Poncho: Enabling Smart Administration of Full Private Clouds

Scott Devoid, Narayan Desai
Argonne National Lab
Lorin Hochstein
Nimbis Systems

Abstract

Clouds establish a new division of responsibilities between platform operators and users than have traditionally existed in computing infrastructure. In private clouds, where all participants belong to the same organization, this creates new barriers to effective communication and resource usage. In this paper, we present *poncho*, a tool that implements APIs that enable communication between cloud operators and their users, for the purposes of minimizing impact of administrative operations and load shedding on highly-utilized private clouds.

1. Introduction

With the rise of Amazon EC2 and other public Infrastructure-as-a-Service (IaaS) clouds, organizations are starting to consider private clouds: using a self-service cloud model for managing their computing resources and exposing those resources to internal users.

Open source projects such as OpenStack[19], CloudStack[20], Eucalyptus[21], Ganeti[22], and OpenNebula[23] allow system administrators to build local cloud systems, offering capabilities similar to their public cloud counterparts: the ability to provision computational, storage and networking resources on demand via service APIs. The availability of these APIs provide many advantages over the previous manual approaches: applications can scale elastically as demand rises and falls, users can rapidly prototype on development resources and seamlessly transition to production deployments. On private cloud systems, these activities can occur within a more controlled environment than the public cloud: within the company's private network and without paying a third party (and presumably a profit margin) for resource usage. These systems are quickly becoming a major force in computing infrastructure.

Private clouds face different operational difficulties compared to other large scale systems such as public clouds, traditional server farms, and HPC systems. Private cloud resource management features lag those of public clouds, HPC systems and enterprise infrastructure. Most importantly, resource management capabilities lag other systems, forcing resource underutilization in many cases, and lacking the ability to enforce resource allocation priorities. But the most difficult issue faced by private cloud operators is the user model. On clouds, users become responsible for some administrative functions, while basic platform management is left to the cloud operators. There is no structured interface between cloud users and cloud operators, resulting in poor coordination between the two. These coordination problems become dire when systems are highly utilized, due to the absence of slack. This is primarily a technical issue, as similar systems are effectively used in large scale compute clusters at similar load. We propose the creation of such an interface, in order to improve effectiveness of private cloud platforms, as well as to ease the operations of these platforms. This effort is the primary contribution of this work.

At Argonne we operate the Magellan system [11], an OpenStack-based private cloud platform dedicated to computational science workloads. Magellan consists of approximately 800 compute nodes of heterogeneous configurations totaling around 7,800 cores, 30 TB of memory and 1.2PB of storage. Resource use is unmetered, but basic system quotas, such as core count, memory, storage capacity, and numbers of VM instances, are enforced. Magellan has been in operation for nearly 30 months as an OpenStack system and for much of this time was the largest deployment of OpenStack in the world. The system supports a large variety of user groups with different workloads, requirements, and expectations.

During this time, we have experienced a variety of issues caused by this lack of communication. Many of these were caused by ill-informed user expectations and high coordination costs. Initially, users drew from their experiences with single physical machines, resulting in lots of independent, unique instances. Even worse, there

was a widespread lack of understand of the ephemeral storage concept that is widely used in systems. These factors conspired to result in issues where serious user data could be (and occasionally was) lost due to the failure of ephemeral resources. These factors resulted in substantial work in case of resource failures, and caused us to be concerned in cases where service operations required termination of ephemeral virtual machines. In turn, this greatly increased our communication burden when preparing for service operations. This kind of event is representative of a larger class of events where the user support service level should be carefully considered when deploying such a system.

The cloud model for applications, that of horizontally scalable applications with robust fault tolerance, dynamic scalability, and extreme automation, is a poor match for legacy workloads, or some computational science workloads. The former architecture is ideal for cloud operators, as user services are tolerant to failures of underlying instances, while the latter is what many users need in order to achieve their goals. This mismatch is one of the challenges facing private cloud operators. Worse yet, this incongruity is hidden behind the abstractions provided by cloud APIs, limiting the ability of cloud operators and users to effectively collaborate.

