Dependable Cloud Computing with OpenStack

Johannes Eschrig, Sven Knebel, Nicco Kunzmann

Operating Systems and Middleware Group
Hasso Plattner Institute Potsdam
first.last@student.hpi.de

Offering infrastructures as a service by means of cloud computing is gaining popularity. High availability aspects of these cloud computing systems are of great importance, as outages can be extremely costly. Setting up a cloud computing environment is very complex, thus making dependability testing non trivial. In our work, we introduce a system for installing a virtual OpenStack cloud computing environment and running dependability experiments on it[1]. The installation as well as the experiments are automated in order to achieve reproducible test results as easily as possible. We propose a first selection of experiments for our testing framework and describe the results.

1 Introduction

As cloud computing becomes more and more popular, there are an increasing number of implementations to offer various cloud-service models like infrastructure as a service (IaaS), platform as a service (PaaS) or software as a service (SaaS). While many companies offer commercial solutions like the Amazon Elastic Compute Cloud (EC2) or HP Helion, there are also open source alternatives that can be freely installed and configured to meet the needs of ones projects with respect to the underlying hardware available.

One of the open source variants for achieving a cloud computing system is OpenStack. OpenStack is a cloud software stack which allows for offering infrastructure as a service, almost independent of the underlying hardware setup. OpenStack itself can be seen as a collection of services that can be setup depending on the specifications of the planned use cases. The most important components that OpenStack offers are the networking, virtualization and storage services. Furthermore, it is possible to add further components to an OpenStack installation, e.g., services that handle billing or allow for object storage in the cloud.

This paper will describe the results of the masters project "Dependable Cloud Computing with OpenStack" of the summer term 2015 at the Hasso Plattner Institute Potsdam. An important factor, especially in cloud computing, is dependability. When offering such a service, it should be highly available, meaning that the system should be continuously operational without failing. Therefore, our main task

[1] https://github.com/MasterprojectOpenStack2015/sourcecode

was to analyse dependability mechanisms of OpenStack. To do this, we chose to manually setup a clean OpenStack environment (i.e. none provided by a third party like HP Helion) on which we would be able to run the specific analyses. We turned this manual installation into an automated one in order to simplify and speed up the process of setting up a working OpenStack test environment and making the resulting analyses of dependability reproducible. Since no OpenStack installation is exactly the same, the reproducibility of the results of such analyses is not an easy feat. We tackle this issue by making the test environment for the experiments completely virtual. Thus we circumvent tedious hardware setup, hardware errors that disturb the experiments. This also allows an fast rerun of the experiments and switching off network infrastructure.

2 Related Work

In this chapter we will introduce work related to our masters project. In a first part we will describe work related to OpenStack and its possibilities of installation. The second part will cover the related work to dependability in OpenStack.

2.1 OpenStack Installation

In order to analyse the dependability of OpenStack, it is necessary to install an OpenStack instance. Due to the fact that such an analysis might require a clean and fresh OpenStack installation after each test run, a quick installation is of advantage. Further, independence of underlying hardware is vital to reproduce results. Merging the requirements of an easy and quick installation that achieve reproducible results leads us to look for possibilities to automatically install OpenStack completely in a virtual environment. There are various OpenStack derivatives both commercially and freely available.

HP Helion[2] is available both as a commercial-grade edition and a free-to-license community edition. The latter is available on promotional USB drives given out by HP. The HP Helion community edition installation is made to provide an easy installation routine with little need for configuration by means of such an USB drive. Further, it is possible to install HP Helion as an all-in-one system on virtual machines in addition to deploying it on bare-metal.

DevStack[3] is another possibility for an easy OpenStack installation. It is a development environment for OpenStack. Being designed for development on OpenStack, it is mainly used for an one-node installation of OpenStack. Additionally, DevStack also offers an option for a multi-node setup.

