The dashboard that can be accessed through a web browser to simplify OpenStack configuration tasks and provide an intuitive user interface to manage VMs.

nova The compute service that manages creating, scheduling and destroying VMs.

neutron The new network service that was introduced with the Folsom release of OpenStack. This service should eventually replace the legacy "nova-network" service [3]. This service manages the network connections for devices that are managed by other OpenStack services, like VMs. Since the CEE does not use real VMs, the legacy nova-network service suffices.

keystone The identity service that handles authentication and authorisation for all OpenStack services.

glance The image service that manages virtual machine disk images to be used by the nova compute service.

The CEE discussed in this paper uses the listed services. Additional unused services include swift for unstructured data management, cinder for block storage, ceilometer for monitoring and metering, heat for cloud orchestration and trove for database-as-a-service functionality.

The main host is called "controller node" [2] and runs the OpenStack base-installation as well as the keystone and glance services. In order to allow me to use multiple compute nodes to host the hypervisor part of OpenStack, I used LXC containers [4]. Each LXC container runs its own nova compute service to serve as a hypervisor for the OpenStack cloud. The controller sees the LXC containers as real compute hosts since the LXC virtualisation allows for a complete sandboxing of the compute host to simulate a real host.

II.2. OpenStack Installation

The whole installation was done for the Juno release of OpenStack with the help of the official OpenStack installation guide for Ubuntu 14.04 [2]. The guide provides step-by-step instructions to set up an OpenStack installation with a controller node and one compute node. The installation required 10 individual passwords to set up proper authentication for all services. The most important password is the keystone password for the admin tenant. Tenants are "containers used to group or isolate resources. Tenants also group or isolate identity objects" [5]. The admin tenant allows full access to the nova compute service.

Initial setup of the OpenStack installation included preparing the SQL database which is used by most OpenStack services to store their information and setting up the controller node as an NTP server for the compute nodes to synchronise time with. MariaDB was used as an SQL server as it was recommended by the installation guide.

II.2.1 Identity Service

The use of the identity service and its authentication can be simplified by using a small bash script that can be sourced to set the proper username, password and authentication URL for the identity service as environment variables.

```
export OS_TENANT_NAME=admin
export OS_USERNAME=admin
export OS_PASSWORD=
    ADMIN_PASSWORD
export OS_AUTH_URL=http://
    controller:35357/v2.0
```

Listing 1: Identity Settings

Listing 1 shows the bash script that can be sourced with `$ source identity_settings.sh` to set up the authentication variables to use the command line clients in OpenStack without needing to supply the authentication data as parameters for every command.

II.2.2 Image Service

The image service provides an interface for nova compute to access the VM images it needs. The authentication works through keystone and is set up permanently in the glance configuration file. In order to properly use the glance image service, I created an initial image using CirrOS [6] that I later used as the main image to start new instances since all VMs will only be simulated by the fake driver and the image will only be needed for issuing a valid instance-boot command.

II.2.3 Compute Service

To manage hypervisors for OpenStack compute purposes, the nova compute service is needed. The service has to be installed in two steps: First step is to install the compute service parts for the controller node which consist of:

nova-api To handle user API-requests to the compute service.

nova-cert To manage certificates.

nova-conductor To mitigate between nova services and the underlying SQL database.

nova-scheduler To schedule new VM instances to the correct compute host.

nova-consoleauth, nova-novncproxy To allow authentication for console access like VNC to VM instances.

python-novaclient A command line client to directly interface with the nova API to execute nova compute commands. This client was heavily used to build the CEE and to test the performance properties of the CEE.

The second step is to install the nova compute services that are needed on the individual compute hosts.

II.2.4 Initial LXC Setup

The LXC-based compute hosts only needed the setup of the nova-compute service to accept VM instances that were scheduled by the controller and to successfully manage these instances throughout their lifecycle.

Since the LXC compute hosts deviated from normal compute nodes in that they themselves are virtualised, the initial installation did not work as outlined in the installation instructions [2]. After thoroughly analysing the nova-compute upstart log files in `/var/log/upstart/nova-compute` [7], I realised that the nova compute service tries to load the kernel module"nbd" for network block devices[8] in its upstart script located in `/etc/init/nova-compute.conf` . Since LXC containers are not allowed to load kernel modules, the default upstart script failed and I needed to remove the kernel module loading. This presents no problem since no real VMs are running on the compute host and therefore network block devices are not needed. It would however be possible to still use the nbd kernel module within an LXC container by loading it on the LXC container's host and since the LXC container shares the kernel with the host, the module would already be loaded when the LXC container starts [9].

