Architecting a Large-scale Elastic Environment
Recontextualization and Adaptive Cloud Services for Scientific Computing

Paul Marshall[1], Henry Tufo[1,2], Kate Keahey[3,4], David La Bissoniere[3,4] and Matthew Woitaszek[2]
[1]*Department of Computer Science, University of Colorado at Boulder, Boulder, CO, U.S.A.*
[2]*Computer Science Section, National Center for Atmospheric Research, Boulder, CO, U.S.A.*
[3]*Computation Institute, University of Chicago, Chicago, IL, U.S.A.*
[4]*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, U.S.A.*

Keywords: Infrastructure-as-a-Service, Cloud Computing, Elastic Computing, Recontextualization.

Abstract: Infrastructure-as-a-service (IaaS) clouds, such as Amazon EC2, offer pay-for-use virtual resources on-demand. This allows users to outsource computation and storage when needed and create elastic computing environments that adapt to changing demand. However, existing services, such as cluster resource managers (e.g. Torque), do not include support for elastic environments. Furthermore, no recontextualization services exist to reconfigure these environments as they continually adapt to changes in demand. In this paper we present an architecture for a large-scale elastic cluster environment. We extend an open-source elastic IaaS manager, the Elastic Processing Unit (EPU), to support the Torque batch-queue scheduler. We also develop a lightweight REST-based recontextualization broker that periodically reconfigures the cluster as nodes join or leave the environment. Our solution adds nodes dynamically at runtime and supports MPI jobs across distributed resources. For experimental evaluation, we deploy our solution using both NSF FutureGrid and Amazon EC2. We demonstrate the ability of our solution to create multi-cloud deployments and run batch-queued jobs, recontextualize 256 node clusters within one second of the recontextualization period, and scale to over 475 nodes in less than 15 minutes.

1 INTRODUCTION

Utilization of resources, such as a batch-queued cluster, fluctuates as users gather data, setup their applications and simulations, execute them, and analyze the results. Demand for resources typically bursts when users are actively debugging their applications and running simulations to generate results for a deadline. Demand often decreases between deadlines when users focus on gathering data or writing code. Physical resource deployments, however, only provide static capacity. If a resource is under-utilized, compute cycles that could be used to run scientific simulations are effectively wasted. At other times, demand for the resource may surpass the capacity of the static physical resource, resulting in increased queue wait times and possibly missed deadlines.

With the recent introduction of infrastructure-as-a-service (IaaS) clouds (Armbrust, 2009), users can choose to outsource computation and storage when needed. IaaS clouds offer pay-for-use virtual infrastructure resources, such as virtual machines (VMs), to users on-demand. On-demand resource provisioning is an attractive paradigm for users working toward deadlines or responding to emergencies. Users in the scientific community, in particular, have begun to adopt IaaS clouds for their workflows (Juve 2010; Rehr 2010; Wilkening 2009; Jackson, 2010). Additionally, the pay-for-use charging model means that a resource provider can choose to reduce his initial purchase of capital equipment, selecting a resource that meets the needs of his users the majority of the time, and opt to budget for future outsourcing costs. When the demand of the physical resource exceeds its capacity, the resource provider can elastically extend the static resource with IaaS resources or deploy a standalone elastic environment in the cloud to process excess demand. In previous work we developed a cloud charging model for high-performance compute (HPC) resources (Woitaszek, 2010), allowing a resource provider to analyze the cost of cloud resources in the context of physical

Marshall P., Tufo H., Keahey K., La Bissoniere D. and Woitaszek M..
Architecting a Large-scale Elastic Environment - Recontextualization and Adaptive Cloud Services for Scientific Computing.
DOI: 10.5220/0004081704090418
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 409-418
ISBN: 978-989-8565-19-8

HPC resource deployments. The virtual nature of IaaS clouds is another advantage for users. With virtual resources, users can customize the entire software stack, from the operating system (OS) upward, often a key requirement for complex scientific workflows.

IaaS clouds provide the underlying building blocks for elastic computing environments. However, policies are needed to ensure the environment adjusts appropriately to demand; tools to deploy and manage the environments are also needed. In (Marshall, 2012) we propose policies for elastic environments that balance user and administrator requirements; we evaluate them using workload traces and a discrete event simulator. In (Marshall, 2010) we developed a prototype elastic IaaS environment that extended a PBS/Torque (Bode, 2000) queue with Nimbus (Keahey 2005; Nimbus, 2012) and Amazon EC2 clouds (Amazon Web Services, 2012). However, the prototype had a number of limitations. It was not sufficiently scalable, scaling to 150 nodes in 60 minutes. It also lacked the ability to recontextualize, preventing the environment from executing parallel jobs. And it could not leverage multiple clouds simultaneously.

