University of Victoria
Engineering & Computer Science Co-op
Work Term Report
Fall 2017

Containerized Cloud Scheduling Environment

Department of Physics
University of Victoria
Victoria, BC

Tahya Weiss-Gibbons
V00832231
Work Term 2
Computer Science
tahyaw@uvic.ca

December 21, 2017

In partial fulfillment of the academic requirements of this co-op term

Supervisor's Approval: To be completed by the Co-op Employer

This report will be handled by UVic Co-op staff and will be read by one assigned report marker who may be a co-op staff member within the Engineering Computer Science/Math Co-operative Education Program, or a UVic faculty member or teaching assistant. The report will be either returned to the student or, subject to the student's right to appeal a grade, held for one year after which it will be destroyed.

I approve the release of this report to the University of Victoria for evaluation purposes only.

Signature: _____ Position: _____ Date: _____

Name (print): _____ Email: _____

Company Name: _____

Abstract

Cloudscheduler is a cloud management system developed by HEPRC, which schedules batch jobs and manages virtual machines. A simple testing and demonstration environment for cloudscheduler was desired, as an easy way for future users to test the software. Using Docker, a Linux container system, a containerized scheduling environment was developed. Very minimal installation and configuration is required to use the containerized environment, a user needs only to install Docker, pull the image from a central location and then start the container to have a full cloudscheduler deployment. Some testing was done with Devstack, an developers Openstack deployment, as a fully containerized method for using cloudscheduler without access to an external cloud. While it did give a fully contained cloud environment, Devstack was resource intensive to the run and difficult to ship within a container as it was not stable through container restarts. For this reason other solutions were investigated, including simulating a cloud environment using libvirt, a virtualization API, within the cloudscheduler container.

Contents

List of Figures

Containerized Cloud Testing Environment

Tahya Weiss-Gibbons, tahyaw@uvic.ca

December 21, 2017

1 Report Specification

1.1 Audience

This report is intended for anyone with an interest in batch computing on clouds. For users, this document will serve as an introduction to cloud computing concepts. For developers, this report will give insight into the inner workings and design choices of the cloudscheduler container.

1.2 Prerequisites

Some basic knowledge of Linux, container software, virtual machines, batch processing, Openstack software, python and yum is required.

1.3 Purpose

The purpose of this report is document the process of developing a cloudscheduler container, including the investigation options for using cloudscheduler without access to an external cloud.

2 Introduction

2.1 High Energy Physics Research Computing Group

The High Energy Physics Research Computing group (HEPRC) at the University of Victoria is actively engaged in a variety of projects for the analysis of data from particle physics experiments as well as providing advice to researchers in other fields [1]. Activities include high speed networking, virtualization and cloud computing. As part of their work on cloud computing, a program called cloudscheduler has been developed, which helps launch and manage virtual machines (VMs) for jobs on different computational clouds. It uses HTCondor as a batch system. HTCondor, "is a specialized workload management system for compute-intensive jobs" [2].

2.2 Creating a Testing Environment

Since cloudscheduler is a program written by the HEPRC group, it would be useful to them to have a ready to use cloud scheduler deployment to show other interested parties. This could be used for either demonstration purposes or their own testing with or without a existing cloud environment. For this containers could be quite useful, as a cloud scheduler container could be created, so a user would only need to install the container system and then pull the cloud scheduler image to get started with cloud scheduler. Work was done on configuring cloud scheduler to connect to an external cloud from within a container, as well as to create a fully contained testing environment with cloud scheduler and Devstack, an Openstack deployment and using libvirt to simulate a cloud within the container.

3 Cloud Scheduler in a Container

3.1 Docker Containers

Linux containers is a capability of the Linux operating system which provides an application with a self contained environment, including processes, file systems and memory. Docker is a container management software which aids in the development and sharing of containers. See Figure 1. Docker has mainly been developed for Linux systems. While Docker is also available on Mac and Windows, these options were not tested or used throughout this project.