[2] `http://www8.hp.com/us/en/cloud/hphelion-openstack.html`
[3] `http://docs.openstack.org/developer/devstack/`

A manual installation of an OpenStack instance can take very long and be quite cumbersome, depending on the setup one is aiming to achieve. It is therefore of great advantage to automate the installation. As an OpenStack installation is distributed among a number of nodes, using an orchestration tool like Ansible[4] is advisable. Ansible manages nodes using SSH and Python.

openstack-ansible[5] is an existing automated OpenStack installation project, which installs OpenStack on Vagrant virtual machines. We ran the installation script of this project, however encountered some bugs. In order to understand the underlying mechanisms of OpenStack ourselves, we decided to follow a similar approach to openstack-ansible based on KVM virtual machines.

2.2 Evaluating OpenStack Dependability

Due to the complexity and variety of possibilities to set up an OpenStack system, evaluating the dependability of OpenStack in general is no easy task. For this reason, we have decided to make simplifying assumptions about an OpenStack installation and define a test environment on which we can then run dependability experiments. An more general approach is also possible, i.e. building a framework for injecting faults into various OpenStack deployments, as was done for example by [3] or [2]. Both works thereby created a frameworks for injecting faults into OpenStack. [3] follows a similar approach to that of our masters project and uses a virtual environment for the setup of OpenStack, however only implements one simulated failure as proof of concept. [2] on the other hand focuses more on the fault injection aspect, especially targeting service communications, uncovering 23 bugs in two OpenStack versions. In our masters project, we aim to provide a framework for the evaluation of the cloud system OpenStack with the advantage of a fast and easy virtual installation of the system itself and easily extendable experiments for dependability testing.

The previous masters project on OpenStack also gave some insights on fault tolerance of OpenStack in [1], presenting a fault tree based on the high availability setup presented by [4].

3 OpenStack Test Environment

As described in Chapter 2, we tried various possibilities to install an OpenStack system on a virtual environment. In this chapter we outline the challenges we faced with these possibilities, ultimately leading to the decision to create our own

[4] http://www.ansible.com/

[5] https://github.com/openstack-ansible/openstack-ansible

installation routine for a virtual OpenStack environment. Also, we will describe this test environment in detail.

3.1 Existing OpenStack Installation Possibilities

We first tried installing the cloud computing environment HP Helion community edition. It promises an easy installation routine with little need for configuration. Further, it is possible to install HP Helion as an all-in-one system on virtual machines in addition to deploying it on bare-metal. This is an advantage with respect to our requirement of achieving a virtual test environment for reproducible test results. However, even though HP Helion has its advantages for setting up one's own IaaS system for a real use scenario, we came to the conclusion that it is not suitable for our needs. The installation of HP Helion took around 90 minutes on our hardware, which is not feasible for repeated installations. Further, we found that HP Helion does not survive a reboot of the host or the virtual machines it is running on. Fixing this issue would have required understanding the underlying installation scripts, which would still not have been beneficial in understanding OpenStack itself. Additionally, with our limited knowledge of HP Helion, it would have been a challenge to customize the system to fit our needs. We thus concluded that we would not use HP Helion for analyzing the dependability of OpenStack in line with this masters project.

A further option for an OpenStack deployment to work with for dependability testing we considered was DevStack. Due to the nature of the use cases for which DevStack is made, multi-node and high-availability setups are not the main focus, and therefore not documented well enough for us to customize DevStack and use it for dependability analyses. Also, it is not possible to generalize results of dependability analyses run on DevStack to a full OpenStack installation, as DevStack is not designed with real deployment in mind. As a result, we decided to use a full OpenStack installation for running our analyses.

3.2 Specifying our own OpenStack Environment

In order to be able to make the OpenStack test environment installation as easy and quick as possible, so that one can concentrate on the dependability analysis, we chose to install OpenStack virtually. A further advantage of this virtual installation is the reproducibility of experiment results, which is important to be able to make scientific statements about the dependability of OpenStack. We used libvirt[6] to create a number of virtual machines based on simple configuration files and

[6] `http://libvirt.org/`

Ansible[7] to orchestrate the installation of OpenStack on these nodes. The details of this automated installation are described in Chapter 4.