In this paper we address these limitations of our previous prototype. We present an architecture for a large-scale elastic computing environment that is highly available, scalable, and adapts quickly to changing demand. We extend an open-source elastic IaaS manager to support the Torque batch-queue scheduler, allowing existing scientific workflows to integrate with elastic IaaS environments for minimal software engineering cost. To support parallel jobs, we develop a lightweight REST-based contextualization broker that recontextualizes the cluster (e.g. exchanges host information between nodes) as nodes join and leave the environment. For evaluation, we deploy our solution using NSF FutureGrid (FutureGrid, 2012) and Amazon EC2. Our solution leverages multiple clouds simultaneously, recontextualizes 256 node clusters within one second of the recontextualization period, and scales to over 475 nodes in less than 15 minutes.

2 APPROACH

In previous work (Marshall, 2010) we discuss the specifics required to extend a physical cluster with IaaS resources. Therefore, in this paper we will only briefly discuss our previous prototype and its limitations before presenting our large-scale architecture and implementation.

2.1 Initial Prototype

Our initial prototype (Marshall, 2010) implemented a standalone service that created elastic IaaS environments, providing valuable insight into the challenges faced by large-scale elastic computing environments. The prototype elastic manager service monitored a Torque queue and responded by launching cloud instances, which joined the cluster and executed jobs, and then terminating idle instances once all jobs completed. The Nimbus context broker (Keahey, 2008) contextualized the cloud nodes with the head node, exchanging host information and SSH keys. However, the prototype contained a number of design and implementation limitations that prevented it from scaling appropriately, using multiple cloud providers simultaneously, and recontextualizing the environment.

In particular, the prototype elastic manager service called Torque commands directly to gather information about the queue, requiring that both the service and Torque run on the same system. It also called the Java-based Nimbus cloud client, which polls continuously for updates (sometimes taking up to a few hundred seconds), to perform the actual launches and terminations. This meant the system could only launch approximately a dozen nodes every few hundred seconds. Additionally, the initial prototype used a single cluster image with preinstalled software for each cloud provider. Due to VM image compatibility differences between cloud providers, adding support for a new cloud required installing and configuring the cluster software from scratch in a base image on the new cloud. The prototype deployment also used the Nimbus context broker, which facilitates the exchange of host information between nodes in a context, for contextualization. The Nimbus context broker, however, does not allow additional nodes to join the context after the initial launch. This meant that each node exchanged host information and SSH keys with the head node but not with each other, preventing the system from running parallel jobs. Lastly, the prototype elastic manager service was not highly available; it did not contain any self-monitoring or self-repairing capabilities.

2.2 Large-scale Elastic Computing Environments

In this section we build on our prototype architecture in (Marshall, 2010) and present an updated architecture for large-scale elastic computing environments.

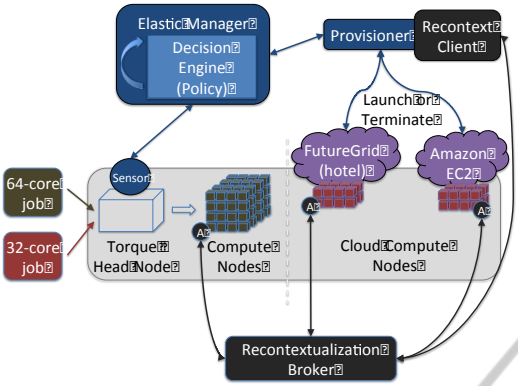

Figure 1: Example elastic Torque deployment.

We address the challenges discussed in the previous section, focusing primarily on improving scalability, leveraging multiple clouds, and developing a solution for recontextualization.

2.2.1 Elastic Management

Instead of tightly integrating the application sensor with the elastic manager service, we select a distributed design as shown in Figure 1.

We extend an open-source elastic manager service, the Elastic Processing Unit (EPU) (Keahey, 2012), under development by the Ocean Observatories Initiative (OOI) (OOI EPU, 2012). The EPU aims to be a highly available and scalable service. It is publicly available on GitHub (GitHub EPU, 2012) as an alpha release for initial testing. It contains the necessary framework and functionality to create elastic computing environments. The EPU consists of three major components that we leverage for our elastic environments: sensors to monitor application demand, a decision engine that responds to sensor information, and a provisioner that interfaces with various cloud providers to launch, terminate, and manage the instances. The provisioner interfaces with IaaS clouds using their REST APIs. Unlike the Java-based cloud client, these operations return quickly, typically within a second or two. The decision engine loops continually, processing sensor information and executing a policy that elects to launch or terminate a specific number of instances. We refer to each loop iteration as a policy evaluation iteration. The sensors are deployed throughout the environment and monitor demand (e.g. jobs in a queue).

In the EPU's current implementation, however, it only supports a "pull" queue model where individual workers request tasks from a central queue. Therefore, we extend the OOI EPU model to support a "push" queue model where a central scheduler, such as a batch-queue scheduler, monitors worker instances and dispatches single-core or parallel jobs to workers in the environment. In our model, the sensors both monitor demand and execute commands on behalf of the decision engine.