3.2 Initial Configuration

To containerize cloudscheduler, initially a CentOS 6 base docker image was used, as this was the operating system which cloudscheduler had been most extensively tested on. To run cloudscheduler, it needed a few specific requirements, most importantly a running instance of HTCondor. This is where it will get it's job information from, and from there will assign or launch VMs on a specified cloud as needed. First step then was to install HTCondor inside of a container. Since containers do not by default have system administrative capabilities to the host, in the condor configuration file DISCARD_SESSION_KEYRING_ON_STARTUP had to be set to false. For a sample of the HTCondor configuration file see Figure 2.

Once HTCondor was running, cloudscheduler could be installed. It was discovered that the current master branch of cloudscheduler failed due to versioning errors with some required packages. This was fixed by switching to the development branch and manually installing cloudscheduler instead of through pip, the Python package management software. There was also an issue with python versions. CentOS 6 comes with Python 2.6 installed on it, this was required by the package manager yum. Cloudscheduler needs at least python 2.7 or higher to run. This was a problem as yum could not use the updated Python version. To work around this issue, the two versions of Python were used, with pointers so the correct programs used the correct versions. The default command line version was set to Python 2.7, as not to cause any issues with cloud scheduler, but in the yum file, the Python bash

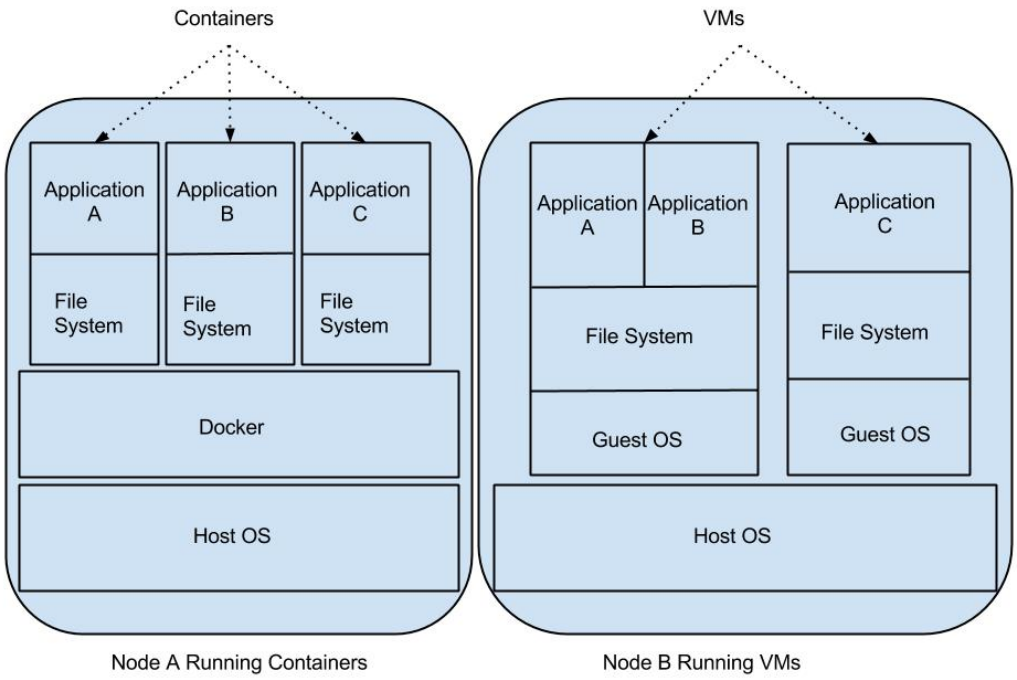

Figure 1: Linux container systems are based on the idea of creating multiple isolated Linux environments using the same host kernel [3]. While VMs have their own full guest OS and virtualized hardware, containers rely on the hosts OS and hardware. This makes container software more light weight and minimizes resource usage, when compared to VMs. [4].

```
CONDOR_HOST = $(FULL_HOSTNAME)
COLLECTOR_NAME = condor collector
START = TRUE
SUSPEND = FALSE
PREEMPT = FALSE
KILL = FALSE
DAEMON_LIST = COLLECTOR, MASTER, NEGOTIATOR, SCHEDD
TRUST_UID_DOMAIN = TRUE
ALLOW_WRITE = *
ALLOW_NEGOTIATOR = *
ALLOW_NEGOTIATOR_SCHEDD = *

COLLECTOR_SOCKET_CACHE_SIZE=10000
COLLECTOR.MAX_FILE_DESCRIPTORS = 10000
LOWPORT = 40000
HIGHPORT = 40500
ENABLE_SOAP = FALSE

DISCARD_SESSION_KEYRING_ON_STARTUP = FALSE
```