In order to create an useful test environment for dependability experiments, it was necessary to define an architecture. We chose this architecture to be a simplified OpenStack instance, meaning that we focus only the most important of the OpenStack services available. This can be seen as a bottom-up approach, as we focus on evaluating a simpler system than one might encounter when looking at an OpenStack system in production mode. One advantage of this approach is that, due to the simplicity, it is easier to make statements about OpenStack in general, than on one specific system. Libvirt and Ansible allow to add more nodes, create a more complex OpenStack system and, extend the proposed architecture by means of high availability mechanisms. We then draw conclusions about their effectiveness.

Figure 1 shows our virtual test environment architecture, which is the proposed architecture of the official OpenStack install guide[8]. All nodes are virtual machines running on a physical host. The tenant virtual machines are dispatched on the compute node by means of nested virtualization. By default, our environment contains the following nodes:

- One controller node: this node runs the OpenStack Dashboard (Horizon), the API services, the MySQL database, the RabbitMQ message queue server, the scheduler for the compute resources, Identity (Keystone) and Image (Glance) services.
- One network (Neutron) node: this node handles the internal and external routing and DHCP services for the virtual networks.
- Two compute nodes: the compute nodes are the computing resources for running the virtual machines of the OpenStack users. They run the hypervisor and services like `nova-compute`, which is responsible for creating and terminating virtual machine instances through the hypervisor APIs.
- Two object storage (Swift) nodes: these nodes operate the OpenStack container and object services and each contain two local block storage disks for persisting the objects.

Further, the following networks are used to communicate between nodes and instances:

[7] `http://www.ansible.com`

[8] `http://docs.openstack.org/kilo/install-guide/install/apt/content/index.html`

- Management network: this network is used for the OpenStack administration, i.e., it connects the OpenStack services on the different nodes.
- Tenant or tunnel network: these networks can be created by the OpenStack users to achieve communication between projects or instances.
- External network: this network provides internet access to the instances.

This architecture is comprehensive enough to test various OpenStack use cases and analyze the dependability of the system.

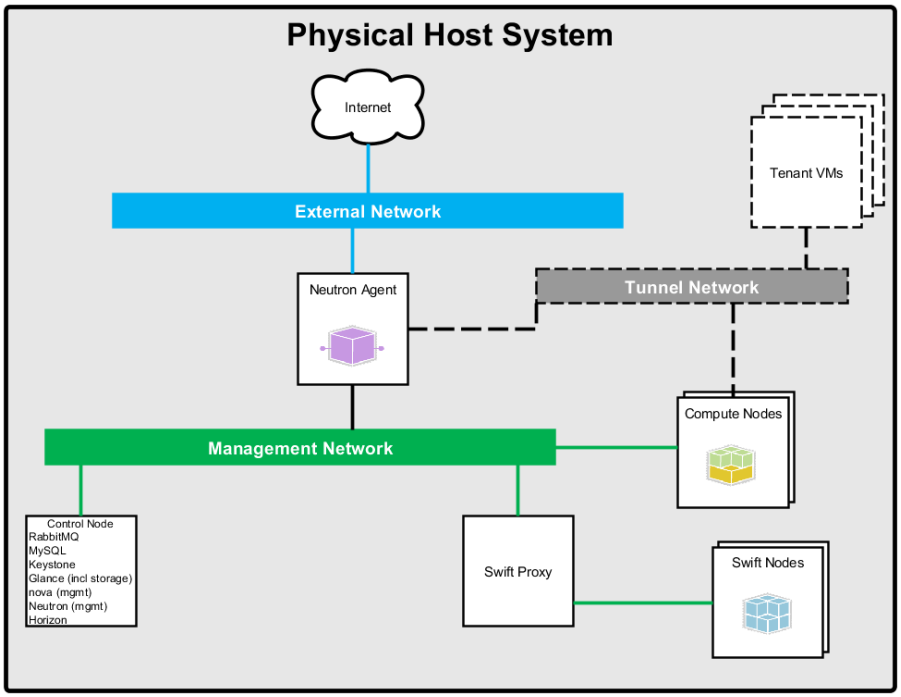

Figure 1: Our test environment architecture, based on the one proposed by the OpenStack install guide