To simplify the IP address assignment for large amounts of LXC containers, the dnsmasq DHCP server [10] was used to automatically assign IP addresses based on the host name. For each host, an entry with the form `dhcp-host=hostname,ip_address` was set up in the file `/etc/lxc/dnsmasq.conf` on the controller node. This allows for a convenient centralised IP management of a large amount of LXC containers.

While testing the CEE, I ran into the problem of dnsmasq leases that did not expire fast enough and caused wrong IP address assignments for LXC containers. This was mostly due to LXC containers having been removed and new ones being created with the same name for test purposes. The issued leases for dnsmasq are stored in `/var/lib/misc/dnsmasq.lxcbr0.leases` . To remove old leases, they can just be removed from the leases file. To make these changes effective, the lxc-net service has to be restarted with `# service lxc-net stop` `# service lxc-net start` . It is important to restart lxc-net with stop followed by start instead of restart because there is a known bug that prevents restart from correctly restarting dnsmasq [11]. The CEE in it's final configuration does not exhibit this problem since all hosts are pre-created and only get started or stopped according to the needed amount of hosts, which does not create the discussed lease problem.

II.2.5 LXC Usage

I created the initial LXC container from a 64bit Ubuntu 14.04 image that was directly downloaded through the lxc CLI with `$ sudo lxc-create -t download -n nova_0` to create an LXC container with the name nova_0. This container was then customised to work as a generic nova compute host template to be cloned to generate a multitude of containers.

```bash
#!/bin/bash

count=`ls /var/lib/lxc | tail -1 | tr -d "nova_"`
new_nova_id=$((count + 1));
new_nova_name='nova_'$new_nova_id;

lxc-clone -o nova_0 -n $new_nova_name

# Add new dnsmasq resolver
new_nova_ip_ending=$((100 + new_nova_id));
echo "dhcp-host=nova_$new_nova_id,10.0.3.$new_nova_ip_ending" >> /etc/lxc/dnsmasq.conf
service lxc-net restart

# Change ip in nova.conf
sed -i 's/IP/10.0.3.'$new_nova_ip_ending'/g' /var/lib/lxc/$new_nova_name/rootfs/etc/nova/nova.conf
```

Listing 2: LXC Cloning Script

The bash script in Listing 2 simplifies the cloning of the nova_0 template by automatically creating an LXC container with incremented host name of the form nova_number and adding the necessary host entry to the dnsmasq configuration file as well as setting up the correct IP address in the new hosts nova configuration.

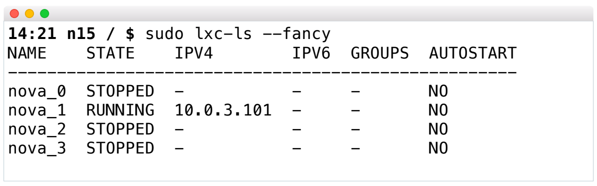

```
14:21 n15 / $ sudo lxc-ls --fancy
NAME    STATE    IPV4        IPV6  GROUPS  AUTOSTART
----------------------------------------------------
nova_0  STOPPED  -           -     -       NO
nova_1  RUNNING  10.0.3.101  -     -       NO
nova_2  STOPPED  -           -     -       NO
nova_3  STOPPED  -           -     -       NO
```

Figure 2: LXC container list

To list the existing LXC containers, you can use `$ sudo lxc-ls --fancy` which produces an output similar to Figure 2 with the states and IP addresses of all LXC containers that are currently available on the system.

To start container nova_1, execute `$ sudo lxc-start -dn nova_1` to start the container as a daemon. To access container nova_1, use `$ sudo lxc-attach -n nova_1` to open a shell connection as user root. To finally stop container nova_1, you can use `$ sudo lxc-stop -n nova_1` .

II.2.6 Networking Component

As described in section II.1, the CEE uses nova-network as the networking component for the nova services. I primarily made this choice because no real VMs are present and therefore providing a fully featured networking component like neutron would have been pointless. The nova-network service is however still needed as I discovered: When starting new instances, the network state of the instance is requested from the network service. With no network service available, the request will time out and cause the VM to go into an ERROR state from which it can neither recover nor be destroyed. A workaround for this problem was to directly destroy the instance through the MySQL database [12]. To simplify this process, I wrote a small SQL procedure that can directly be called with an instance-UUID: `destroy_instance(instance_uuid)` . This procedure could be used for different scenarios where an instance can't be recovered from an ERROR state but is no longer needed for the outlined network problem as the nova-network service solves this.

