Simplified Virtualization in a HEP/NP Environment with Condor

W Strecker-Kellogg, C Caramarcu, C Hollowell, and T Wong
RHIC/ATLAS Computing Facility, Brookhaven National Lab, New York
E-mail: willsk@bnl.gov

Abstract. In this work we will address the development of a simple prototype virtualized worker node cluster, using Scientific Linux 6.x as a base OS, KVM and the libvirt API for virtualization, and the Condor batch software to manage virtual machines. The discussion in this paper provides details on our experience with building, configuring, and deploying the various components from bare metal, including the base OS, creation and distribution of the virtualized OS images and the integration of batch services with the virtual machines. Our focus was on simplicity and interoperability with our existing architecture.

1. Introduction
Faced with the problem of maintaining legacy operating systems here at the RHIC/ATLAS Computing Facility (RACF), we examine what it would take to implement a cloud-computing solution with a minimum of extra expense and software.

Cloud computing, otherwise known as Infrastructure as a Service (IaaS), or on-demand virtual-machine instantiation, provides some exciting opportunities for systems administrators and physicists alike in the HEP community, and we hope to leverage this technology in implementing our solution. Our solution involves integration with Condor (our batch system), ensuring security, access to data, and adequate performance all while keeping the additional software and requirements minimal [1].

The final result is that users have the ability to request that they run code inside of a virtual machine with a few simple additions or modifications to their condor job-description files. The batch system and wrapper scripts abstract all of the details away and their code runs transparently inside of a VM for each job.

2. Image Content and Distribution
The first problem was to consider what images to run and how to distribute them to our farm. Trusted (“golden”) images are provided, which are copies of our own image at the RACF from various past OS versions. Their distribution is discussed below in §2.1. In this schema there is also the possibility of running untrusted, user-provided images in a secure manner, but these images are distributed the traditional way—either using the batch system’s file-transfer mechanism or putting them in an NFS directory. The fundamental divide between user images and trusted images will be discussed further in §3. For now we focus on our image-distribution as implemented for trusted images.
2.1. HTTP Based Distribution

Our golden images are held on a webserver, transferred over HTTP, and cached locally using a simple cache-refresh cycle at the beginning of each job. This was designed to be similar to and easily implementable using CVMFS [2].

Images are kept on the webserver in a compressed format, along with a file containing MD5 checksums of the uncompressed images, and are distributed by running an image-refresh cycle at the start of each job. On a farm node, the checksum file is downloaded and compared with a local copy to determine if the requested image is present and up to date. If not, the image is downloaded, decompressed, and re-verified against the new checksum file. This way only images that are requested are kept on the local disk and these images get updates as necessary. A node-wide lock file is used to serialize instances of this image-refresh script because it re-downloads all out-of-date images at once, and running multiple instances creates a race condition.

2.2. Storage Concerns

The golden images we provide are ~6Gb in size when uncompressed and on our farm we have room to store 5 of these images in our scratch space without impacting any operations. An image expiration process needs to be implemented if the number of golden images grows, which is easy to do. Each time an image is run a record is updated in a file in the cache directory. When space runs low, the oldest unused images are removed in order until enough free space is attained.

3. Security

As mentioned in §2 there are two main classes of images, trusted “golden” images and user-supplied ones. For security reasons, user supplied images cannot be allowed access to our network directly and must not be able to access shared resources. Many services on our network use UID-based authentication, which relies on the fact that root-level privileges are required to
bind to low ports (< 1024), the most important of which is NFS. For untrusted images, access to these ports is blocked as discussed in §4.3, which presents new problems in allowing users access to their data (see §5 for a discussion of this). For trusted images we handle this as discussed in §3.1.

3.1. Image Trust Levels and Privileges
In order that users cannot get root on our network it is important to ensure that the trusted images run code only as the user who runs the job. To do this we use libguestfs [3] to inject a custom rc.local that becomes the user before executing the job’s code, ensuring the user doesn’t have root on our network.

User supplied directories must be mounted with the “nosuid” mount flag or the “uid” flag set to their own uid to ensure that they cannot provide themselves with, for example, a shell executable with the setuid bit enabled!

3.2. Wrapper Security
To prevent users from instantiating their own VM’s with our setuid wrapper, we include code in the wrapper that looks at the path to the executable of the parent process to ensure it is a batch system process (/proc/<ppid>/exe → /usr/bin/condor_starter). Since the path to this executable is only changeable by root, this prevents unprivileged users from running this wrapper on their own (not underneath the batch system).

4. Machine Instantiation
Machines are managed underneath our batch system using a custom setuid-binary that runs the image-refresh code (cf. §2) followed by a script that configures and boots the virtual machine.

Trusted images are QCOW2 [4] formatted so as to enable the golden image directory on disk to be a read-only backing store for a temporary image that writes its diffs into the scratch-directory for the job. This both automates the cleanup of user modifications and enables the separation of multiple instances of the same VM.

4.1. Contextualization
Trusted VM’s are stored on disk and a fresh copy-on-write image is created per-job, which we then use libguestfs to contextualize. The username to run as, the path to run, and the environment mapping are injected into the VM as three files. Then a custom rc.local reads these, becomes the user (possible because our custom images share our site LDAP), then executes the script. Once the user code exits, this rc.local wrapper shuts off the virtual machine.