This issue can, and must, be alleviated by improving the communication between users and operators. In this paper, we will discuss concrete operational and usability issues caused by this shortcoming, many of which are specific to private clouds. We will present poncho[1], a lightweight conduit for API-driven communication via instance annotations, as well as comparing it with comparable facilities in public clouds. This system is currently in the early stages of deployment, with users beginning to incorporate annotations into their workloads.

2. Operational Challenges of Private Clouds

Operationally, private clouds are distinct from public clouds and traditional computing infrastructure (HPC systems and server farms) in several ways. Private clouds are, by their nature, operated by an organization for internal users. While these are similar in many ways to public clouds, this model implies an alignment of goals between system operators and users that does not exist in the market-based interactions of public clouds. This alignment means that operators and users are invested in deriving the most institutional benefit from private cloud systems, and are expected to collaborate effectively. In many ways, this is analogous to server farms or HPC systems, where incentives are similarly aligned. The cloud user model becomes even more challenging on private clouds; responsibilities are divided responsibilities in a far more complex ways than on traditional infrastructure, and both end users and system operators are expected to collaborate. These factors combine to cause operational challenges in a variety of dimensions. We will discuss these in turn.

2.1 Private Clouds

Private clouds are motivated by a desire to have the best of all possible worlds. Effectively, organizations want the benefits of public clouds in terms of flexibility, availability, and capacity planning, with the greater than 95% utilization rates of large scale HPC systems, and performance of traditional server farms. Also, many organizations want large multi-tenant systems, which enable economies of scale unavailable in unconsolidated infrastructure. Finally, organizations want a cloud where operators and users have aligned incentives, and can collaborate on organizational goals.

As always, the devil is in the details. When building private clouds, several challenges make it difficult to realize this ideal system goal. The state of private cloud software, while improving quickly, lags behind large scale public clouds like AWS. The flexibility of the cloud resource allocation model, where users have unfettered access to resources, requires that different groups perform specialized functions: operators build the cloud platform, while users build services and applications using these resources, and the APIs that encapsulate them. These APIs are insufficient to express the full range of user goals, rather, users specify requests in resource-centric notation. The end results of this approach are a stream of requests that the cloud resource manager attempts to satisfy, with no knowledge of their relative importance, duration, or underlying use case. Because there is no conduit for user intent information, it is difficult for users and operators to coordinate effectively. Moreover, this makes direct collaboration between users and operators, a key benefit of private clouds, considerably more difficult.

[1] Ponchos are useful in circumstances directly following clouds filling.

2.2 The Private Cloud/Openstack Resource Management Model

OpenStack provides APIs to access compute, storage, and networking resources. Resource allocations in OpenStack have no time component; that is, there is no duration. This shortcoming has several important effects, all of which center on resource reclamation. First, resources can't be reclaimed by the system when needed for other work. This limits the ability of the scheduler to implement priority scheduling, as resources are committed to a request once they are awarded, until the user releases them. Second, when the system fills, it becomes effectively useless until resources are released. This disrupts the appearance of elasticity in the system; if users can't request resources and be confident in their requests being satisfied, it causes them to behave pathologically, hoarding resources and so forth. Finally, this model poses serious challenges to the effectiveness goal of private clouds, because the system can't reclaim resources that are being ineffectively used or left idle altogether. This is a distinct goal of private clouds, because resource provider and resource consumer incentives are aligned.

OpenStack only has two methods for implementing resource management policies: request placement, and quotas. Both of these methods are inadequate for multi-tenant systems, where users have competing goals. Resource placement includes methods for selection of resources when new requests arrive. These decisions are *sticky*, that is, they persist until the allocation is terminated, so they aren't useful for implementing policy in steady state operations. Quotas are a component of the solution, and are the only method to implement fairness by default. Because these quotas are static, and are hard quotas, they are a blunt instrument, and can't be used to softly change user behavior.