II.2.7 Fake Driver

The central part of the nova-compute service is the virtualisation driver to interface with the hypervisor. There are many compute drivers available that can be used [13]. The default driver is the libvirt driver which is also the driver that has been tested the most [14]. The libvirt driver supports a multitude of hypervisors [15] and would therefore be a good choice as a basic driver. The CEE is actually able to fully use the libvirt driver to start real VMs inside the LXC-based compute host. This was very helpful to observe realistic behaviour in regards to scheduling or diagnostic information.

In order to allow useful simulations in the CEE, a fake driver needs to be used to bypass the actual hypervisor and just fake VM actions like creating, starting, stopping or getting diagnostic information. To set up the compute hosts for use with the fake driver, `compute_driver=fake.FakeDriver` has to be set in `/etc/nova/nova-compute.conf` . After restarting the nova-compute service with `# service nova-compute restart` , the changes will take effect and the fake driver will now handle all VMs.

To make sure that the fake driver won't be constrained by the actual resources that are available on the physical machine, the quotas for each tenant can be adjusted to allow unlimited amounts of memory, cpu cores and disk space to be used by VMs within the tenant [16]. To change the quotas, the admin tenant id has to be requested with `# keystone tenant-list` to use it to set the quotas: `# nova quota-update --instances -1 --cores -1 --ram -1 --fixed-ips -1 --floating-ips -1 tenant_id` . Since OpenStack is written in Python and is open source, the fake driver can be changed to accommodate our needs. The python source for the fake driver is located in `/usr/lib/python2.7/dist-packages/nova/virt/fake.py` . Since every LXC compute host will need this source, changes to the driver would need to be installed separately on each LXC container. To circumvent this, I set up a shared folder on the physical machine that will be mounted on each LXC container [17]. I copied all relevant Python source files to `/opt/nova` on host N15 and set the mount point in the shared LXC configuration file `/usr/share/lxc/config/ubuntu.common.conf` which is read by all LXC containers.

II.3. Nova Compute Command-line Client

The compute command line client [18] can be used to directly talk to the nova API.

There are some essential commands to manage VMs and get diagnostic information about compute hosts and VMs that I will list in the following sections.

II.3.1 VM Management Commands

Create VM To create a new VM, you first have to get the flavor-id through `# nova flavor-list` and the image-id of the CirrOS image through `# nova image-list` . With both ids, you can then execute `# nova boot --flavor flavor_id --image image_id vm_name`

Start VM `# nova start vm_name`

Stop VM `# nova stop vm_name`

II.3.2 VM Information Commands

List all VMs `# nova list`

Show high-level info about a VM `# nova show vm_name`

Show VM diagnostics `# nova diagnostics vm_name`

Show VMs for host nova_1 `# nova hypervisor-servers nova_1`

List diagnostics for all VMs on nova_1 `# nova host-describe nova_1`

II.3.3 Host Information Commands

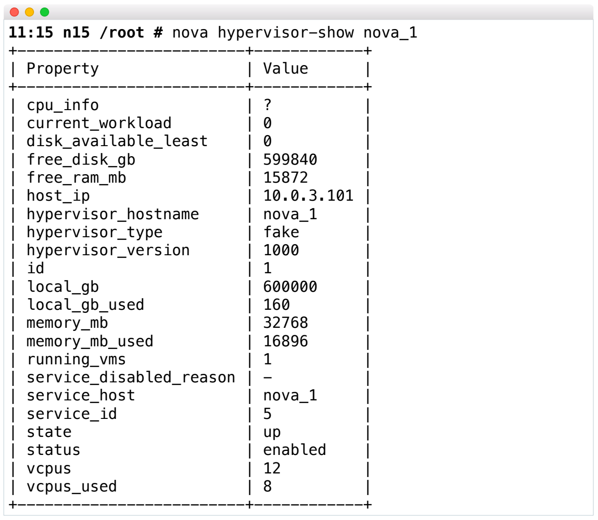
Show all OpenStack services `# nova service-list`