The virtual OpenStack installation requires far less hardware resources than a distributed bare metal installation. It is possible to install a fully functional simplified test environment with one compute and object storage node on a quad-core Intel Xeon machine with 8GB RAM. For the full installation, more resources are recommended. We used a 16-core machine with 64GB RAM.

4 Automated Installation of OpenStack

In this chapter, we describe the automated installation process of OpenStack on our virtual environment. We give an introduction to the usage and a conceptual overview.

4.1 How to Install OpenStack using our System

The installation scripts are developed and tested on a Ubunutu 14.04 LTE (Trusty Tahr) desktop version. All required dependencies (e.g., Ansible, libvirt, etc.) are installed automatically, thus an internet connection is required. It installs OpenStack virtually creating the architecture described in Capter 3.2. The virtual machines are automatically created. The installation takes between 10-15 minutes

After the successful installation, a snapshot named "initial" is created. This allows for thorough dependability testing without re-installing the whole system after each experiment. Snapshots can be created manually as well. The snapshotting mechanism will shut down all virtual machines, snapshot the virtual hard drives of all nodes, and bring back up all machines.

4.2 Creating the Virtual Environment

The virtual environment, i.e., the virtual networks and the virtual machines, are defined in the `config` folder as libvirt network and libvirt domain XML-files. These XML-files define for example the IP addresses of the networks or the hardware specifications (size of RAM, number of cores, virtual hard drives, network interfaces) of the virtual machines. The virtual machines are then created with an Ubuntu cloud image[9], which are pre-configured images customized especially for running on cloud platforms.

The initialization of the virtual machines is done using cloud-init[10], which allows for setting passwords and SSH keys for easily connecting to the them afterwards, which is also a prerequisite for utilizing Ansible in the next steps. Further, using cloud-init, the correct `etc/network/interfaces` configuration files are copied to the virtual machines.

In addition to the virtual machines for the OpenStack nodes, a further virtual machine named "aptcache" is created. This machine is used as a package repository by the others. The packages it delivers are not updated, meaning that the versions of all installed packages are frozen. This is important for acquiring a reproducible test environment and prevents different experiment outcomes to be caused by different package versions throughout the system.

[9] `https://cloud-images.ubuntu.com/trusty/`
[10] `https://cloudinit.readthedocs.org/`

These virtual machines create the base of the virtual OpenStack test environment and are now ready for the actual OpenStack installation.

4.3 Installing OpenStack on Virtual Machines with Ansible

In order to be able to install OpenStack on the created virtual environment, Ansible must be configured in such a way that virtual machines are grouped. This allows for installing different parts of OpenStack on the different nodes. These groups are defined in a so called Ansible hosts file, which assigns the IP addresses of the different nodes to different groups. In our case, each node type is a group, i.e., "controller", "network", "compute" and "object". This allows for the so called Ansible playbooks to be executed on the specified groups of nodes in parallel.

The Ansible playbooks that run the installation of OpenStack on the virtual nodes are strongly based on the official OpenStack installation guide[11], which is very extensive. This gives the advantage of not having to write any further OpenStack related documentation additionally to the documentation of the technicalities of the creation of the virtual machines and the Ansible installation. The arrangement of the Ansible playbooks in a folder structure derived from the content of the OpenStack documentation allows for easily finding the corresponding part of the documentation should one require information about a certain part of the installation.