2.2.2 Automating Deployment and Configuration

To support multiple cloud providers seamlessly, the installation and configuration of worker nodes should be automated using a system integration framework, such as Chef (Jacob, 2009), instead of preinstalling and preconfiguring VM images manually. With this approach, a base image that is likely available on any cloud (e.g. a Debian 5.0 image) can be used. When a new worker boots the base image, the system integration framework downloads the cluster software, installs it, and configures the worker to join the cluster head node automatically. This can be a lengthy process if a lot of software needs to be installed, therefore, software packages can either be cached on the node or they can be completely installed, allowing nodes to boot quickly and be appropriately configured.

2.2.3 Contextualization

Creating a shared and trusted environment, or context, is a key requirement for elastic computing environments. To support parallel jobs, all resources in the environment need to exchange information and data. However, with elastic computing environments, contextualization is a continual process since demand fluctuates constantly with nodes joining and leaving the environment. Perhaps the most salient example is a context where nodes exchange SSH keys to support SSH host-based authentication. In this case, all nodes must add the hostname, IP address, and SSH key of all other nodes in the context to its ssh_known_hosts file. Existing contextualization solutions, such as the Nimbus context broker, provide mechanisms to exchange host information between all nodes that are launched at the same time. However, the Nimbus context broker does not support recontextualization of existing contexts, that is, it doesn't allow additional nodes to join the context at a later time.

We propose a recontextualization service that periodically exchanges information between all nodes in the context. The recontextualization service maintains an ordered list of nodes in the context. The list includes hostnames, IP addresses, SSH keys, and a generic data field for all nodes. As nodes join or

leave the context, the recontextualization service updates the ordered list. Every node in the context periodically checks in with the recontextualization service. If there are updates to the context, the recontextualization service sends the updated section of the list to the node, which applies the updates in order (e.g. adding the SSH keys of nodes that joined the context to the SSH known hosts file).

3 IMPLEMENTATION

For our elastic cluster implementation we extend the EPU to support the Torque batch-queue scheduler. We also create a set of scripts to setup and configure worker nodes automatically. Lastly, we develop a service for recontextualization that periodically exchanges host information between all nodes in the environment. An example deployment, shown in Figure 1, consists of the main EPU components (a sensor, decision engine, and provisioner), our recontextualization service, and a deployment across multiple IaaS clouds.

3.1 EPU Extensions

The EPU uses the Advanced Message Queuing Protocol (AMQP) (Vinoski, 2006) to communicate between its various components. In the EPU's current implementation it only implements sensors for AMQP queues, which use a "pull" queue model. Many scientific workloads would need a significant investment in software engineering to support AMQP natively. Therefore, we develop a custom EPU sensor to integrate with a widely used cluster resource manager, Torque, thereby allowing any Torque-based workload to use the elastic environment seamlessly. We also develop a custom decision engine to respond to the Torque sensors.

The Torque sensor, written in Python, monitors the queue and sends information to the decision engine. The pbs_python package (pbs_python, 2012), a Python wrapper class for the Torque C library, is used to gather Torque information. This includes job information, e.g., the total number of queued jobs and the total number of cores requested by queued jobs. It also includes node information, e.g., a list of Torque worker hostnames along with the current state of the node (free, busy, offline, down, etc.). The Torque sensor also executes commands sent to it by the decision engine. For example, when the EPU service launches a new instance, it sends the hostname to the sensor on the Torque head node and instructs it to add the node to Torque's host file. Other commands include marking a Torque node as offline and removing the node.

The custom EPU decision engine elects to launch or terminate instances. At each policy evaluation iteration, the decision engine performs a number of actions: it determines whether or not to launch instances based on queued jobs and available workers, instructs the Torque sensor to add any recently launched instances to Torque, instructs the sensor to mark any idle Torque workers as offline in preparation to terminate them, and finally, it terminates any nodes that were previously marked offline. To determine the number of instances to launch, the decision engine first calculates the total number of available workers, that is, the number free Torque instances plus the number of pending instances (those that have launched but have not yet joined the Torque environment). The number of instances launched by the decision engine is then simply the total number of cores requested by queued jobs minus the number of available workers. If the result is a positive integer, the decision engine notifies the provisioner of the number of instances to launch. The decision engine also replaces stalled instances after 10 minutes (a configurable option). Pending instances are considered stalled if there are no changes to them within 10 minutes.

3.2 Automated Worker Deployment

After the EPU launches worker instances, software packages need to be installed, the instances need to be configured as Torque worker nodes, and they need to join the cluster. We use the Chef (Jacob 2009; Chef 2012) systems integration framework to download, install, and configure cluster and user software. Chef uses "recipes" to automate system administration tasks, such as downloading Linux packages, installing software, or running bash scripts. For our elastic environment we develop a set of Chef recipes to download, compile, and install Torque, the pbs_python package, and user software. This allows us to use any base Linux image on any IaaS cloud as a worker image, greatly reducing the time required to support additional clouds. When the base Linux node boots, a simple Python agent, bundled in the image ahead of time, automatically retrieves the Chef recipes from a GitHub repository and then executes them, installing and configuring the node from scratch. To speed up the deployment and limit possible points of failure, the recipes and software can be cached in the base image.