Show services of all hosts `# nova host-list`

Show resources of host nova_1 `# nova hypervisor-show nova_1`

II.3.4 Diagnostic Data for Simulations

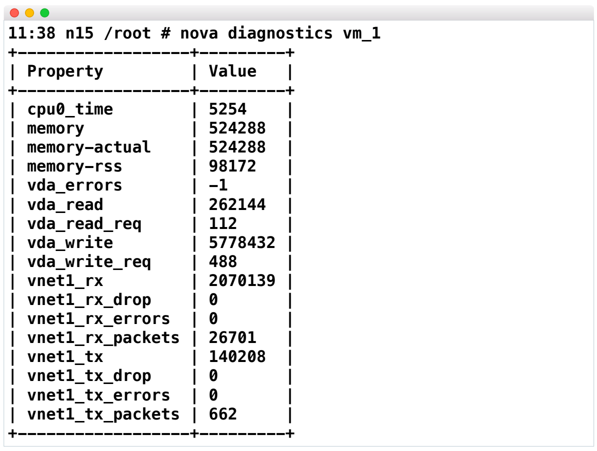
In order to perform useful simulations, there needs to be a way to obtain diagnostic information from the CEE. This can be achieved through the outlined command-line client commands from the last two sections. I would like to highlight two of the most useful commands in regards to simulations. When simulating resource usage in the CEE, the most interesting metrics include: **CPU time**, **memory usage**, **disk usage** and **network usage** of the individual VMs as well as the compute hosts themselves.



```
11:15 n15 /root # nova hypervisor-show nova_1
+-------------------------+------------+
| Property                | Value      |
+-------------------------+------------+
| cpu_info                | ?          |
| current_workload        | 0          |
| disk_available_least    | 0          |
| free_disk_gb            | 599840     |
| free_ram_mb             | 15872      |
| host_ip                 | 10.0.3.101 |
| hypervisor_hostname     | nova_1     |
| hypervisor_type         | fake       |
| hypervisor_version      | 1000       |
| id                      | 1          |
| local_gb                | 600000     |
| local_gb_used           | 160        |
| memory_mb               | 32768      |
| memory_mb_used          | 16896      |
| running_vms             | 1          |
| service_disabled_reason | -          |
| service_host            | nova_1     |
| service_id              | 5          |
| state                   | up         |
| status                  | enabled    |
| vcpus                   | 12         |
| vcpus_used              | 8          |
+-------------------------+------------+
```

Figure 3: Hypervisor Stats

Figure 3 shows the output of the `# nova hypervisor-show nova_1` command to display hypervisor information about the compute host nova_1. We can get detailed information about disk- as well as memory-usage from the host as well as a general overview about VCPU usage.



```
11:38 n15 /root # nova diagnostics vm_1
+------------------+---------+
| Property         | Value   |
+------------------+---------+
| cpu0_time        | 5254    |
| memory           | 524288  |
| memory-actual    | 524288  |
| memory-rss       | 98172   |
| vda_errors       | -1      |
| vda_read         | 262144  |
| vda_read_req     | 112     |
| vda_write        | 5778432 |
| vda_write_req    | 488     |
| vnet1_rx         | 2070139 |
| vnet1_rx_drop    | 0       |
| vnet1_rx_errors  | 0       |
| vnet1_rx_packets | 26701   |
| vnet1_tx         | 140208  |
| vnet1_tx_drop    | 0       |
| vnet1_tx_errors  | 0       |
| vnet1_tx_packets | 662     |
+------------------+---------+
```

Figure 4: VM Diagnostics

To get information about the VMs, Figure 4 shows the output of the `# nova diagnostics vm_1` command to obtain diagnostic information about VM "vm_1". These information are particularly interesting as they show CPU time, memory usage, disk usage and network usage. The CPU time can be used to calculate the current CPU usage in percentage with the total cpu time from `/proc/stat` [19].

III. Performance Limitations

An important aspect of the CEE is the ability to simulate big loads with many hosts as well as many VMs running. To test the performance of the CEE, I conducted a set of tests and gathered metrics about the physical host to determine the maximum load that could be simulated.

III.1. Compute Host Limitations

The physical host limits the amount of LXC containers that can be created since every container needs a certain amount of disk space as well as memory to run.

nova_1	nova_2	nova_3	nova_4	nova_5
213.6	249.9	249.9	201.3	202.1

Table 1: LXC Container Maximum Memory Usage in MB

Table 1 shows the maximum memory usage for the initial 5 LXC test-containers. To obtain these metrics, I needed to bind the cgroup mounts to `/sys/fs/cgroup` for all LXC containers [20]. After that, I was able to obtain the maximum amount of memory allocated to each container during it's lifetime by viewing the content of `/sys/fs/cgroup/memory/lxc/nova_*/memory.max_usage_in_bytes`. I calculated an average memory usage of 223.3 MB which means the physical host N15 with a total of 62 GB of memory would have enough memory to run 273 LXC containers considering that N15 itself needs an additional 3 GB for it's own operating system.