The Ansible installation of OpenStack starts with an initial preconfiguration of the virtual nodes. In this step, the hosts files are created in order to be able to connect to the nodes by their host names. These host names are then also added to the SSH `known_hosts` files to enable the SSH connection without warning messages. Further steps include setting the locale of the virtual machines to prevent locale errors as well as deactivating the `/etc/cloud/cloud.cfg` file, as all configuration of the images is done in the previous step, see Chapter 4.2.

A special virtual machine named "aptcache" is set up first and independently of the others. It runs Apt-Cacher-NG[12], a caching proxy for Linux package repositories. All other VMs are set up to request their packages through it. This a) decreases network traffic and wait times and b) can be used to repeat the setup process while using the exact same package versions as the first time: The first install seeds the package cache, a repeated installation then can receive all packages from the cache instead of fetching potentially newer versions from upstream. To allow this, the cached data is not stored in the VM, but in a folder shared from the host machine.

[11] `http://docs.openstack.org/kilo/install-guide/install/apt/content/index.html`

[12] `https://www.unix-ag.uni-kl.de/~bloch/acng/`

The first step of the actual OpenStack installation is installing the basic environment. This includes adding the OpenStack package repository to the nodes and installing the MySQL database on the controller and the message queue RabbitMQ on all nodes.

The next step is the installation of the OpenStack identity service (Keystone) on the controller node. This service is responsible for the permissions of users and keeping track of the available OpenStack services with their endpoints. The installation of this service includes creating a database, installing the Keystone client packages and populating the database. A demo and an admin tenant are initially created. As with all OpenStack services, the installation is finalized by creating the service entity and the API endpoint.

Next, the image service Glance is added on the controller node. This service allows for retrieving and registering virtual machine images. Again, a database for this service is created along with the installation of the Glance client packages and the creation of an API endpoint.

OpenStack compute is then setup on the controller and the compute nodes. This component is responsible for the administration and hosting of the computing systems. It allows - among other things - for creating and terminating virtual machine instances through hypervisor APIs and is responsible for scheduling on which compute node an instance should run. To install the compute service Nova on the controller, the respective database and API endpoint is created and the nova service is configured. On the compute nodes, the nova-compute packages are installed and configured to run a QEMU hypervisor with the KVM extension.

The OpenStack networking Neutron components are installed next on the network, controller and compute nodes. The main responsibility of the networking component is to provide connectivity between the instances running on the compute node. On the controller node, database and endpoint API are created, the networking server component is configured and the Modular Layer 2 plug-in is configured. This plug-in is responsible for the networking framework of the instances running on the compute nodes. On the network node, the networking components are installed and configured accordingly. The layer-3 agent for routing services, the DHCP agent, the metadata agent and the Open vSwitch service are configured. Lastly, the networking components, the Modular Layer 2 plug-in and the Open vSwitch service are configured on the compute node. The external and tenant networks are then created to finalize the installation.

The OpenStack web interface dashboard Horizon is then installed on the controller node. This allows administrators and users to access and manage their resources on the OpenStack cloud.

The last component that is added to OpenStack in regards to this masters project is the object storage Swift. This allows to create containers, upload and download files and management of the objects on the storage nodes. On the controller node,

the proxy service that handles the requests for the storage nodes is installed and configured. The storage nodes require two empty storage devices for persisting the objects. The virtual machines for these nodes contain two virtual devices in the qcow file format. The OpenStack Swift components are then installed and configured on the object storage nodes. Following the installation, the initial account, container and object rings are created.

5 Running Dependability Experiments

In this chapter, we will describe our OpenStack dependability experiments and their results. The implementations of these experiments all follow the same structure, so that it is easy to add further experiments if needed. In our experiments, we focused on covering the failure of various OpenStack components or nodes. We will give insights on how these failures affect the OpenStack environment and how the system deals with each fault. This serves partly to show weak points of the setup and partly to document details about its behaviour.