2.3 Private Cloud User Model and the Role of Platform Operators

One of the major features of private clouds is a reformulation of responsibilities centering on the role of users and platform operators. In the private cloud model, platform operators are responsible for the health of the underlying cloud platform, including API endpoints, and hardware infrastructure, as well as aiming to meet the SLAs for allocated resources. Users are responsible for everything that happens inside of resources. Furthermore, these resources are black boxes; platform operators don't have any visibility into user allocations, or their states. This disconnect is problematic from a variety of perspectives. First, operators are unable to accurately assess the impact of failures, terminations, and service actions. Second, operators can't determine which resources are in use for tasks important to users, versus lower priority tasks they may be running. Building a channel for communication between users and operators creates an opportunity for explicit collaboration, where only ad-hoc methods previously existed.

3. User/Operator Coordination on Private Clouds

While private clouds are a quickly growing architecture for computing resources, the current state of the art leaves several operational gaps, as described above. In order to address these issues, we propose the addition of two methods for coordination between users and operators. The first of these is an annotation method, whereby users can describe the properties of their VMs. This enables users to communicate requirements and expectations to cloud operators unambiguously. Also, these annotations allow system operators to reclaim resources and take other actions while minimizing user impact. The second component is a notification scheme whereby users are told when their resources are affected by failures, resource contention or administrative operations. Both of these mechanisms are used by the third component, which plans "safe" operations based on user annotations and notifies users as needed. In this section, we will discuss the explicit use cases this work addresses, as well as design and implementation of these features.

3.2 Use Cases

Many private cloud operations are impacted by the lack of good information flows between users and operators, as well as the basic model offered for resource management. We find that users have particular use cases for each of their instances--information that should be communicated to the cloud operators. Operators need to perform a variety of service actions on the resources that comprise the cloud and lack the tools to plan actions while minimizing user impact.

3.2.1 Instance Use Cases

Most of the activity on our system is centered around the following broad use cases. Each of these is impacted by the lack of good communication between operators and users.

Service instances - Service instances implement network accessible services. Often, these services must answer requests immediately, hence have availability

requirements, and have provisioned resources in a high availability configuration. They are managed with the help of auto-scaling software such as AWS CloudFormation or Openstack Heat. Fault tolerance is often implemented at the application layer, which can provide additional flexibility for the platform.

Compute-intensive instances - These instances perform batch-oriented computation or analysis workloads. They are throughput oriented workloads, where the results of computation are needed, but not immediately. Batch queues or task managers usually manage this workload internal to the allocation and can restart failed tasks.

Development instances - These instances have the interactive character of service instances, but none of the HA qualities; users access resources directly for development reasons. These instances are not heavily utilized, as with the previous two use cases, and are only used when the user is active. They may contain unique data in some cases.

Ad-hoc/*Bespoke* instances - These instances are the wild west. Users treat some instances like physical machines, building custom configurations and running ad-hoc tasks. These instances are the most difficult to support, as they likely contain some unique data, and may have long-running application state that could be lost in event of failures or instance reboots.

3.2.2 Operator Use Cases

Operators need to be able to perform a variety of service actions on the cloud. In both of these cases, user-visible impact must be minimized. This goal is made more difficult by the poor flow of information between users and operators.

Resource Maintenance

Components of the cloud need proactive maintenance, for reasons ranging from software updates and security patches to signs of impending failure. In these situations, operators need to effectively coordinate with users. These processed may be manually or automatically initiated, and depending on the circumstances may be synchronous (in the case of impending failures) or asynchronous (in the case of software updates that may be delayed for a limited time).

Rolling updates fall into this category. These updates need to be performed, but do not necessarily have a short-term deadline. Updates could be performed opportunistically when a resource is free, however, opportunity decreases as utilization increases. While this approach can result in substantial progress with no user-visible impact, long-running allocations prevent it from being a comprehensive solution; user-visible operations are usually required on system-wide updates.

Load Shedding

In some cases, the cloud needs available resources for new requests, requiring some resource allocations to be terminated. This can occur due to hardware failure, single tenant deadlines, or a lack of fairness in the schedule. Ideally, load shedding minimizes visible impact to user-run services, as well as the loss of local application state. In short, when resource allocations must be terminated, choose wisely. Our initial load shedding goal is to support a basic, synchronous model. More complex policies will follow as future work.