The bottleneck in respect to the amount of LXC containers that can be sustained on N15 is the amount of disk space that an LXC container needs. Since all containers are clones of container "nova_0" they share the same amount of disk space they need to work. All files for container "nova_0" are stored in `/var/lib/lxc/nova_0/rootfs` which amounts to 994 MB. Since N15 only has 67.3 GB of usable disk space, it could only run a maximum of 69 LXC-based compute hosts.

III.2. VM Limitations

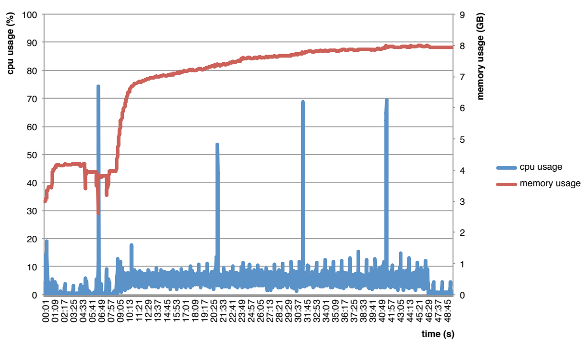


Figure 5: Memory and CPU Usage for 1000 VMs

In regards to VMs, the CEE turned out to be much more capable. In an attempt to stress the physical host, I started 1000 VMs with one compute node (nova_1) running. I used dstat to log the CPU and memory usage [21] that resulted in Figure 5 which clearly shows that the amount of memory needed to support the VMs initially increases but starts to plateau around the 30 minute mark. The memory usage never exceeds 8 GB and it has to be noted that the test was run on a freshly booted machine.

The behaviour in respect to CPU usage is also to be expected since the VMs are running on a faked hypervisor and therefore consume no CPU time. The individual spikes every 10 minutes could be caused by other background processes and do not have such a huge significance as they only last for a single second.

IV. CRRA Implementation

The cloud runtime resource allocation or CRRA is an important addition to the existing CEE that allows for simulated workloads to be assigned to the VMs. Since the VMs do not actually exist and are only faked by the fake hypervisor, CRRA becomes an important tool to conduct simulations within the CEE. The nova command-line client that was discussed in section II.3 allows detailed information-gathering with respect to compute hosts as well as VMs but in order to understand where this diagnostic data comes from, we need to take a look at how scheduling in OpenStack works.

IV.1. OpenStack Scheduling

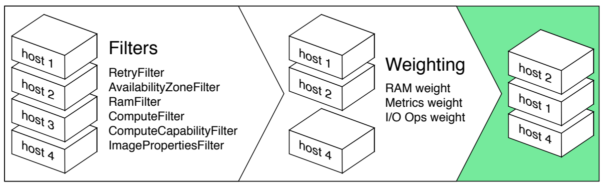


Figure 6: OpenStack Scheduling

As Figure 6 shows, scheduling takes three steps in OpenStack: First, all hosts are filtered by a predefined set of filters, then they are weighted and finally, a host is chosen from the resulting list of potential hosts to run the VM [22].

IV.1.1 Resource Consumption Data

In order to apply the filters and weights that will be described in the following two sections, the nova scheduler needs resource consumption data of all hosts. This data is directly accessed from the database by the scheduler and stored there by each host in a default time interval of 60 seconds as defined in `/usr/lib/python2.7/dist-packages/nova/openstack/common/periodic_task.py` [27].

IV.1.2 Filtering

The first step in scheduling is filtering the complete list of all nova compute hosts. The set of filters to be used is defined in `/etc/nova/nova.conf` [23]. The default set of filters includes:

RetryFilter Skips nodes that have already been attempted for scheduling.

AvailabilityZoneFilter Makes sure the VM gets started on a node that is in the availability zone as described in the VM instance properties. Availability zones are nodes on a single cloud site that are somehow isolated from the other nodes, for example by distinct power supplies [24].

RamFilter Chooses nodes that have enough free RAM. Since the ratio of physical to virtual RAM isn't always 1:1, the default nova.conf setting `ram_allocation_ratio = 1.5` is used to allow overcommitting of RAM. For example, if a compute node only has 1 GB RAM, it is still able to run VMs that require 1.5 GB. This makes sense since the allocation does not directly result in the complete consumption of the allocated RAM. If the VM that is set to require 1.5 GB RAM actually only uses a maximum of 1 GB RAM, then the host it runs on can easily support it [23].

ComputeFilter Filters nodes that are either disabled or that are offline.

ComputeCapabilitiesFilter Checks if nodes satisfy additional capabilities that could be needed by the instance. Extra capabilities can also be set through the flavor as `key:value` pairs.

ImagePropertiesFilter Filters nodes that don't satisfy the architecture, hypervisor type or virtual machine mode that was set for the VM's image.