Experiments consist of multiple stages: The *setup* stage creates all elements necessary to run the experiment. The *break* stage then breaks things. An optional *heal* stage tries something simple to undo the damage (e.g. reboot a shutdown node). After each of this stages, a *check* step is executed, which observes the state of the system and reports its findings to the user. Generally, after the setup stage all checks should be successful. Where user observations are useful (e.g. by looking not just at API results, but seeing how Horizon represents situations), the user is prompted to do so.

Using the snapshotting mechanism, the user can always completely restore the system, even if it didn't survive an experiment. This is not done by default because a) it takes some time and is not always necessary and b) to allow the user to inspect the system state after the experiment has concluded, e.g. to find or work out steps to fix remaining issues.

5.1 Experiment Results

This section describes our four example experiments and discusses their results.

5.1.1 Experiment 1: Control Node crash

In this experiment, a crash of the control node in the system is simulated.

Technical Background The controller node stores global information of the OpenStack cluster and runs the sub-services depending on it, which then give services on other nodes the specific information they need to e.g. run a specific

instance or to build network connectivity. In our setup, it also provides the dashboard Horizon and runs the authentication service. A failure of this node obviously is going to have a large impact, but some things that already are set up on other nodes continue to operate.

Experiment The experiment script creates an instance and then uses ping to verify availability of both the compute node and the started instance. It also tries to access OpenStack APIs. To simulate the fault, it either issues a shutdown command, simulating the unavailability of the node to the controller or more severely, just turns off its infrastructure virtual machine to simulate a full crash.

Results and Conclusion While the control node is turned off, outside connectivity to the already created instance remains, since it runs on the compute node and its network connection to the outside (managed by Neutron, via the network node) is unaffected. On the other hand, all attempts to use OpenStack APIs fail. Most user-facing APIs are accessed via the controller and thus completely unavailable. Others report errors, since every action has to be authorized using the Keystone service, which only runs on the compute node in our setup.

After the controller node is up again, OpenStack takes a few minutes to reestablish all service connections and in most cases is fully operational again. It is possible that the compute instances fail to reconnect to RabbitMQ and their services have to be restarted manually[13].

This shows that already running instances on OpenStack are generally not impacted by temporary failure or maintenance of quite a few central OpenStack components, but during their unavailability no changes can be made and recovery might need manual intervention by an operator.

If the controller node was shut down hard, obviously many more failure scenarios related to the underlying operating system or services are possible, e. g. missing information in databases or damaged file systems.

5.1.2 Experiment 2: Memchached Service Loss of Data

This experiment shows the effects of Keystone losing authentication tokens due to the design of its storage mechanism.

Technical Background A common way to authenticate for operations against the OpenStack API is to use tokens. A more complex and privileged authentication (e.g. a password check) is done once to obtain a security token. These tokens have a limited lifetime and can be limited in scope, so a user can generate a token for a specific task and pass it to a service, which then can use it to access other services in the users name. Tokens also are internal to OpenStack, whereas other

[13] `http://docs.openstack.org/openstack-ops/content/maintenance.html#cloud_controller_storage`

authentication might require accessing an external authentication provider (e.g. an LDAP server).

The Keystone service stores these tokens in memcached[14], which is, as the name alludes to, an in-memory caching service. Designed as just a fast caching layer, it neither has persistence to disk nor does it guarantee to keep all data it stores. If it deems necessary it can evict any information at any time (i.e. because it is under memory pressure).[15]

Experiment The admin credentials are used to create a token. To verify the tokens validity, it is used to authenticate an operation against the Keystone API. To cause memcached to evict it from the cache, the command for memcached to delete all its data is issued.

Results and Conclusion Subsequent attempts to use the token fail, since it has been deleted. This shows the consequences of OpenStack using an unreliable data store to save a central element of its authentication system, instead of using memcached as designed only as a cache to improve lookup speeds.