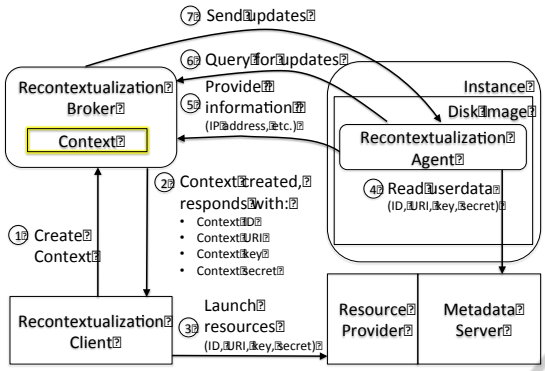

Figure 2: The recontextualization process.

3.3 Recontextualization Broker

The recontextualization broker is the final component of our large-scale elastic computing environment. It is a lightweight, REST-based recontextualization service that securely exchanges host information between all nodes in the same context. It is also important to note that while our recontextualization solution was developed specifically for use with IaaS clouds and VMs, it can also be used on physical resource deployments.

All components of our recontextualization solution are written in Python and use representational state transfer (REST) over HTTPS for communication; symmetric keys are used for both user and context security. Our solution consists of three components: a client for managing contexts, an agent that runs on all nodes in the context, and a central broker service to facilitate the secure exchange of host information between agents. The information exchanged by nodes in the context includes the short hostname, full hostname, IP address, SSH public host key, and a generic text data field. In the current implementation, the broker stores this information in a RAM-based SQLite database, although this could easily be changed to a more robust database solution. The agent contains a set of scripts (e.g. bash), typically written by the administrator deploying the agent, which allows the administrator to customize how the agent should process updates to the context (e.g. by adding other node's SSH keys to the ssh_known_hosts file). The scripts are grouped into four categories: initialization scripts that are called by the agent only once, when it first starts, "add node" scripts that are called by the agent when it is processing an update for a node that has been added to the context, "delete node" scripts that are called by the agent when it is processing an update for a node that has left the context, and restart scripts that are called for all updates.

The recontextualization process is shown in Figure 2. Recontextualization begins when a user creates a context with the client, which sends a request to the broker service. The broker responds with the newly generated context ID, a uniform resource identifier (URI) for the context (e.g. context 1 would have the following URI: https://hostname: port/ctx/1), and a unique context key and secret. The context key and secret are simply random strings and allow only those that know them to join the context. When the user launches a cloud instance, this information is passed to the instance via the IaaS cloud's userdata field. The agent starts when the instance boots and reads the instance's userdata field to get the context information. The agent then sends its node information (hostnames, etc.) to the broker. The broker maintains an ordered list of nodes that join or leave a context, beginning with list ID 0, signifying no updates. Each entry in the list denotes whether the node joined or left the context.

After the agent sends its information to the broker, it enters into a loop referred to as the recontextualization period, requesting updates from the broker and applying them. To request updates, the agent sends its current ID in the ordered list, beginning at 0, to the broker. The broker compares the node's current list ID to the latest list ID for the context. If they do not match, the broker sends the updated portion of the list to the requesting agent. The agent then applies the updates, in order, by calling its scripts for each entry in the list of updates. After the agent applies the updates, it calls the restart scripts to restart any services impacted by the updates and then sleeps for a short time before looping again. The amount of time the agent sleeps between loop iterations determines the frequency that the environment recontextualizes. Highly adaptive environments should loop frequently whereas relatively stable environments would loop less often.

4 EVALUATION

For evaluation, we examine the reactivity of the environment, its ability to recontextualize quickly, and its scalability. We do not compare the performance of different cloud providers or instance types, which are tied directly to the performance of the underlying hardware and software configuration of the particular cloud. Other studies have examined the performance of virtual environments and IaaS clouds (Huang 2006, Gavrilovska 2007, Ostermann 2010, He 2010, Ghoshal 2011). And in November 2011, Amazon EC2 ranked #42 on the Top500

(Top500, 2012). Users with applications that have strict performance requirements should select appropriate clouds for their deployment.

NSF FutureGrid and Amazon EC2 are used for the evaluation. On FutureGrid we use the Hotel (fg-hotel) system, at the University of Chicago (UC), and Sierra (fg-sierra), at the San Diego Supercomputer Center (SDSC). Both systems use Xen (Barham, 2003) for virtualization. We deploy the recontextualization broker and cluster head node in separate VMs on Hotel for all tests. The recontextualization broker runs inside a VM with 8 2.93 GHz Xeon cores and 16 GB of RAM. The head node runs on a VM with 2 2.93 GHz Xeon cores and 2 GB of RAM. The head node contains the Torque 2.5.9 server software, Maui 3.3.1 (Jackson 2001, Maui 2012), as well as the EPU provisioner and decision engine. The setup process for the head node is completely automated using the cloudinit.d tool (Bresnahan, 2011). We created a cloudinit.d launch plan to deploy a base VM on Hotel and then install and configure the software, allowing us to repeatedly deploy these environments with a single command. Worker nodes are deployed on both Hotel and Sierra and consist of 2 2.93 GHz Xeon cores with 2 GB of RAM. For EC2 worker nodes we use 64-bit EC2 east micro instances, primarily due to cost, since Amazon only charges two cents per hour. Micro instances use Elastic Block Storage (EBS) and contain *up to* 2 EC2 compute units with 613 MB of RAM. (An EC2 compute unit is defined as the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.) Deployment and configuration of worker nodes is completely automated as well. Worker nodes use a base Linux image on each cloud and Chef installs and configures the software, specifically, the Torque 2.5.9 worker software (pbs_mom) and an EPU sensor. Worker nodes also use NFS to mount the /scratch directory on the head node.