IV.1.3 Weighting

After the initial host list has been run through all filters, the list is weighted according to the following ruleset: Three weighers are applied to the host list. After a weight has been assigned to each host, the weights are normalised by scaling each weight to the range between the maximum and minimum weight recorded: $\frac{weight-minimum}{range}$. Each normalised weight-value is then multiplied by the defined weight multiplier. Following three weighers are available:

RAMWeigher The more free RAM, the higher the weight. The weight multiplier is set in nova.conf as `ram_weight_multiplier` and defaults to 1.0 [23].

MetricsWeigher Different metrics of the compute host can be weighed according to defined weighting settings that are configured through nova.conf as `weight_setting` with the form `metricName=ratio` . As an example, if hosts with low previous load (low cpu utilisation) are wanted, a weight setting of the following form could be used: `weight_setting = cpu.percent=-1.0"` [25].

IoOpsWeigher The amount of input and output operations specify the weight for the host. More I/O operations give the host a higher weight. The default weight multiplier is -1.0, setting a preference on light workload compute hosts. To set a preference on heavy workload compute hosts, a positive multiplier has to be chosen. The weight multiplier can be set in nova.conf as `io_ops_weight_multiplier` [26].

From the weighted list, a potential host is chosen randomly. The choice is however influenced by the nova.conf variable `scheduler_host_subset_size` . If this variable is bigger than the size of the host list, the final host will be randomly chosen among all possible hosts from the list. If the variable is smaller than 1, it will be set to 1 which results in the first host from the list to be chosen as the final host. Every other value in between makes the scheduler choose a host randomly from a subset of the weighted host list.

IV.2. Integrating CRRA

The main problem concerning the CRRA integration into the CEE was the communication between the CRRA cloudlet and the numerous LXC-based nova compute hosts. The CRRA cloudlet itself is written in Java [28] but the fake driver it needs to communicate with is written in Python. To solve the communication problem, I looked into different approaches for interprocess communication. There are generally two good ways to do interprocess communication on a unix based system: Either through named pipes or through sockets [29]. After trying out named pipes with a small Python to Python prototype that I developed, I realised that it is not very well suited for fast message exchanges [30] and there is also the problem of named pipes that are missing, have already been created or have the wrong permissions set. I finally decided to use TCP sockets for the interprocess communication [31]. The main advantage of this approach is the way I can run the CRRA cloudlet as a single instance on the physical host without needing to create multiple cloudlets for each LXC container. This is based on the fact that each LXC container has it's own IP and can therefore be directly contacted over TCP. Centralising the CRRA component of the CEE eliminates the need of a time synchronisation between simulations as they all run within the same simulation instance.

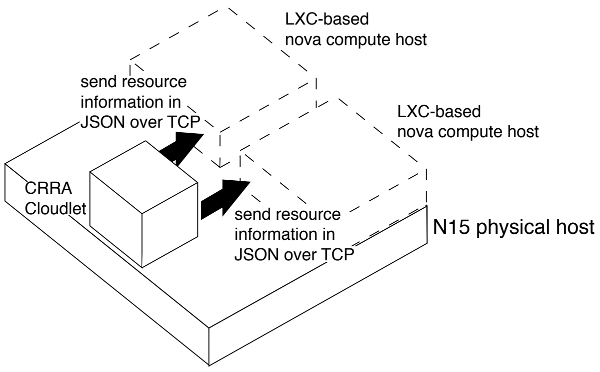


Figure 7: CRRA Cloudlet in the CEE

Figure 7 shows the overall architecture for the communication between CRRA cloudlet and OpenStack. The CRRA cloudlet directly opens a TCP socket to each LXC host and sends it updated resource information every second about each VM that is currently running on the compute host. The compute host itself only listens for incoming TCP connections as soon as the fake driver is initialised.

The communication takes place in the port range from 50001 to 50050 with 50 LXC-based compute hosts to avoid collisions with existing TCP services [32]. The resource information is exchanged in JSON because it is a well documented object notation with implementations for Java as well as Python [33] [34].

IV.2.1 Starting Simulations

At the beginning of a simulation, the CRRA cloudlet starts the needed amount of LXC containers to serve as nova compute hosts through the LXC command line client with `lxc-start -dn nova_*` . After the required amount of LXC hosts have been started, the cloudlet informs the LXC hosts through the TCP socket about their resource configurations in order to make sure the scheduling will work realistically based on the resource-limitations as defined by the CRRA cloudlet input. To start a VM instance, the cloudlet can directly communicate with the nova API through curl [35].