Notifications

A cross-cutting issue with all operator workflows is providing the appropriate notifications to users when actions are taken against resources. Sending an email or opening a service ticket works if an operator manually makes a few service actions during the day. But as service actions are automated, notifications must also become automated.

3.3 Design

The design of poncho is centered around the basic notion that users and operators can coordinate through a combination of resource annotations and system notifications. That is, users and operators agree to mutually beneficial coordination for operations which can potentially cause user-visible outages. These are subtly different from traditional SLAs, where the system operator agrees to provide a particular service level. Rather, in this case, users specify their goals, and the operators provide a best-effort attempt to minimize high impact changes. These goals are approached individually on a tenant by tenant basis, so inter-tenant prioritization doesn't need to be expressed here.

These goals have a few major parts. The first component encodes the impact of service actions on a given instance, and describe conditions where an action will have acceptable impact on the user workload. An example of this is "instance X can be rebooted during the interval between 10PM and 2AM", or "instance Y can be rebooted at any time". The second, closely related part describes how resources should be deallocated, when the system does so. For example, some resources should be snapshotted prior to shutdown, while others can be terminated with no loss of data. A third class of

annotations describe actions the system should take on the user's behalf, such as killing instances after a specified runtime.

The particular annotations we have chosen enable a key resource management capability: load shedding. With the addition of load shedding, more advanced resource management strategies can be implemented, where they were not previously possible. This outcome is a key deliverable of our design; its importance cannot be understated.

The other major part of poncho's architecture is a notification function. Users can register to be notified when service actions are performed. These notifications describe the resources affected, the action taken, and a basic reason for the action. For example, a notification might tell a user that "instance Z was terminated because of a load shedding event". This would signal that requests to re-instantiate the instance would likely fail. Alternatively, a notification like "instance Z was terminated due to failure" would signal that capacity is likely available for a replacement allocation request. Notifications are delivered on a best effort basis, with a limited number of immediate retries, but no guarantee of reliable delivery. As most of this information is available through default APIs in an explicit way, applications can poll as a fallback.

3.3.1 Annotation API

We have modeled instance annotations as a series of key/value pairs, stored as instance metadata via the pre-existing mechanism in OpenStack. [2] These values are described in the table below. Examples of common use cases are show in the following examples section.

Table 2 : Instance annotations, metadata

Key Name	Description
reboot_when	Semicolon delimited list of conditions, see Table 3.
terminate_when	Semicolon delimited list of conditions, see Table 3.
snapshot_on_terminate	Boolean; create a snapshot of the instance before terminating.
notify_url	URL of service receiving event notifications.
ha_group_id	Tenant-unique ID of service HA group.
ha_group_min	Minimum number of instances within the HA group.

Table 3 : Conditional grammar

Condition example	Description
"MinRuntime(duration)"	True if the instance has been running for the specified duration.
"Notified(interval)"	True if the interval has elapsed since a scheduled event notification was sent.
"TimeOfDay(start, stop, tz)"	True if the time of day is between start and stop with the optional time zone offset from UTC. Example: "TimeOfDay(22:00, 02:00, -05:00)".

These attributes specify user goals pertaining to instance reboots and termination, as well as whether instances should be snapshotted upon termination. Users can specify a notification URL where events are submitted, and a tenant-specific high availability group ID. The priority attribute is used to choose between instances when load shedding occurs. If a tenant is chosen for load shedding, and multiple instances are flagged a terminatable, these instances are ordered in ascending order by priority, and the first instance(s) in the list are selected for termination. Priority settings of one tenant do not affect which instances are shed in another tenant.

The high availability group annotations provide a limited set of features: they ensure that cloud operators do not load shed instances that are part of that group and leave it with less than the minimum number of instances allowed. In this implementation the user is still responsible for determining scale-up needs and identifying an HA group failures that occur outside of planned operations.

The conditional grammar terms shown in Table 3 describe when terminate or reboot actions have acceptable consequences to the user. If multiple predicates are specified, all must be satisfied for the operation to be deemed safe. Note that this condition is merely advisory; failures or other events may result in resource outages causing user impacting service outages regardless of these specifications. This difference is the major distinction between these specifications and SLAs.