Figure 2: Example HTCondor configuration for running inside a container

line was set as Python 2.6

3.3 Upgrading the OS

While this gave a running implementation of cloudscheduler in a container it was preferable to use a newer OS for a few reasons. First the python versioning problem. While a work around was found having only one version of python was preferred. Second was the fact that CentOS 6 was not being fully supported anymore and it was desirable to have the newer version of CentOS. While most of the migration was trivial, this did lead to one major problem, which was getting systemd to run inside of the CentOS 7 container. For more information on this problem and solution see section 5.

4 Connecting to an External Cloud

4.1 Two-Way External Communication with Containers

Docker, on install sets up a private bridge network which is used as the default for any containers. This gives the host machine two way connectivity with the containers, the containers can connect to external networks but external networks cannot connect back to the

containers. This is because Docker will create IP table rules so that any outward connections from inside a container on the Docker bridge network will masquerade as the host IP. So if something were to try and connect back over the same connection it would only reach the docker host machine, not the container. Two way external connections with containers in docker are done through port mapping. When a container is run certain available ports on the host machine can be mapped to the docker container, so that anything trying to connect on that port through the host will be forwarded to the docker container. This can also be done by using an available IP address on the host network, and mapping the ports on the docker container to that address. So while the docker container still exists on the private docker network, anything trying to communicate through the allocated IP address on the mapped ports will be forwarded to the container.

4.2 HTCondor Ports

All VMs that are launched with cloudscheduler must register with the HTCondor host in order to run jobs. For this to happen, HTCondor needs certain ports available to connect with workers. The standard port is 9618, which is for the HTCondor collector daemon. As well as this port, an ephemeral range is needed for the other HTCondor daemons. On average, the number of ports needed in this range is proportional to the number of machines which report to that HTCondor host. The central manager of the HTCondor pool needs, by default, at least 5+16 ports in this range for internal condor daemon communication, where 16 is the default number of ports given for NEGOTIATOR_SOCKET_CACHE_SIZE [5]. A port range of $\tilde{5}00$ ports was advertised for testing purposes. While this was a sufficient number for a small pool, the port range would need to be increased if more jobs were being run using this scheduler. A typical cloud scheduler instance has an ephemeral port range for condor of $\tilde{5}000$. One reason the port range was kept small was because of the increase in start up time when getting docker to bind to a large number of ports. Assigning a port range of 5000 ports to a container can increase the start up time to 20-30 minutes. A port range of 500 ports gave a startup time for the container of 30 seconds to a minute. The port range assigned to a container cannot be dynamically changed after the container is initially launched. For a small testing instance of cloud scheduler, 500 ports was sufficient.

4.3 Networking

Even with the proper ports published for the container, cloud scheduler could not to connect to Beaver, the Openstack cloud owned and operated by the HEPRC group. For this, a better understanding of the local networking was required. For a visualization, see Figure 3. The disconnect was due to the Access Control List (ACL) rules that were set up in between the physics network and the HEP network. While the host machine for the container was able to connect with Beaver, the container was not able to connect as the connection was blocked by ACL rules. The easiest way to work around this problem was to move the container outside of the physics network, so that those ACL rules were no longer applicable. A VM was setup to act as the Docker host on the HEP network and the container was migrated