If a user is logged into the Horizon dashbord while the experiment is running, it also sometimes allows to observe failures: The dashboard site is still accessible (because the session on the web server still exists), but no information from OpenStack is shown because attempts to retrieve it fail due to the invalid token. The user has to log out and back in again to create a new token.

5.1.3 Experiment 3: Compute Nodes Unavailable

This experiment documents the behaviour if connectivity to compute nodes is lost.

Technical Background Instances are distributed among the compute nodes by the Nova scheduler, which runs on the controller node. Inspecting the state of VMs, e. g. directly via the Nova API or via the Horizon Dashboard is also done via information stored by Nova on the controller node, where it collects information from all compute hosts. If the controller looses connectivity to compute nodes, it can't accurately report the state of individual instances, as seen in this experiment.

Experiment This experiment first creates an instance to observe throughout the different stages. Then the first level of fault is introduced: All compute nodes are removed from the management network. Then an attempt to start a second instance is made. After this, a more severe fault is created by shutting down all compute nodes. Then a "fix" is attempted by restarting the VMs.

Results and Conclusion After the first fault, Nova on the controller has no connectivity to the compute nodes and is therefore unable to actually start the second instance. The first instance is still reported as being active, which in this

[14] `http://memcached.org/`

[15] `https://code.google.com/p/memcached/wiki/NewUserInternals#How_the_LRU_Decides_What_to_Evict`

case is correct: it is still running and can be reached from the outside network. Since it already has lost all connectivity to the compute nodes, things look exactly the same after the compute nodes are actually shut down, but in this case the state information is wrong: the VMs obviously are not active, but shut down together with the host they were running on. After the reboot of the compute nodes, they reconnect to the controller and the state of all VMs is reported correctly again.

This shows that when Nova looses connectivity to a compute node it can't report accurate information, which is one of the reasons why Nova doesn't implement functionality to automatically restart such VMs. Short interruption of Nova services, e.g. due to software updates, do not necessarily disrupt instance operations. Such decisions are left to other systems that can collect more information (e.g. by running active tests against VMs) and can be configured for specific application needs, like OpenStack Heat.

We find it interesting that the developers choose to report such instances as active and did not introduce another state to represent this special situation.

6 Conclusion

In our masters project we created a platform for the automated installation of an virtual OpenStack environment as well as a framework for some first dependability experiments on this environment[16]. The advantage of our platform is the very fast installation (ten to fifteen minutes) of a complete OpenStack environment on very limited hardware compared to a full bare metal installation. This makes running dependability experiments comparatively easy. Features like snapshotting would not be available on a bare metal setup, but are very useful when repeating such experiments. Further, our platform is easily extendable, both for adding further OpenStack components as well as further experiments.

These features can be the foundation for future work. Due to the limited time and personal resources, we were not able to implement the installation of a full OpenStack high availability setup. Such a setup could make it possible to compare the dependability of the "normal" OpenStack setup to the high availability one, by running the experiments on both. Further, due to the fact that we use Ansible for the installation, the playbooks themselves can be easily used for installing OpenStack on bare metal. For this, the scripts for configuring the virtual environment could be extended in order to make it possible to configure a bare metal setup. This would allow running the experiments on a bare metal OpenStack installation.

[16] `https://github.com/MasterprojectOpenStack2015/sourcecode`

References

[1] M. Bastian, S. Brueckner, K. Fabian, M. Hopstock, D. Korsch, and D. Stelter-Gliese. *Cloud Computing with OpenStack*. Report Masters Project. 2015.

[2] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva. "On Fault Resilience of OpenStack". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Available at http://doi.acm.org/10.1145/2523616.2523622. Santa Clara, California: ACM, 2013, 2:1–2:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523622.

[3] M. Kollárová. "Fault injection testing of OpenStack". Available at http://is.muni.cz/th/325503/fi_m/. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno, 2014.

[4] Q. Teng. *Enhancing High Availability in Context of OpenStack*. Available at https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/enhancing-high-availability-in-context-of-openstack. 2014.