3.3.2 Notification API

The primary goal of the notification API is to inform user about system actions that impact their instances. By annotating the instance with a "notify_url" tag, the user can specify a URL that listens for events from poncho. Events are sent as JSON encoded HTTP POST requests to the "notify_url". All events contain the following basic attributes:

- "timestamp" : A timestamp for the event
- "type" : An event type, from a fixed list.
- "description" : A descriptive explanation of why this event is happening.

Specific event types contain additional attributes, listed in Table 4.

Table 4. Description of notification event types

Event Type	Description and supplemental information
reboot_scheduled	A reboot has been scheduled. Includes the instance ID, name and reboot time.
rebooting	The instance is now rebooting. Includes the instance ID, name.
terminate_scheduled	The instance has been scheduled to be terminated. Includes the instance ID, name and a termination time.
terminating	The instance is now being terminated. Includes the instance ID and name.
terminated	The instance was terminated at some point in the past. This notification is used for service failures where the instance cannot be recovered. Includes the instance ID and name.
snapshot_created	A snapshot of the instance has been created. Includes the instance ID, instance name and the ID of the created snapshot.
ha_group_degraded	The HA group for this instance no longer has the minimum number of instances. Includes the HA group ID and a list of instance IDs for instances still active within that group. Sent once per HA group.
ha_group_healthy	The HA group for this instance has transition from degraded to healthy. Includes the HA group ID and the list of instances active within the group. Sent once per HA group.
shed_load_request	A request by the operators to the tenant to deallocate instances if possible. This is sent out once for every unique notification URL within the tenant.

Currently instances default to no notification URL. We have implemented an optional configuration of Poncho that formats messages for these instances as an email to the instance owner. For the HA group and shed-load events, messages are sent as emails to the tenant administers.

User-written notification agents are fairly simple. A server responds to the HTTP endpoint registered as a notification URL, and takes appropriate actions. While simple notification agents are fairly general, we have found that most tenants want custom policies depending on their needs.

3.4 Implementation

We implemented poncho in three parts. The first is a set of scripts that provide a user-centric command line interface to annotate nodes. The second is a notification library that is used by administrative scripts to notify userspace agents upon administrative action. The third is a set of administrative scripts that can be run interactively or periodically to shed load, service nodes, or kill time limited tasks. This final component is run periodically in our initial prototype. The primary goal of this prototype is to gain some experiences coordinating with users in a productive fashion, so the system itself is deliberately simplistic until we validate our basic model.

Our initial implementation of poncho is intended to function as a force multiplier, whereby administrators

and users perform roughly similar sorts of tasks with the aid of scripts that streamline these processes. Operators gain the ability to perform some service actions in an automated fashion, and begin to understand the impact of service options. Users gain the ability to submit allocation requests for fixed duration, with automatic termination, as well as the ability to communicate information about their workloads, like the impact of instance outages.

Poncho is an open-source Python application, leveraging existing OpenStack Python APIs and is compatible with any Openstack deployment running Essex or newer releases. It is available on Github [12].

While we hope to integrate similar functionality into Openstack, this version has been implemented in a minimally invasive fashion. Our goal in this effort is to gain sufficient experience to develop a comprehensive model for user/operator interactions. Once we have some confidence in our model, we plan to develop an Openstack blueprint and an implementation suitable for integration into Openstack itself.

3.5 Example Use Cases

For instances that have no annotations, a default annotation is assumed which meets most users expectations for cloud instances:

{ “terminate_when” : false, “reboot_when” : true }

This annotation declares instance reboots to be safe at any time, but terminations to be deemed unsafe at any time.

Running an instance for development work is a common use case on Magellan. For this case, we define a minimum runtime of twelve hours, a full day of work, before the instance can be terminated; we also enable automatic snapshotting since the user may have important work that needs to be saved. Our conservative policy is for tenants to delete unnecessary snapshots.