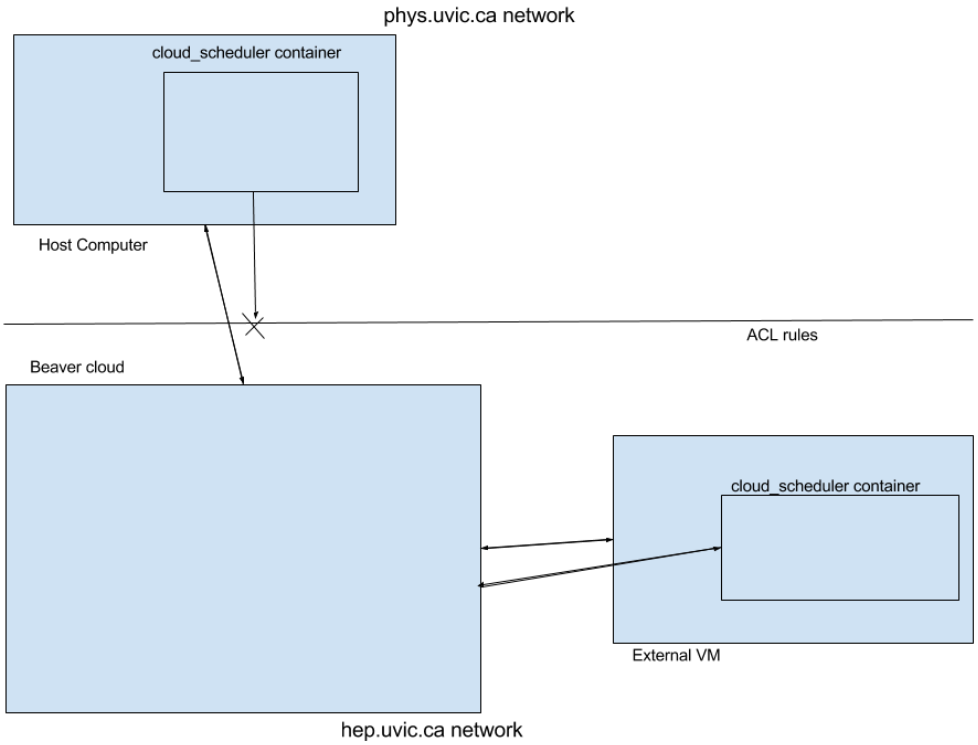

Figure 3: Local Network Configuration. Initially the container was on a host inside of the Physics network, while the Beaver cloud was on the HEP network. Due to the ACL rules between the two networks blocking the connections from the container to the cloud, the container was then moved to the HEP network, using an external VM as the new host

there. Without the ACL rules blocking, Beaver and the cloud scheduler container were now able to communicate with each other. Cloud scheduler could then launch VM's on Beaver when needed for jobs submitted to condor.

4.4 CCB Connection

The Condor Connection Broker(CCB) creates two way connections between condor hosts and registered worker machines [5]. When a VM is launched by cloud scheduler, it starts and configures condor inside of the VM to be a worker for the cloud scheduler host. It will register with condor using the host address given to it by cloud scheduler, after which it will wait for a job from the condor host. It does not know the internal container IP address, only the address given to it by cloud scheduler, which is the address that the condor ports were mapped to by Docker. Internally though, the container does not know of a external IP address, it only knows its private, host only address on the docker bridge network. So once a VM laucnhed by cloudscheduler has registered with the HTCondor

central manager on the container, HTCondor on the host will send the VM a job to run using the private container network IP. The VM cannot connect back the container over this IP, so it will fail to broker a CCB connection and the job will fail. To get around this issue, the TCP_FORWARDING_HOST must be set in the condor configuration file on the host as the external IP given to the container. This way the container knows for all requests to workers, it must tell the worker to respond to the request on the address specified. Then the worker will see that the request came from the host it has already registered with and be able to respond to and run the job. With this, cloudscheduler could launch a VM on Beaver, the VM would register with the HTCondor host and then jobs could run.

5 Containers and Systemd

5.1 Background

Systemd is used by Linux distributions to manage system processes. Systemd is not used in CentOS 6, but it is used in the newer CentOS 7. For Docker, the standard is one process per container, which leaves little need for a system manager other than the docker daemon. For this reason, systemd is not automatically included in any docker OS container image. The developers of systemd have argued that a system management process should always run in a container, especially to deal with zombie processes which will not be killed without one. For the purposes of this report, this issue will only be dealt with as it relates to needing to run multiple processes in one container. Recently changes have been made to both systemd and Docker to make it possible to enable a system management process within a container.

