Cloud services for the Fermilab scientific stakeholders

S Timm[1], G Garzoglio[1], P Mhashilkar[1*], J Boyd[1] , G Bernabeu[1], N Sharma[1], N Peregonow[1], H Kim[1], S Noh[2], S Palur[3], and I Raicu[3]

[1]Scientific Computing Division, Fermi National Accelerator Laboratory

[2]Global Science experimental Data hub Center, Korea Institute of Science and Technology Information

[3]Department of Computer Science, Illinois Institute of Technology

E-mail: {timm, garzoglio, parag, boyd, gerard1, neha, njp, hyunwoo }@fnal.gov, rsyoung@kisti.re.kr, psandeep@hawk.iit.edu, iraicu@cs.iit.edu

Abstract. As part of the Fermilab/KISTI cooperative research project, Fermilab has successfully run an experimental simulation workflow at scale on a federation of Amazon Web Services (AWS), FermiCloud, and local FermiGrid resources. We used the CernVM-FS (CVMFS) file system to deliver the application software. We established Squid caching servers in AWS as well, using the Shoal system to let each individual virtual machine find the closest squid server. We also developed an automatic virtual machine conversion system so that we could transition virtual machines made on FermiCloud to Amazon Web Services. We used this system to successfully run a cosmic ray simulation of the NOvA detector at Fermilab, making use of both AWS spot pricing and network bandwidth discounts to minimize the cost. On FermiCloud we also were able to run the workflow at the scale of 1000 virtual machines, using a private network routable inside of Fermilab. We present in detail the technological improvements that were used to make this work a reality.

1. Introduction

The Fermilab scientific program includes several running experiments, both the CMS experiment at the Energy Frontier, and the various neutrino and muon experiments on the Intensity Frontier. The ongoing data analysis and simulation for running experiments, combined with a large simulation load for future facilities and experiments, results in an unprecedented level of computing demand. This paper describes recent progress in the ongoing program of work to expand our computing to the distributed resources of grids and public clouds.

As part of the joint collaboration between Fermilab and KISTI, we have a program of work building towards distributed federated clouds. In the summer of 2014 the primary goal of this program was to demonstrate a federated cloud running at the 1000 Virtual Machine scale, using our local private cloud nodes and Amazon Web Services EC2. Our application of choice for this is the Cosmic Ray simulation of the NOvA experiment far detector at Ash River [9]. This application

[*] To whom any correspondence should be addressed.

requires negligible input and produces about 250MB of output per job, and is quite computationally intensive. The NOvA experimenters spent considerable effort in optimizing their code to run at various sites outside of Fermilab, including loading their code into the OSG OASIS CVMFS server. The NOvA experiment supplied us with a set of files and scripts, which would generate one full set of their cosmic ray Monte Carlo, 20000 input files in all, with one job per file.

2. Challenges in using cloud resources at large scale

The main challenge that had to be addressed in expanding to larger scale on the public cloud was finding a scalable way to distribute the code to 1000 simultaneous jobs. We also needed the capacity to quickly convert a local virtual image to Amazon Web Services format so we could make changes in the setup as needed. On the private cloud we had to find a faster and more scalable way to deliver virtual machine images, and find a more scalable virtual machine instantiation API, as well as add a significant number of machines to our private cloud. We were able to leverage existing and well-known tools for these requirements with minor modifications and adjustments.

2.1. Description of Squid Caching and Shoal Discovery Services

The CERN Virtual Machine File System (CVMFS) [2] is widely adopted by the High Energy Physics (HEP) community for the distribution of project software. CVMFS is a read-only network file system that provides access to files from a CVMFS Server over HTTP. Though initially developed for virtual machines it is also used in non-virtualized environments as well. When CVMFS is used on a cluster of worker nodes, the Squid HTTP web proxy can be used to cache the file system contents, so that all subsequent requests for that file will be delivered from the local HTTP proxy server. Typically, a HEP computing site has a local or regional Squid HTTP web proxy [3], with the central CVMFS servers located at the main laboratory, such as CERN for the LHC experiments. In a remote cloud deployment we have found it necessary both for security reasons and for network latency reasons to co-locate the squid web proxies in the cloud.

In IaaS cloud resources the compute nodes and the squid caching proxies are launched dynamically with addresses that are not predictable in advance. We need to identify new methods to enable the compute nodes to dynamically discover the Squid services and other services which may be needed, and reconfigure the compute nodes to use these services. The Shoal mechanism was developed at the University of Victoria for these purposes [4,5]. We use Shoal as a service that can dynamically publish and advertise the available Squid servers. Shoal is ideal for an environment using both static and dynamic Squid servers, and distributed across multiple locations including on-site machines as well as commercial clouds.