{ “terminate_when” : “MinRuntime(12h)”, snapshot_on_terminate : true }

For workloads that are throughput oriented, there are a number of annotation configurations that might work. The following annotation ensures that a minimum number of instances are working for the HA group, that the user is notified one hour before any scheduled events and that this instance is only considered after instances in the same tenant with a lower priority score:

{ “terminate_when” : “Notified(1h)”, “ha_group_id” : 12,
“ha_group_min” : 5, priority : 10,
“notify_url” : “http://example.com/notifications” }

4. Experiences and Discussion

An initial version of poncho has been deployed to users on Magellan. Many of our tenants are invested in helping us to develop the user model and resource management capabilities, because they notice the lack of communication, and feel they are using resources inefficiently. Initial user responses have been enthusiastic.

At this point, our two largest tenants have begun to use these interfaces. One of these tenants has a throughput dominated workload, and had previously communicated this to us in an ad-hoc manner. In effect, the interfaces provided by this system formalize a manual arrangement. Another of our major tenants has a development heavy workload, and had been looking for a system that reaped old instances after taking snapshots for several months. A third tenant, with a workload that consists of a combination of development and throughput-oriented instances has also agreed to begin using these interfaces as well. At this point, we have only tested poncho’s functionality in an artificial setting; we have not needed to shed load or service poncho-mediated resources yet; the system is fairly reliable, and all instances are not yet tagged.

Our initial experiences with users have shown two basic models. Users with throughput oriented workloads are able to integrate these methods into their workflows relatively easily, as all of their resources are started up in a uniform way. Interactive users, largely instantiating development instances, start their instances in a variety of different ways, making uniform adoption considerably more difficult. These latter kinds of instances consume resources in a bursty fashion, while occupying resources consistently. In our experiences, tenants want to set custom policies for their development instances. This approach is similar in philosophy to the one taken by Netflix’s Janitor Monkey, and consolidates policy at the tenant level, not with either the system or individual users.

It remains an open question how broad adoption of these APIs will be across tenants on our system. For this reason, it was critical to set a reasonable default set of annotations for instances. Once clear conservative option is to define both instance reboots and terminations as invasive. Another more flexible option allows re-

boots with 24 hour calendar notice emailed to users while still deeming terminations as invasive. We have chosen this latter option, as it gives users some incentive to learn these APIs and put them into use if they have a sensitive workload.

With the addition of load-shedding capabilities, we enable Openstack to implement a range of scheduling algorithms familiar from public clouds and HPC systems. This core capability is the fundamental infrastructure for AWS spot instances, and HPC system scavenger queues. We plan to explore these options for improving system productivity.

Clouds put operators/system administrators into the role of building API driven services for their organizations. These new services implicitly include a collaborative function with users, with a division of responsibilities (which is familiar to administrators) and an abstraction barrier, which is new. In our view, it is critical that cloud operators be proactive, and help to design effective coordination facilities to enable users to use resources effectively, both in throughput-oriented computational workloads and availability-oriented service workloads. APIs, which provide services to users, can just as easily be used to provide services to operators. With this sort of approach, traditionally expensive problems can be simply solved. Solutions of this kind are critical if private clouds are to grow to their full potential.

5. Background and Related Work

The problems of effective communication with users to enable efficient resource management, including load-shedding, are old ones in system management. In HPC systems, resource allocations are explicitly annotated with a maximum runtime. HPC schedulers, such as Maui[17], and Slurm[18] can use these annotations to implement scheduling algorithms such as conservative backfill[25]. The availability of maximum runtimes also enable deterministic draining of resources, a luxury unavailable on private clouds due to the private cloud resource allocation model. In [24], the authors explore instituting explicit resource leases on top of the cloud resource allocation model.

Public clouds have some features that enable effective coordination between platform operators and cloud users. Amazon's Spot Instances[15] are a prime example of that, and is specifically built on top of load-shedding techniques.

AWS and Rackspace both have an API specifically for coordinating planned outages. Instance and hypervisor reboots are scheduled and that schedule is queryable by the user. In some cases users may elect to reboot instances at a time of their choosing before the scheduled maintenance window.[1,3] This enables users to perform the required upgrades in a controlled fashion, e.g. when personnel are available to diagnose and fix unexpected issues with the upgrade.