4.2. Condor Tie-in
In order to instantiate a VM a user is required to add at least the following flags to their Condor job-description file (JDF):

```
Command = /usr/local/racf/vm
Args = -i sl_53
+VM_Command = /experiment/u/username/script.sh
+VM_Args = data1 2 3 output
```

These tell the wrapper program to instruct the VM to boot our Scientific Linux 5.3 image and run the command provided in VM_Command with arguments listed in VM_Args. There are additional options available whose functionality will be discussed in further sections but these are the minimum required to boot and run a trusted image.
4.3. Networking
Networking is done via libvirt [5], where we set up a NAT for each VM. For security reasons discussed in §3 we cannot allow untrusted images to access low ports on our network, so if the image is not trusted libvirt sets up the following port-remapping via iptables: ports $\leq 1024 \rightarrow$ ports $> 1024$.

The network devices last only as long as the machine is running and thus are defined as transient devices inside libvirt which ensures the devices and the corresponding iptables forwarding rules are cleaned up after each job exits.

4.4. Machine Specifications
In a Condor pool configured for such behavior, jobs can request how many CPU’s and how much RAM they require, and the batch system partitions the node accordingly. The wrapper script inspects the condor job description and boots the virtual machine with the requested resources. Leveraging the batch system for this functionality is advantageous as it precludes users from over-provisioning the node without requiring complex logic in the wrapper script to handle this. Additionally the user’s usage-accounting as done by the batch system scales appropriately for fair-share purposes— a two-core VM counts the same as two single-core VM’s against the same user’s usage.

4.5. Life-cycle Management
Since each VM is managed by a Condor job, the batch system does the job of life-cycle management to first order. However, in order to handle job eviction and runaway user code in a manner more elegant than sending SIGKILL to a running qemu process, the wrapper scripts need to do a bit of signal handling. When Condor vacates a slot it sends SIGTERM and starts a short countdown before a kill signal. The wrapper can handle this and issue a shutdown command to the running VM that will pass along a SIGTERM to the user code running inside the VM to give it a chance to flush I/O buffers and clean up after itself.

Inside trusted images, the custom rc.local wrapper that executes the user code will issue a shutdown command upon user-code completion to ensure the VM is destroyed as soon as it can be. If a custom VM is provided the user is responsible for providing a fully contextualized image and for providing an automatic shutdown mechanism. If any user-provided VM’s are found to be running a long time without producing any load, we assume they are stuck and it is easy to write a Condor policy that looks at the reported LoadAverage from the job and kills VM jobs that leave it too low for too long.

5. User Data Access
User data is most easily accessed inside a trusted image through a network-accessible resource like NFS, dCache, or XRootD. This has the added benefit of being transparent to the user because this is already how they access their data inside non-virtual jobs. Output is mostly written to NFS, but sometimes to a local disk, which poses a problem. Users can request a blank image be created and mounted at “/scratch” inside the VM. This file is then optionally transferred back to the user upon job-completion. The QCOW2 [6] format can be slow with disk-writes compared to a raw image, so users who transfer large data-files to the local disk but don’t want the output can do so with greater performance by using this scratch area.

Like output, input data usually comes from network-accessible resources, but can sometimes come directly from the user. In order to handle this we allow users to supply a disk image that is mounted read-only inside the VM as “/data”. This image is transferred to the execute node as a standard input file in Condor.
Table 1. Summary of options available to each image type

| Feature                | Trusted Images | User-Supplied Images |
|------------------------|----------------|----------------------|
| Low Port Remapping     | No             | Yes                  |
| User-Supplied Data     | Yes            | Yes                  |
| NFS/AFS Access         | Yes            | No                   |
| Inject User Code       | Yes            | No                   |
| User-Exportable Scratch| Yes            | Yes                  |

6. Conclusions
We presented a method of integrating a cloud-computing-like design alongside a currently running batch system with a minimum of changes to the current infrastructure. With some software and a repository of trusted images, it is not very hard to implement a scalable, secure, and minimally intrusive scheme that allows users to run inside legacy OS’s.

Presented in Table 1 are the limitations and features of this system and how they apply to trusted and user-supplied virtual machine images, with. The purpose of this scheme was not to provide an end-product with lots of functionality, but rather to prove the concept that it is relatively easy to do “Cloud Computing” as a simple layer above an already-existing batch system without the use of large middleware projects. Additional features and complexity can be added to the software while still keeping it far simpler than a full cloud-deployment stack.

References
[1] Buncic P, Aguado Sanchez C, Blomer J, Harutyunyan A and Mudrinic M 2011 The European Physical Journal Plus 126(1) 1-8 ISSN 2190-5444 10.1140/epjp/i2011-11013-1 URL http://dx.doi.org/10.1140/epjp/i2011-11013-1
[2] Blomer J, Buncic P and Fuhrmann T 2011 CernVM-FS: delivering scientific software to globally distributed computing resources NDM ’11 (New York, NY, USA: ACM) pp 49–56 ISBN 978-1-4503-1132-8 URL http://doi.acm.org/10.1145/2110217.2110225
[3] Libguestfs, library for accessing vm disk images URL http://libguestfs.org/
[4] Qcow2-kvm URL http://www.linux-kvm.org/page/Qcow2
[5] Libvirt: The virtualization api URL http://libvirt.org/
[6] Le D, Huang H and Wang H 2012 Understanding Performance Implications of Nested File Systems in a Virtualized Environment USENIX FAST’12 URL https://www.usenix.org/system/files/conference/fast12/le.pdf