5.2 CentOS 7 and Systemd

The cloudscheduler container breaks the one process per container guideline of docker, since for cloudscheduler to run it also needs HTCondor running in the same container. To get both running in a CentOS 7 container, a system management process is needed. Some guidance for a systemd enabled CentOS 7 container is given in the documentation for the base CentOS 7 image [6]. See Figure 4 for the Dockerfile used to build the docker image. This could then be used as the base image for the cloud scheduler container. There were a few extra configurations needed to properly enable systemd within the container, and that was to do with administrative capabilities of the container. To get around this issue, the cloudscheduler container was run with the privileged flag, which gave it all needed permissions. Some success was had in testing with a Ubuntu image with systemd enabled that did not need privileged permissions [7], though this was not implemented for CentOS 7. Future work could include creating a systemd enabled CentOS 7 container which does not need privileged permissions, but that is beyond the scope of this report.

```
FROM centos:7
ENV container docker
RUN (cd /lib/systemd/system/sysinit.target.wants/; for i in *; do [ $i == \
systemd-tmpfiles-setup.service ] || rm -f $i; done); \
rm -f /lib/systemd/system/multi-user.target.wants/*;\
rm -f /etc/systemd/system/*.wants/*;\
rm -f /lib/systemd/system/local-fs.target.wants/*; \
rm -f /lib/systemd/system/sockets.target.wants/*udev*; \
rm -f /lib/systemd/system/sockets.target.wants/*initctl*; \
rm -f /lib/systemd/system/basic.target.wants/*;\
rm -f /lib/systemd/system/anaconda.target.wants/*;
VOLUME [ "/sys/fs/cgroup" ]
CMD ["/usr/sbin/init"]
```

Figure 4: Systemd Enabled CentOS 7 Dockerfile [6]

6 Containerized Cloud

6.1 Devstack

Devstack is a series of scripts which creates a running Openstack instance. It is meant as a development tool, and is able to run on a single host. This made it a good candidate for creating a full testing environment with cloud scheduler. Devstack does have some requirements, including fairly high memory requirements, as it does institute a full Openstack instance. At least 4GB of free RAM is required to even start the Devstack cloud, though to launch VM's at least 8GB is recommended. It is also strongly recommended that Devstack be run on a clean OS, as it makes substantial changes to the system. For this a container was well suited as it isolated the instance. Devstack has a minimum OS requirement, so CentOS 7 was used for the container. This was mainly due to initial testing that had both Devstack and cloud scheduler in a single container. They were later separated into two containers, but the OS was kept the same. Devstack also required a service management system to start the different projects used by Openstack, so the systemd enabled container previously discussed was used, see Section 5 for more detail. Devstack could easily be migrated to the Solita/Ubuntu image [7] if desired, so it would no longer need to be run with the privileged flag.

6.2 Fully Contained Testing Environment

Using the systemd CentOS 7 container, run with the privileged flag, Devstack could then be setup inside of a container. The configuration file used for the setup can be seen in Figure 5. IPv6 addressing had to be disabled in both docker and Openstack to prevent addressing confusion between the two containers. Since both the cloud instance and cloud

```
[[local|localrc]]
ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
IP_VERSION=4

[[post-config|/etc/neutron/dhcp_agent.ini]]
[DEFAULT]
force_metadata=True
```

Figure 5: Configuration File for Devstack inside of a Container

scheduler were both in the docker bridge network, no port forwarding was needed. The TCP_FORWARDING_HOST still needed to be set in the condor configuration file to the docker internal IP of the host. With this, a user could fully test a cloud scheduler instance without having access to an external cloud, they merely had to use the two Docker images to create a running container instance of cloud scheduler and Openstack.

6.3 Limitations

There were some limitations with this method, one main one being the difficulty in restarting a Devstack container and creating a Devstack image. For every stack instance, Openstack needs to be stacked. When a container is stopped and saved to an image, launching a new container is similar to rebooting the system, which Devstack as a development instance was never created to support. Stacking a Devstack instance takes anywhere from 15 to 40 minutes, mainly dependant on the network connection speed. Also trying to restart an instance lead to many different errors and failures, due to a wide variety of things such as loop back devices not being properly cleaned on exit, socket files no longer existing after reboot and volume files not being loaded correctly. A small workaround for this is initially stacking the instance, unstacking it and then saving the container as an image. It could then be restarted and stacked again in offline mode, where no new packages are downloaded and the previously existing ones are used. This can cut down on time as it is no longer reliant on the network. In testing this cut the time needed to stack the instance by approximately half. It was essential that the instance be unstacked before saving the image, as not unstacking it would cause the new instance to fail on stacking. It was also very important that before deleting a Devstack container, both the unstacking script and the clean script are run, as not doing so could result in errors for a future Devstack container.