For all tests, the EPU sensor queries Torque every 60 seconds to gather job and worker information. The EPU decision engine executes every 5 seconds, querying IaaS clouds for changes in instance state and executing its policy. Recontextualization agents send their information to the broker when they first launch and query the broker every 120 seconds for updates. For batch-queued workloads, we believe that a 120 second recontextualization period is sufficiently adaptive. Many scientific workloads contain jobs that run for hours, if not days.

As a metric, we define the *reactivity time* to be the time from when the first job is submitted until the time the last job begins running for a group of jobs submitted at the same time. In the case of MPI jobs, enough cores must be available to run all tasks for all jobs. We also define the metric *recontextualization time* to be the time from when a new node attempts to join a context by sending its information to the broker until the time when all nodes in the context have received and applied the update for the new node. For example, if 256 nodes are running and all 256 nodes are aware of each other and have exchanged information, and then a new node attempts to join the context, the recontextualization time is the amount of time from when the new node sends its information until the time the other 256 nodes receive the update and apply it. In addition to these metrics, we examine the ability of the elastic environment to leverage multiple clouds simultaneously and scale to hundreds of nodes, shown with a series of traces.

For the reactivity and recontextualization tests, the workloads consist of a simple MPI application that sleeps for 60 minutes. For the recontextualization test, 60 minutes is more than enough time for the initial nodes to stabilize and wait for an additional node to launch and join the context. For the multi-cloud and scalability tests, the workloads consist of individual "sleep" jobs that sleep for 30 minutes, demonstrating the ability of the environment to scale up and down as demand changes.

4.1 Understanding System Responsiveness

To understand system responsiveness we consider two metrics. First, we examine the reactivity time. This includes the time to detect the change in demand, execute the policy, request instances from an IaaS provider, and wait for the instances to boot. Once the instances boot, they must then install and configure worker software and join the cluster. To measure reactivity time, we configure the EPU to launch EC2 east micro worker nodes (single core). We perform a series of tests, beginning with 2 node clusters, increasing to 256 nodes, shown in Figure 3. For each test we submit a simple MPI sleep job for the desired number of nodes, causing the environment to launch the needed number of nodes. We run each test 3 times for each cluster size.

As we can see in Figure 3, small cluster sizes, from 2 nodes through 16, all have relatively similar reactivity times. However, interestingly, 32 and 64 node clusters each have one test with a reactivity time similar to smaller clusters while the other 2 tests are much higher. The reason for this is related to the fact that our decision engine detects and re-

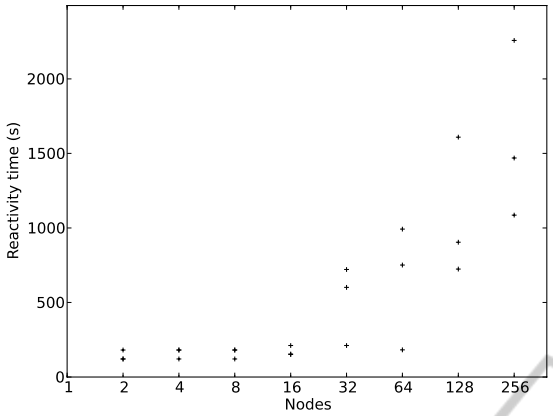

Figure 3: Reactivity time, showing three data points for each cluster size (those showing fewer are cases where values overlap).

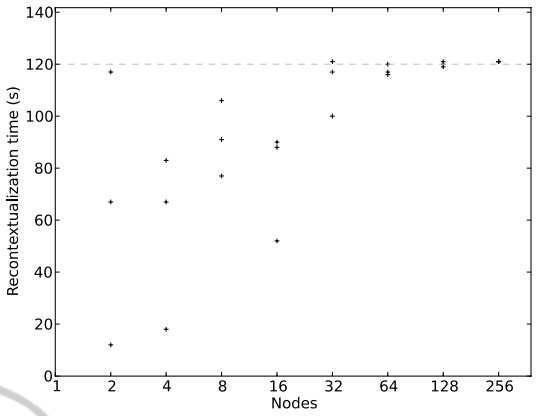

Figure 4: Recontextualization time, showing three data points for each cluster size (those showing fewer are cases where values overlap).

places stalled nodes after 10 minutes. In our evaluation we observed that EC2 micro instances would periodically fail (most often the instance failed to boot or access the network). Because larger cluster sizes boot more instances, they are more likely to encounter these failed instances, even when booting replacements. It should be noted that a small cluster could encounter a failed instance even though we didn't experience this in our evaluation.