Support for high availability service groups are widely supported; AWS and Microsoft Azure include mechanisms to build such services. [1,7] In private cloud software stacks, Openstack (via Heat[16]) and Eucalyptus both support similar mechanisms.

Several systems provide auto-scaling functionality, which could be configured to receive events from poncho. AWS includes integrated auto-scaling services, as does Azure. Heat provides related functionality for the Openstack ecosystem. Some of these systems use VM profiling tools to identify applications with high CPU or memory load to be scaled.

Public cloud operators manage user expectations with SLAs. This approach is quite effective in conjunction with variable pricing across service classes. Our approach is slightly different, using annotations to signal user desires, as opposed to making guarantees to the users.

Rightscale[26] and Cycle Computing[27] are third party resource management environments that implement advanced policies on top of public and private clouds for service and throughput-oriented workloads, respectively.

In [8], the authors propose a strategy of improving private cloud utilization with high-throughput Condor tasks, which can implicitly be terminated at any time. This strategy, similar to Amazon's spot instance strategy, is less flexible than coordination solution presented in this paper, as it solely addresses the utilization problem, not the more general resource evacuation and load shedding problems.

Netflix's Simian Army[13] contains some resource management features that we intentionally designed into poncho. While most of the Simian Army application is concerned with testing the fault tolerance of Netflix's video streaming services, the Janitor Monkey application identifies idle development instances and terminates them after warning the owner. [14]

6. Conclusions

In this paper, we have presented the design and implementation of poncho, a tool that enables better communication between private cloud operators and their users, as well as early experiences with the tool. Our initial implementation of poncho is relatively simplistic, and primarily aims to validate our model for API-driven coordination between users and cloud operators. Poncho has been deployed to users, and initial feedback has been positive, suggesting users are willing to make use of such interfaces if it makes their lives easier or enables more efficient use of computing resources.

One goal in writing this paper was to begin a community discussion of this communication breakdown. As adoption of private clouds grows, these issues will grow more serious, particularly as these systems become more saturated. As a largely non-technical issue, we believe that broad experimentation, on real users, is the best way to develop effective solutions. Moreover, it is critical that system administrators, as they become private cloud operators, remain cognisant of these issues, and strive to minimize their impact.

Finally, as service-oriented computing infrastructure, like private clouds, becomes widespread, operators (and system administrators) will increasingly find their users hidden away behind abstraction barriers, from virtual machines to PaaS software and the like. Both building effective collaborative models with users, and designing APIs that are efficient and productive, are critical tasks that system administrators are uniquely equipped to address within their organizations.

7. Future Work

This work is a set of first steps toward improved communication between users and operators on Openstack private clouds. The prototype described here is a simple implementation of such a conduit, no doubt it will be refined or even redesigned as users develop more sophisticated requirements, driven by their application workloads.

One of the critical pieces of infrastructure provided by this system is a mechanism that can be used for load shedding, as well as a way to communicate with users when this action is required. As a building block, load shedding enables a whole host of more advanced resource management capabilities, like spot instances, advanced reservations, and fairshare scheduling. After our initial assessment of this coordination model is complete, we plan to build an active implementation, that can directly implement these features.

Notifications, particularly the explicit load-shedding request, enable the creation of hierarchical cooperative resource managers, which is probably the best path forward for integration with traditional resource managers.

References

[1] **Amazon EC2 Maintenance Help**, http://aws.amazon.com/maintenance-help/ on 4/30/2013.

[2] **OpenStack API Documentation** http://api.openstack.org/api-ref.html

[3] **Preparing for a Cloud Server Migration,** retrived from http://www.rackspace.com/knowledge_center/article/preparing-for-a-cloud-server-migration on 4/30/13.

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

[5] **A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus**, Sempolinski, P.; Thain, D. *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), 2010*, pp.417-426, Nov. 30 2010-Dec. 3 2010http://www3.nd.edu/~ccl/research/papers/psempoli-cloudcom.pdf

[6] **Nova API Feature Comparison**, retrieved from https://wiki.openstack.org/wiki/Nova/APIFeatureComparison on 4/30/13.