Shoal is divided into three logical modules, a server, an agent, and a client, and is available via the python package index. The Shoal Server maintains a list of active Squid servers in volatile memory and receives AMQP messages from them. It provides a RESTful interface for Shoal Clients to retrieve a list of geographically closest Squid servers. It provides a web user interface to easily view Squid servers being tracked. The Shoal Agent is a daemon that runs on Squid servers to send an Advanced Message Query Protocol (AMQP) [6] message to Shoal Server on a set interval. Every Squid server wishing to publish its existence runs Shoal Agent on boot. Shoal Agent sends periodic heartbeat messages to the Shoal Server (typically every 30 seconds). The Shoal Client is used by worker nodes to query the Shoal Server to retrieve a list of geographically nearest Squid servers and adjust the configuration of the worker node appropriately. Shoal Client is designed to be simple (less than 100 lines of Python) with no dependencies beyond a standard Python installation. AMQP forms the communications backbone of Shoal Server. All information exchanges between Shoal Agent (Squid Servers) and Shoal Server are done using this protocol, and all messages are routed through a RabbitMQ [7,8] Server.

2.2. *Design and Implementation of Squid Service in the Cloud*

We deployed the software stack of CVMFS (network file system), Squid (on-demand caching service) and Shoal (squid cache publishing and advertising tool designed to work in fast changing environments) on FermiCloud (private cloud) and on Amazon Web Services (Public cloud) using Serverless Puppet (Puppet apply). As a part of this work, we developed Puppet modules and scripts for installing and deploying Shoal client, agent and server. We also modified Shoal Server such that it could publish Squid Servers running on EC2 instances where the Squid port is only visible on the internal AWS Virtual Private Cloud network, and contributed this change back to the upstream developers.

The architecture of large scale batch of dynamically instantiated FermiCloud and EC2 worker nodes provided with network file system, on-demand caching service and a cache publishing and advertising tool is shown in Figure 1.

When a new Shoal Server node is instantiated, the Shoal Server and its dependent components including Apache and RabbitMQ server are installed on startup of the machine. For this purpose we constructed a Puppet script which can be run at boot time from a normal Puppet server, or as a standalone script with Puppet apply, using the cloud-init features of AWS or the contextualization features of OpenNebula respectively.

When a Squid Server is instantiated dynamically, the Squid service and the Shoal Agent are installed on the startup of the machine, using Puppet apply. The shell script to execute the Puppet apply command is included as part of the launch of the virtual machine and uses the cloud-init features of AWS, or the contextualization features of OpenNebula respectively. We used the existing Puppet Forge modules for Squid as part of this work.

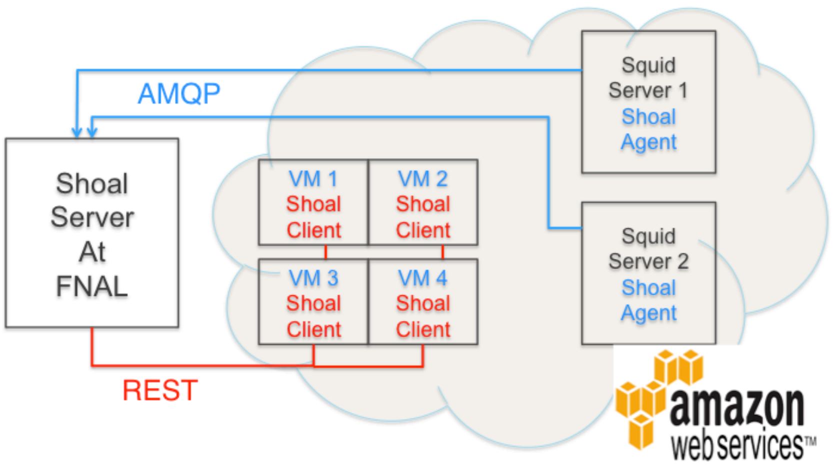

Figure 1: Architecture of Dynamically Instantiated EC2 Instances with On-Demand Caching Service and a Cache Publishing Service

When a worker node is instantiated, the Shoal Client, the CVMFS client, and other unrelated software are installed at the startup of the machine via a startup script that uses Puppet apply. This is done on FermiCloud. We use publically available Puppet modules for the CVMFS client. It could be done on AWS as well but in practice we pre-install it and send it along with our virtual machine worker node image. The Shoal Client is a cron job that queries the Shoal Server using the REST interface to get the closest Squid Server and is configured to run every 2 hours. Shoal Client updates the proxy address in the CVMFS configuration file. When CVMFS client tries to download any software from CVMFS server, the request passes through the configured Squid Server. The Shoal Client was later modified to modify the system configuration files to also send other non-CVMFS related traffic such as fetching certificate revocation lists to the dynamically detected nearest squid server. Experimental software may also need to be reconfigured to use the dynamically detected nearest squid server.