The second area we explore is the recontextualization time where all nodes in the context must query the broker for updates, receive the updates and apply them. To measure the recontextualization time, we again configure the EPU to use EC2 east micro instances (single core). Similar to the reactivity tests, we perform a series of tests from 2 nodes through 256, running 3 tests for each cluster size. We submit a simple MPI sleep job that requests the appropriate number of cores and allow the cluster to boot, contextualize, and begin running the job. Once the cluster stabilizes, we submit another single-core sleep job, causing an extra instance to launch and join the cluster. We measure the time from when the new instance sent its information to the broker until the time when all nodes in the environment receive and apply the update. We rely on Amazon's ability to synchronize the clocks of instances, which is done through NTP, for our time-based measurements.

As we can see in Figure 4, all clusters fully recontextualize within 1 second of the 120-second recontextualization period. The tests that recontextualize before the 120 second period do so simply because all of the existing nodes in the environment check in with the broker shortly after the new node has sent its information to the broker. The three 2-node cluster tests are perhaps the most interesting case. For one of the tests, both nodes check in with the broker shortly after the new node joined the context and contextualize in less than 20 seconds, whereas with another 2-node test, at least one of the nodes waits almost the full 120 second period before it queries the broker again. This is simply the result of the random timing of when nodes boot and install the required software. As the number of nodes in a context increases, the likelihood that at least one of the nodes will query the broker 120 seconds after the additional node joins the context increases.

4.2 Multi-cloud and Scalable Elastic Environments

In addition to system responsiveness, we also examine the ability of the elastic environment to scale up and down as demand fluctuates. For our first test, we configure the EPU to launch workers on Hotel and then we submit 256 single-core jobs that sleep for 30 minutes (not shown). Hotel's workers are dual core instances with 2 GB of RAM. The elastic environment scales up to over 100 VMs, however, it doesn't quite reach 128 VMs (or 256 cores), because Hotel's underlying hardware is unable to deploy 128 VMs within 30 minutes. As the first jobs begin to complete, those instances become free and run the remaining jobs.

In our second test, we configure the EPU to dispatch workers on both Hotel and Sierra, shown in Figure 5. Both Hotel and Sierra deploy dual core workers with 2 GB of RAM. The workers from both clouds are configured to trust each other and process the same jobs from the main queue. (However, it should be noted that if the jobs or workflow are not amenable to multi-cloud environments (typically connected over the Internet with relatively high

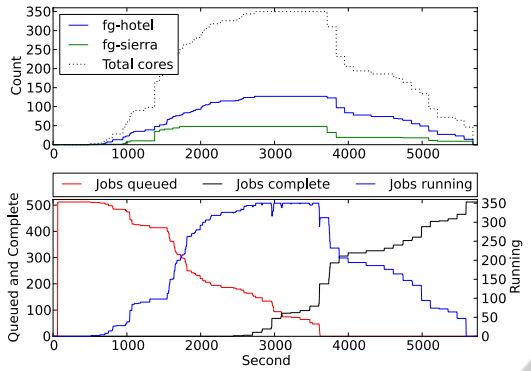

Figure 5: Multi-cloud trace running 512 single-core jobs.

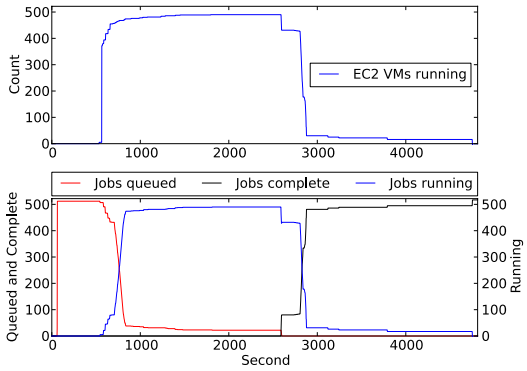

Figure 6: Scalability test on EC2 running 512 single-core jobs.

latencies), that it is possible to configure the environment to restrict parallel jobs to individual resource infrastructures using either node attributes or multiple queues with routing.) For evaluation, we submit 512 single-core jobs that sleep for 30 minutes. In this case we observe the limited scalability of private clouds (likely due to other users on the cloud). Hotel and Sierra are not able to provide 512 cores, as both clouds reach the maximum number of instances that they can deploy at the time, shown by the horizontal line for VMs running on each cloud. Sierra reaches this point just before 2,000 seconds into the evaluation while Hotel reaches this point just before 3,000 seconds.