The cloudlet then sends a request to all LXC containers through the TCP socket to obtain a list of VMs that have been scheduled to run on the hosts by the nova scheduler. With this information, the cloudlet is now able to start simulating the resource consumptions of all VMs in real time and send the resulting changes in resources to the LXC hosts in JSON over the TCP socket. The cloudlet needs to simulate in real time because the resource consumptions need to be requested directly from the compute hosts through the nova API or nova command-line client. Faster than real-time simulations would result in gaps in the resource consumptions that are communicated to the compute hosts and would present a lacking overall picture in regards to the resource changes.

IV.2.2 Conducting Experiments

With the approach outlined in the last section, a researcher would now be able to directly conduct an experiment in the CEE. The input parameters are the same as for a Cloudsim simulation as the CRRA cloudlet directly starts the compute hosts and VMs according to it's input parameters.

The researcher would however need to set up a capturing environment where the output from the nova command-line client or nova API in regards to resource consumptions of VMs and compute hosts would need to be recorded in order to use the data for further analysis.

IV.2.3 Limitations

The outlined approach is mostly in concept stage as there is not yet a modified CRRA cloudlet available that would be able to communicate with the compute hosts due to time constraints. The fake driver modifications are also still missing but experiments in regards to the TCP socket communications have been made that allowed me to output random resource changes through the fake driver.

V. Conclusions

OpenStack is very powerful to manage and run complex cloud configurations. The basic OpenStack documentation was initially very helpful in overcoming hurdles that presented themselves during the OpenStack installation process I went through. The specific requirement of a fake hypervisor driver and nova compute hosts running inside LXC containers however made the process much more complicated since there are little resources available online in regards to the fake driver as well as running compute hosts inside LXC containers. Based on these constraints, the CEE setup turned out to be much more complicated than initially expected. Due to this fact, the implementation of cloud runtime resource allocation could not be completed but a prototypical implementation concept was developed that is based on individual prototype tests as well as discussions with the CRRA cloudlet creator Patrick Taddei, to ensure a sound concept.

Glossary

Cloudlet A class to model cloud-based application services in Cloudsim [36].

Folsom, Juno Release names for different OpenStack versions.

Hypervisor A software or hardware component to create and run virtual machines.

Mounting To make a storage device accessible through the file system.

N15 The physical host in the CSG testbed that runs the CEE.

Tenant Containers used to group or isolate resources. [5]

Virtual Machine An emulation of a computer system.

Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CRRA	Cloud Runtime Resource Allocation
LXC	Linux Containers
NBD	Network Block Devices
RAM	Random-Access Memory
VCPU	Virtual CPU
VM	Virtual Machine

References

[1] Opensource.com: What is OpenStack?, `http://opensource.com/resources/what-is-openstack`, (visited on 06/27/2015)

[2] OpenStack Foundation: OpenStack Installation Guide for Ubuntu 14.04, `http://docs.openstack.org/juno/install-guide/install/apt/content/`, (visited on 06/27/2015)

[3] OpenStack Foundation: Deprecation of the Nova Network Service, `http://docs.openstack.org/openstack-ops/content/nova-network-deprecation.html`, (visited on 06/27/2015)

[4] Ubuntu Documentation Team: Ubuntu Virtualisation Server Guide, `https://help.ubuntu.com/lts/serverguide/lxc.html`, (visited on 06/27/2015)

[5] OpenStack Foundation: OpenStack Identity Concepts, `http://docs.openstack.org/juno/install-guide/install/apt/content/keystone-concepts.html`, (visited on 06/27/2015)

[6] Canonical Ltd.: CirrOS, https://launchpad.net/cirros, (visited on 06/27/2015)

[7] Ask Openstack: Nova-scheduler Driver Setting Instance to Error State, https://ask.openstack.org/en/question/1279/nova-scheduler-driver-setting-instance-to-error-state/?answer=1408#post-id-1408, (visited on 06/29/2015)

[8] Wikipedia: Network Block Device, https://en.wikipedia.org/wiki/Network_block_device, (visited on 06/29/2015)

[9] Michael Foord: Workaround for modprobe inside LXC, https://bugs.launchpad.net/juju-core/+bug/1353443/comments/3, (visited on 06/29/2015)

[10] OpenStack Foundation: Dnsmasq DHCP Server, http://docs.openstack.org/admin-guide-cloud/content/section_dnsmasq.html, (visited on 06/27/2015)

[11] James Page: Workaround for lxc-net Restart Bug, https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1043588/comments/2, (visited on 06/29/2015)