Our goal was to have the FermiCloud VMs on the private net use the on-site Squid servers and the AWS VMs use Squid servers that were instantiated on AWS. In this way we did not have to open the Fermilab Squid servers to the outside Internet and we also delivered a much faster Squid service due to decreased latency. We found that one m3.large Squid server in AWS (2 cores, 7.5GB RAM, 30 GB Disk) was sufficient to serve the load for caching code for 1000 running jobs. Any element code would only be fetched once to the AWS squid server and all other AWS virtual machines would access it from there.

2.3. Virtual machine conversion system

We also required a faster way to upload specialized virtual machines to Amazon Web Services. Our goal was to run on AWS an image as close as possible to the one used on private deployments. We leveraged our existing mechanism that manufactures our stock image for the private cloud and added extra steps to it that strip out a few Fermilab-specific configurations and add a few Amazon-specific configurations. The standard image is uncompressed from qcow2 to raw format. Then it is copied to another running image on AWS which has an extra disk mounted. The extra disk is then saved as a snapshot. Once the virtual machine image is stored on Amazon then all copies of the virtual machines are launched from that. For our worker node virtual machines we used the Amazon m3.large instance type as mentioned above because it had a good match of available cores (2) and enough scratch space (30GB SSD) to host two jobs at once.

2.4. Private cloud scalability improvements

Our private cloud, FermiCloud, was running OpenNebula 3.2 at the time. The “econe” emulation of the AWS EC2 API was functional for launching small numbers of virtual machines and had been used at times when an experiment needed a batch slot with memory larger than the then-default 2GB per batch slot on FermiGrid. There were known problems if you tried to launch 20 or more virtual machines. The file system of GFS2 on SAN was adequate for the initial 23 hosts that it served but could not be expanded to a large scale. In addition we did not have public IP address space readily available to launch 1000 more virtual machines on the public network.

140 old Dell PowerEdge 1950 machines with dual quad-core Xeon processors and 16 GB of RAM were made available for this test. A private virtual LAN was made available to these nodes for use by the virtual machines. This private LAN was assigned a routable private network which could be reached from anywhere inside Fermilab. We installed a new test head node based on OpenNebula 4.8. For an image repository we used space on our BlueArc NAS server. This proved to be quite scalable.

2.5. Job Submission and Provisioning

The NOvA experiment users used the new client/server "Jobsub" submission system to submit their jobs. This is the standard job submission software that Intensity Frontier users at Fermilab use to submit to local Fermilab resource, opportunistic use on the Open Science Grid, and the private and commercial clouds. The GlideinWMS [1] provisioning system then identifies user jobs in the queue that are suitable for running on our private cloud or on commercial clouds, and starts virtual machines on the appropriate cloud if necessary by use of the EC2 Query API. Once the virtual machine starts up, there is a process at boot time that checks the integrity of the virtual machine and starts up a HTCondor daemon. This then calls back to the batch system to declare itself available to run a job. When no further jobs are available, the HTCondor daemon exits and the virtual machine then terminates itself.

GlideinWMS is well suited to distributing workloads evenly across a number of relatively homogeneous computing platforms such as local clusters, grids, and clouds. Since running on Amazon Web Services also requires awareness of available budget, we have identified a program of work to add an external policy engine to handle the financial decision making.

2.6. Performance Evaluation

Figures 1a and 1b show the results of the final and largest trial, in which 1000 jobs ran simultaneously both on our local private cloud and on Amazon Web Services. The completion time of the jobs was relatively the same. The ramp-up factor on AWS was limited only by the submission rate that was configured into our GlideinWMS factory. The slower ramp-up time for jobs on our local cloud was more due to the virtual machine launching pattern we were using at the time, which caused all 8 of the virtual machines to launch on the same node simultaneously, significantly stressing the local disk. We have since switched to a scheduling algorithm that distributes the launch more equally across the cloud.