In our final test, we configure the EPU to deploy single-core EC2 east micro instances, shown in Figure 6. We submit 512 jobs that each request a single core and sleep for 30 minutes. The environment scales to 476 instances in less than 15 minutes. The key difference in the ability of the EC2 test to scale quickly, compared to the FutureGrid tests, is related to the underlying hardware and software configuration differences between EC2 and FutureGrid. Unfortunately, EC2 was not able to reach 512 instances within the first 30 minutes of the evaluation. While the environment scales quickly, it begins to trail off around 400 instances, only reaching a total of 490 instances. The remaining 22 instances simply fail to boot completely on EC2, even after the EPU detects stalled instances and tries to replace them, multiple times. To date we have not been able to diagnose the reason, however, we have not observed any problems with the EPU or recontextualization broker. (Our EC2 instance limit is set well beyond 490.) Finally, it is worth noting that while we use cheap single-core EC2 micro instances for these tests (only running sleep jobs) because of their cost, all of our software and evaluation is run on a per-node basis. For example, our 490-node scalability test would result in a 5,880-core cluster if 12-core worker nodes were deployed. A 512-node cluster on Amazon with their largest cluster instance size, cc2.8xlarge, would cost $1,228.80 for one hour compared to $10.24 for a 512-node cluster of micro instances for one hour.

5 RELATED WORK

Prior to the introduction of IaaS clouds, several projects developed dynamic resource managers that adjust resource deployments based on demand (Ruth 2005, Ruth 2006, Murphy 2009). More recently, applications have begun to add support for IaaS clouds. Specifically, Sun Grid Engine (Gentzsch, 2001), now Oracle Grid Engine contains support for Amazon EC2 resource provisioning (Oracle Grid Engine, 2012). Evangelinos et al. (Evangelinos, 2008) use Amazon EC2 to support interactive climate modeling. In both cases these applications only include support for Amazon EC2, and they both bundle support for EC2 directly into the application. Application-specific approaches, such as these, require that each individual application include support for all cloud providers that users may need. Amazon CloudWatch (Amazon CloudWatch, 2012) is a cloud-specific approach for elastic resource provisioning. CloudWatch can scale environments based on demand; however, it only uses Amazon EC2. OpenNebula (Sotomayor, 2009), an IaaS toolkit, also provides a cloud-specific approach, bundling support for Amazon EC2. Our solution is more general than cloud-specific and application-specific approaches, allowing any application to share a single service and code base for IaaS resource provisioning.

Juve et al. (Juve, 2011) present a generic IaaS management solution, Wrangler, that provisions

IaaS resources across multiple clouds and deploys applications on those resources. Wrangler, however, is not an elastic management solution that adapts the environment based on changing demand; instead, Wrangler focuses on deploying relatively standalone environments for users. The University of Victoria's Cloud Scheduler (Armstrong, 2010) is perhaps the most similar to our solution. Cloud Scheduler monitors a Condor queue and provisions resources across Nimbus clouds and Amazon EC2. Currently, Cloud Scheduler does not contain a mechanism for recontextualizing nodes, and instead it focuses on high throughput computing (HTC) jobs.

6 FUTURE WORK

In future work we will investigate additional data movement solutions. Efficient data movement is a key requirement for elastic IaaS environments where cloud providers may charge (either with allocation credits or actual money) for data transfers. Data movement solutions should avoid sending excessive amounts of data or duplicate datasets to cloud resources. However, data movement solutions must also be easy to use or, if possible, completely automated.

We will also extend a physical batch-queued cluster with our elastic resource manager and examine its impact on a wide variety of workloads. The environment will integrate physical cluster resources with resources distributed across multiple cloud providers. Because some applications may have strict requirements, such as latency sensitive applications, users will be able to specify basic requirements related to application placement on the resources (e.g. specifying whether to run the application on local physical resources vs. distributed across all resources).

7 CONCLUSIONS

In this work we present an architecture for a large-scale elastic computing environment. We extend an open source elastic resource manager, the Elastic Processing Unit (EPU), to support the Torque scheduler. We also develop a lightweight REST-based recontextualization broker to exchange host information between nodes in a context, which allows the environment to support parallel jobs.

We evaluate our elastic environment by examining its ability to scale rapidly and recontextualize quickly as resources join and leave the environment. We demonstrate an environment that is able to leverage multiple clouds, recontextualize 256 nodes within one second of the recontextualization period, and scale to over 475 nodes within 15 minutes.

ACKNOWLEDGEMENTS

We would like to thank the Nimbus team, including John Bresnahan, Tim Freeman, and Patrick Armstrong for their help and advice with this work.

REFERENCES

Amazon CloudWatch. Amazon, Inc. [Online]. Retrieved January 8, 2012, from: http://aws.amazon.com/cloudwatch/

Amazon Web Services. Amazon.com, Inc. [Online]. Retrieved January 8, 2012, from: http://www.amazon.com/aws/

Armbrust M., et al., "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep., February 2009.

Armstrong P., et al., "Cloud scheduler: a resource manager for distributed compute clouds," CoRR, vol. abs/1007.0050, 2010.

Barham P., et al., Xen and the art of virtualization. SIGOPS Oper. Syst. Rev., 37:164--177, October 2003.

Bode B., et al. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. Usenix, 4th Annual Linux Showcase and Conference, 2000.

Bresnahan J., et al., Managing Appliance Launches in Infrastructure Clouds. Teragrid 2011. Salt Lake City, UT. July 2011.