[12] Remy van Elst: Fix inconsistent Openstack Volumes and Instances from Cinder and Nova via the Database, https://raymii.org/s/articles/Fix_inconsistent_Openstack_volumes_and_instances_from_Cinder_and_Nova_via_the_database.html, (visited on 06/29/2015)

[13] OpenStack Foundation: Nova.conf Configuration Options, http://docs.openstack.org/juno/config-reference/content/list-of-compute-config-options.html#config_table_nova_compute, (visited on 06/29/2015)

[14] OpenStack Foundation: HyperVisorSupportMatrix, https://wiki.openstack.org/wiki/HypervisorSupportMatrix#Compute_Drivers, (visited on 06/29/2015)

[15] RedHat: libvirt.org, http://libvirt.org, (visited on 06/29/2015)

[16] Russell Bryant: OpenStack API Mocker or Simulator, https://ask.openstack.org/en/question/28/openstack-api-mocker-or-simulator/?answer=52#post-id-52, (visited on 06/29/2015)

[17] OpenSuSE: LXC Mount Shared Directory, https://en.opensuse.org/User:Tsu2/LXC_mount_shared_directory, (visited on 06/29/2015)

[18] OpenStack Foundation: Compute Command-Line Client, http://docs.openstack.org/cli-reference/content/novaclient_commands.html, (visited on 06/29/2015)

[19] LinuxHowTos.org: /proc/stat explained, http://www.linuxhowtos.org/System/procstat.htm, (visited on 06/30/2015)

[20] Fabio Kung: Memory Inside Linux Containers, http://fabiokung.com/2014/03/13/memory-inside-linux-containers/#comment-3590, (visited on 06/30/2015)

[21] Russell Barnes: Create a Graph of your System's Performance, http://www.linuxuser.co.uk/tutorials/create-a-graph-of-your-systems-performance/4, (visited on 06/30/2015)

[22] OpenStack Foundation: Filter Scheduler, http://docs.openstack.org/developer/nova/filter_scheduler.html?highlight=scheduler, (visited on 06/30/2015)

[23] OpenStack Foundation: OpenStack Scheduling Configuration Reference, http://docs.openstack.org/juno/config-reference/content/section_compute-scheduler.html, (visited on 06/30/2015)

[24] OpenStack Foundation: Scaling, http://docs.openstack.org/openstack-ops/content/scaling.html#az_s3, (visited on 06/30/2015)

[25] Lianhao Lu, Yingxin Chen: Utilization-based Scheduling in OpenStack* Compute (Nova), https://01.org/sites/default/files/utilization_based_scheduing_in_openstack_compute_nova_1.docx, (visited on 06/30/2015)

[26] OpenStack Foundation: Create Nova Scheduler IO Ops Weighter, http://specs.openstack.org/openstack/nova-specs/specs/juno/approved/io-ops-weight.html, (visited on 06/30/2015)

[27] Donald D. Dugger: Compute Node Update Interval, http://lists.openstack.org/pipermail/openstack/2014-July/008216.html, (visited on 06/30/2015)

[28] Patrick Taddei: Cloudsim-RDA on GitHub, https://github.com/pattad/cloudsim-rda/tree/master/src/main/java/ch/uzh/ifi/csg/cloudsim/rda, (visited on 06/30/2015)

[29] Stackoverflow: Fastest Method for IPC between Java and C/C++, http://stackoverflow.com/questions/2635272/fastest-low-latency-method-for-inter-process-communication-between-java-and-c, (visited on 06/30/2015)

[30] Adam Fraser: IPC between Python and Java, http://stackoverflow.com/a/3804547, (visited on 06/30/2015)

[31] How to connect a Python Client to Java Server with TCP Socket, https://norwied.wordpress.com/2012/04/17/how-to-connect-a-python-client-to-java-server-with-tcp-sockets/, (visited on 06/30/2015)

[32] Wikipedia: List of TCP and UDP Port Numbers, https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers, (visited on 06/30/2015)

[33] Veer Shrivastav: Decoding JSON String in Java, http://stackoverflow.com/a/16586100, (visited on 06/30/2015)

[34] Python Software Foundation: JSON Encoder and Decoder, https://docs.python.org/2/library/json.html, (visited on 06/30/2015)

[35] Damion Parry: Create an OpenStack Instance with just Curl, `https://catn.com/2013/04/23/create-an-openstack-instance-with-just-curl/`, (visited on 06/30/2015)

[36] John G. Michopoulos: What is a Cloudlet in Cloudsim?, `http://www.researchgate.net/post/What_is_a_Cloudlet_in_Cloudsim`, (visited on 06/30/2015)