We ran a total of 3300 jobs on AWS in the largest trial, shown in Figure 1A. Each job generated on average 250MB of output, the total output was 467GB for which we incurred $51 in data transfer charges. We incurred $398 in virtual machine charges, with a peak of 525 virtual machines running. This work was done with "On-demand" instances. It could have been done much more cheaply with "spot pricing" from Amazon and our current program of work is now doing this on a regular basis. We also evaluated the possibility of storing the data temporarily to the Amazon Simple Storage Service (S3) before staging it back. For this use case with a fairly small number of machines and small data, it proved to be both faster and cheaper just to stage directly back to Fermilab. We expect that we will at some point need to do staging for more data-intensive tasks and have developed S3-aware data movement tools for this purpose.

3. On-demand scalable services

The summer of 2014 testing demonstrated the successful operation of a classical service discovery model of locating a Squid server. Since then we have demonstrated an automated launch of a Squid service in Amazon Web Services. This service makes use of the Elastic Load Balancer service to provide a single entry point to the service, which can have multiple Squid servers backing it up, and an Auto Scaling Group to automatically add more Squid servers when the load goes higher, and remove them when the load goes down. A CloudFormation service can be used to start the whole stack of services. We plan to use native cloud scaling mechanisms like this wherever possible in future as we address cloud workflows of 100-1000 times larger than the ones described in this work. Amazon's Route53 service is used to attach a predictable address to the scalable Squid service.