[7] **Windows Azure Execution Models**, retrived from http://www.windowsazure.com/en-us/develop/net/fundamentals/compute/ on 4/30/13.

[8] **Improving Utilization of Infrastructure Clouds**, Marshall, P., Keahey K., Freeman, T. Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011), Newport Beach, CA. May 2011.

[9] **Manage the Availability of Virtual Machines**, http://www.windowsazure.com/en-us/manage/linux/common-tasks/manage-vm-availability/ retrived on 30 April 2013.

[10] **Troubleshooting in Windows Azure**, http://www.windowsazure.com/en-us/manage/linux/best-practices/troubleshooting/ retrived on 30 April 2013.

[11] **Magellan: Experiences from a Science Cloud**, Lavanya Ramakrishnan, Piotr T. Zbiegel, Scott Camp-

bell, Rick Bradshaw, Richard Shane Canon, Susan Coghlan, Iwona Sakrejda, Narayan Desai, Tina Declerck, Anping Liu, *Proceedings of the 2nd International Workshop on Scientific Cloud Computing, ACM ScienceCloud '11*, 2011

[12] **Poncho Github Repository**, retrieved from https://github.com/magellancloud/poncho on 4/30/13

[13] **The Netflix Simian Army**, Y. Izrailevsky, A. Tseitlin, *The Netflix Tech Blog*, 19 July 2011, http://techblog.netflix.com/2011/07/netflix-simian-army.html, retrived on 30 April 2013

[14] **Janitor Monkey: Keeping the Cloud Tidy and Clean**, M. Fu, C. Bennett, *The Netflix Tech Blog*, http://techblog.netflix.com/2013/01/janitor-monkey-keeping-cloud-tidy-and.html, retrived on 30 April 2013.

[15] **Amazon EC2 Spot Instances**, http://aws.amazon.com/ec2/spot-instances/, 30 April 2013.

[16] **Heat: A Template based orchestration engine for OpenStack**, S. Dake, *OpenStack Summit, San Diego*, San Diego CA, October 2012, http://www.openstack.org/summit/san-diego-2012/openstack-summit-sessions/presentation/heat-a-template-based-orchestration-engine-for-openstack, retrived on 30 April 2013.

[17] **Core Algorithms of the Maui Scheduler**, David Jackson, Quinn Snell, and Mark Clement, Proceedings of Job Scheduling Strategies for Parallel Processors (JSSPP01), 2001.

[18] **SLURM: Simple Linux Utility for Resource Management**, A. Yoo, M. Jette, and M. Grondona, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44-60, Springer-Verlag, 2003.

[19] **OpenStack: Open source software for building public and private clouds**, http://www.openstack.org/ retrived on 30 April 2013.

[20] **Apache CloudStack: Open source cloud computing**, http://cloudstack.apache.org/ retrived on 30 April 2013.

[21] **The Eucalyptus Open-source Cloud-computing System.** D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D.i Zagorodnov. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pp. 124-131. IEEE, 2009.

[22] **Ganeti: Cluster-based virtualization management software**, https://code.google.com/p/ganeti/ retreived on 30 April 2013.

[23] **OpenNebula: The open source virtual machine manager for cluster computing**, J. Fontán, T. Vázquez, L. Gonzalez, R. S. Montero, and I. M. Llorente, *Open Source Grid and Cluster Software Conference*. 2008.

[24] **Capacity leasing in cloud systems using the opennebula engine.** B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. "In *Workshop on Cloud Computing and its Applications*, 2008.

[25] **The ANL/IBM SP Scheduling System.** D. Lifka. In *Job Scheduling Strategies for Parallel Processing*, pp. 295-303. Springer Berlin Heidelberg, 1995.

[26] **Rightscale Cloud Management**, http://www.rightscale.com, retrieved 20 August, 2013.

[27] **Cycle Computing**, http://www.cyclecomputing.com, retrieved 20 August, 2013.