Chef. Opscode. [Online]. Retrieved January 8, 2012, from: http://www.opscode.com/chef/

Evangelinos C., Hill C., "Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2," The First Workshop on Cloud Computing and its Applications (CCA'08), October 2008.

FutureGrid. [Online]. Retrieved February 29, 2012, from: http://futuregrid.org/

Gavrilovska A., et al., "High-Performance Hypervisor Architectures: Virtualization in HPC Systems," In 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007).

Gentzsch W., "Sun grid engine: towards creating a compute power grid," in Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on, 5 2001, pp. 35 –36.

Ghoshal D., et al., I/O performance of virtualized cloud environments. In Proceedings of the second international workshop on data intensive computing in the clouds, DataCloud-SC '11, 71--80, New York, NY, USA, 2011. , ACM.

GitHub EPU. GitHub. [Online]. Retrieved January 8, 2012, from: https://github.com/ooici/epu

He Q., et al. Case study for running hpc applications in public clouds. In Proceedings of the 19th acm international symposium on high performance distributed computing, HPDC '10, 395--401, New York, NY, USA, 2010. , ACM.

Huang W., et al. A Case for High Performance Computing with Virtual Machines. In Proceedings of the 20th Annual International Conference on Supercomputing, Queensland, Australia, 2006.

Jackson D., et al., Core algorithms of the maui scheduler. In D. Feitelson and L. Rudolph, editors, Job scheduling strategies for parallel processing, volume 2221, page 87-102. Springer Berlin / Heidelberg, 2001.

Jackson K. R., et al., "Seeking supernovae in the clouds: a performance study," in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 421– 429.

Jacob A., "Infrastructure in the cloud era," in Proceedings at International O'Reilly Conference Velocity, 2009.

Juve G., Deelman E.,"Automating application deployment in infrastructure clouds," Cloud Computing Technology and Science, IEEE International Conference on, vol. 0, pp. 658– 665, 2011.

Juve G., et al., "Data sharing options for scientific workflows on amazon ec2," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9.

Keahey K. and Freeman T., Contextualization: Providing One-Click Virtual Clusters, eScience 2008, Indianapolis, IN. December 2008.

Keahey K., et al., Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid. Scientific Programming Journal, vol 13, No. 4, 2005, Special Issue: Dynamic Grids and Worldwide Computing, pp. 265-276.

Keahey, K., et al., "Infrastructure Outsourcing in Multi-Cloud Environments," submitted to XSEDE 2012, Chicago, IL.

Marshall P., Keahey K., and Freeman T., "Elastic Site: Using clouds to elastically extend site resources," in IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 2010.

Marshall P., Tufo H., and Keahey K., Provisioning Policies for Elastic Computing Environments, 9th High-Performance Grid and Cloud Computing Workshop (HPGC), Proceedings of the 26th International Parallel and Distributed Processing Symposium (IPDPS 2012), Shanghai, China, May 2012 (to appear).

Maui. [Online]. Retrieved February 29, 2012, from: http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php

Murphy M., et al., "Dynamic Provisioning of Virtual Organization Clusters" 9th IEEE International Symposium on Cluster Computing and the Grid, Shanghai, China, May 2009.

Nimbus. [Online]. Retrieved January 8, 2012, from: http://www.nimbusproject.org

OOI EPU. [Online]. Retrieved February 29, 2012, from: https://confluence.oceanobservatories.org/display/syseng/CIAD+CEI+OV+Elastic+Computing

Oracle Grid Engine. Oracle. [Online]. Retrieved January 8, 2012, from: http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

Ostermann S., et al. A performance analysis of ec2 cloud computing services for scientific computing. In cloud computing, volume 34, page 115-131. Springer Berlin Heidelberg, 2010.

pbs_python. [Online]. Retrieved January 8, 2012, from: https://subtrac.sara.nl/oss/pbs_python

Rehr J., et al., "Scientific computing in the cloud," Computing in Science Engineering, vol. 12, no. 3, pp. 34 – 43, may-june 2010.

Ruth P., et al. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. IEEE International Conference on Autonomic Computing, 2006.

Ruth P., et al., VioCluster: Virtualization for Dynamic Computational Domains, Cluster Computing, 2005. IEEE International, pages 1-10, Sept. 2005.

Sotomayor B., et al., "Virtual infrastructure management in private and hybrid clouds," Internet Computing, IEEE, vol. 13, no. 5, pp. 14 –22, sept.- oct. 2009.

Top500 List. [Online]. Retrieved February 29, 2012, from: http://top500.org/list/2011/11/100

Vinoski S., "Advanced message queuing protocol," Internet Computing, IEEE, vol. 10, no. 6, pp. 87 –89, 2006.

Wilkening J., et al., "Using clouds for metagenomics: A case study," in Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on, 31 2009-sept. 4 2009, pp. 1 –6.

Woitaszek M. and Tufo H., "Developing a cloud computing charging model for high-performance computing resources," in 10th IEEE International Conference on Computer and Information Technology, Bradford, UK, June 2010.