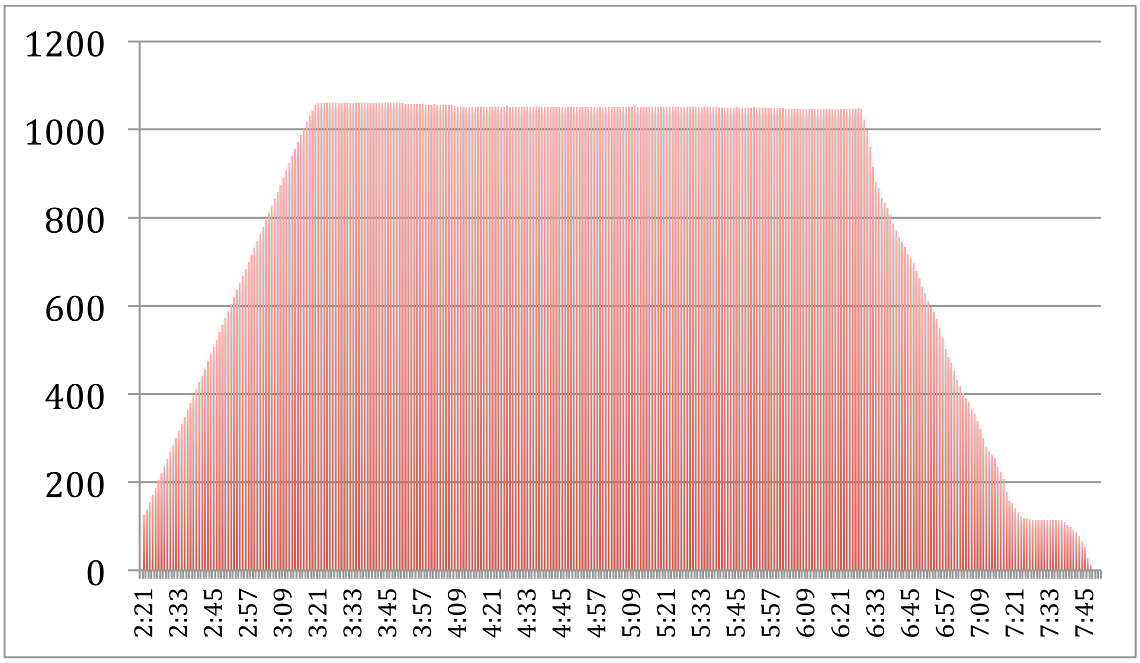

Figure 2a. Number of jobs running as a function of time of day, Amazon AWS

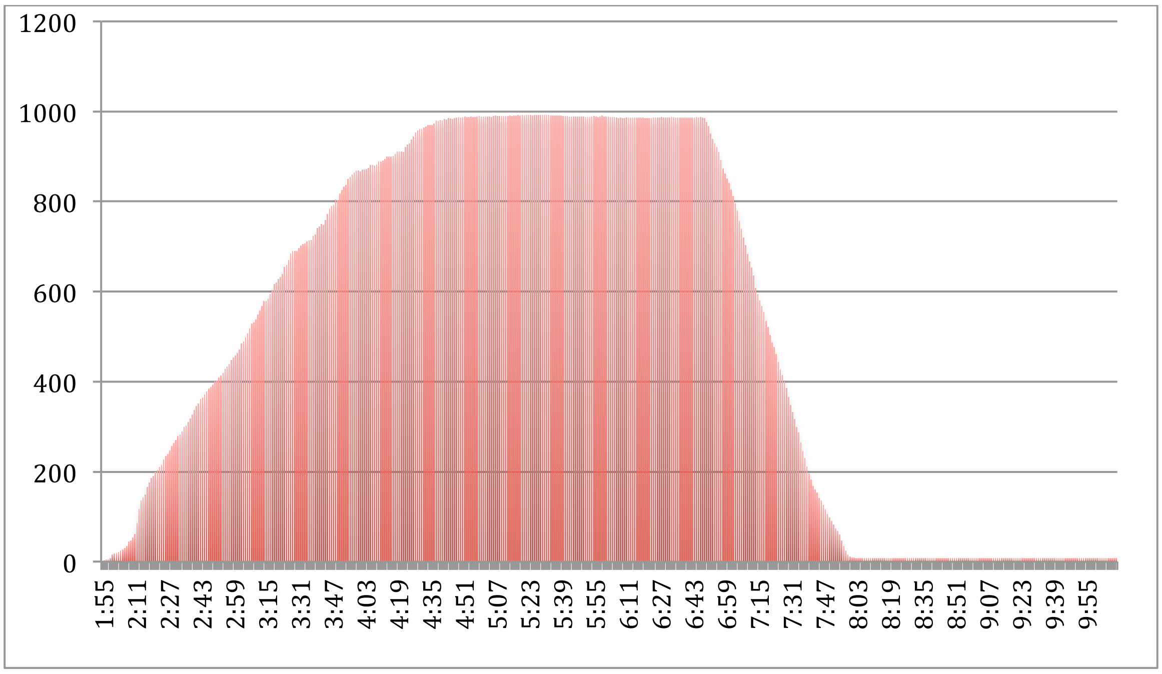

Figure 2b. Number of jobs running as a function of time of day, Fermilab private cloud

4. Conclusions and Future Work

We have demonstrated the capacity to successfully run an Intensity Frontier application at scale both in the public and the private cloud. The cloud submission capacities remain available to our production users. Our future program of work will now focus on integrating the commercial cloud providers more closely into our facility operations. The overall goal is to make the presence of the cloud resources be transparent to our end users, running even the most data-intensive applications on the cloud if necessary. We are partnering with the US research networks and commercial cloud providers to facilitate this work. We are grateful especially for all the work that was done by the NOvA experimenters to help make this happen.

Acknowledgements
Work supported by the U.S. Department of Energy under contract No. DE-AC02-07CH11359, and by CRADA FRA 2014-0002 / KISTI-C14014.

References

[1] Sfiligoi, I., Bradley, D. C., Holzman, B., Mhashilkar, P., Padhi, S. and Wurthwein, F. (2009). "The Pilot Way to Grid Resources Using glideinWMS", 2009 WRI World Congress on Computer Science and Information Engineering, Vol. 2, pp. 428–432. doi:10.1109/CSIE.2009.950.

[2] J. Blomer et al, Status and future perspectives of CernVM-FS J. Phys.: Conf. Ser. 396052013, doi:10.1088/1742-6596/396/5/052013

[3] Squid - HTTP proxy server http://www.squid-cache.org, 2015

[4] Gable, Ian, et al. Dynamic web cache publishing for IaaS clouds using Shoal. *Journal of Physics: Conference Series*. Vol. 513. No. 3. IOP Publishing, 2014.

[5] I. Gable et al, A batch system for HEP applications on a distributed IaaS cloud J. Phys.: Conf. Ser. 331062010, doi:10.1088/1742-6596/331/6/062010

[6] Python Package Index https://pypi.python.org/, 2015

[7] RabbitMQ - AMQP Messaging software, http://www.rabbitmq.com, 2015

[8] S.Vinoski, Advanced Message Queuing Protocol, IEEE Internet Computing 1087, doi:10.1109/MIC.2006.116

[9] A. Norman, The NOvA Experiment, A Long-baseline Experiment at the Intensity Frontier. PoS, HQL2012 (2012).