7 Alternate Testing Clouds

7.1 Cloud Type LocalHost

Due to the limitations of Devstack previously discussed, other options were explored for a test cloud environment. An alternate solution would be to create a new cloud type within the cloudscheduler code which could launch new VMs directly within the container using libvirt. Libvirt is an opensource project to manage virtualization hosts [8]. If libvirt was included inside of the container, and the host was enabled for virtualization, a local cloud type could be created as to launch a VM directly within the container. The advantages of this over Devstack are that it eliminates the need for a separate container, as it is all run internally in the cloud scheduler container. It is also far less resource intensive than Devstack and does not have the start up time and stability issues of Devstack. To be able to use libvirt inside of a container, the container needed to be run in privileged mode to give it the necessary permissions to launch VMs.

8 Conclusion

The advantage of the work done inside of a container is the portability and short start up time for anyone looking to test this technology. Docker can run on any Linux host and can also be run for Mac and Windows, though neither of those deployments were tested. Cloudscheduler worked well inside of a container, as it runs as an isolated system with only HTCondor and its other dependencies. It can be stopped and restarted easily, only needing to restart the HTCondor and cloudscheduler services upon a container restart. With using CentOS 7, while it solved other Python versioning issues, it did require systemd to be enabled and the container to be run in privileged mode which could potentially serve as a security threat. Future work could include getting a systemd enabled container for CentOS 7 working with the privileged flag, similar to Solitas Ubuntu container [7]. With the proper port forwarding and configuration set, cloudscheduler in a container was also well suited as a host for external VM's. Devstack was less well suited for a container. While it did offer the isolation needed by Devstack, it was resource intensive, which could make it difficult to run on an average laptop. It was also time consuming to stack Devstack upon starting the container every time and did not ship well as a container image. For these reasons, adding a new cloud type into cloudscheduler was a better alternate solution. It needed less setup by a user and used the preexisting cloudscheduler mechanisms for adding cloud types and monitoring VMs without having to configure a new cloud.

9 Acknowledgments

I would like to thank Randy Sobie for the opportunity to do this coop. I would also like to thank Colin Levitt-Brown, Kevin Casteels and Micheal Paterson for all of the help and guidance.

10 Glossary

ACL Access Control List, a list of permissions controlling connections in a network

Beaver An Openstack cloud maintained and used by HEPRC

CCB Condor Connection Brokering, creates a connection in a condor pool

Condor HTCondor, also simply known as condor, is a batch management system

Dockerfile A file for automatically creating a docker image, a set of instructions for docker

HEP High Energy Physics group, or the High Energy Physics Research Computing group

IP Internet Protocol Address. For this project IPv4 was used

Openstack An open source project for cloud computing

OS Operating System

TCP Transmission Control Protocol

VM Virtual Machine

References

[1] "High Energy Physics Research Computing", *High Energy Physics Research Computing* [Online] Available: http://heprc.phys.uvic.ca/

[2] "What is Condor?", *HTCondor: High Throughout Computing* [Online] Available: https://research.cs.wisc.edu/htcondor/description.html

[3] "Linux Containers", *archLinux* [Online] Available: https://wiki.archlinux.org/index.php/Linux_Containers

[4] "What is a Container?", *Docker* [Online] Available: https://www.docker.com/what-container

[5] "HTCondor: 3.7 Networking (includes sections on Port Usage and CCB)", *HTCondor Documentation* [Online] Available: http://research.cs.wisc.edu/htcondor/manual/v7.6/3_7Networking_includes.html

[6] "CentOS: Image Documentation", *Docker Hub* [Online] Available: https://hub.docker.com/_/centos/

[7] "solita-ubuntu/systemd", *Docker Hub* [Online] Available: https://hub.docker.com/r/solita/ubuntu-systemd/

[8] "libvirt: The virtualization API", *libvirt* [Online] Available: https://libvirt.org/