Managing Allocatable Resources

(Invited Paper)

Kate Keahey*, Pierre Riteau†, Jason Anderson‡ and Zhuo Zhen‡
*Mathematics and Computer Science Division
Argonne National Laboratory, Lemont, IL 60439
Email: keahey@anl.gov
†StackHPC Ltd, Bristol, UK
Email: pierre@stackhpc.com
‡University of Chicago, Chicago, IL
Emails: jasonanderson,zhenz@uchicago.edu

***Abstract*—Infrastructure cloud computing allows its clients to allocate on-demand resources, typically consisting of a representation of a compute node. In general however, there is a need for allocating resources other than nodes and managing them in more controlled ways than simply on demand. This paper generalizes the familiar "compute power on demand" pattern by introducing the abstraction of an allocatable resource, describing its properties, and implementation for different types of resources. We further describe architecture for a generic allocatable resource management service that can be extended to manage diverse types of resources as well as the implementation of this architecture in the OpenStack Blazar service to manage resources ranging from bare-metal compute nodes to network segments. Finally, we provide a usage analysis of this service on the Chameleon testbed and use it to illustrate the effectiveness of resource management methods as well as the need for incentives in usage arbitration.**

***Keywords*-cloud computing, advanced reservation, allocatable resources, OpenStack Blazar**

I. Introduction

Over the last decade or so, infrastructure Cloud computing [1] revolutionized how we think of resource procurement by making available remote resources via isolated containers for dynamic exclusive usage. Roughly the same time period has seen the emergence of scalable (i.e., serving large user communities) experimental systems like Grid'5000 [2], GENI [3], Emulab [4], and FutureGrid [5]. These systems implemented the concept of a scalable production testbed, i.e., production services that provide and manage many temporary "breakable environments", composed of distributed compute nodes, networks, and storage units, used for individual experimentation. In today's cloud parlance, these systems developed the concept of a "testbed as a service": while individual isolated testbeds are configured for experimentation that may get out of hand, the services that yield them are expected to be production quality. These testbeds emphasized the need for interactive experimentation, as well as co-scheduling of multiple resources of different kind, and thus time controlled access to isolated resources.

The Chameleon testbed [6], [7] provides highly configurable access to large-scale resources. The hardware consists of an investment in 15,000+ cores of homogeneous resources (Intell Haswell nodes) to support large scale experimentation, along with smaller investment in diversity including GPUs, FPGAs, storage-rich deployments, as weall as a range of different architectures. These resources are spread over two sites, University of Chicago and TACC, connected with 100G network. Users allocate them individually or in large and complex ensembles and can reconfigure them at bare metal level, boot from custom kernel if needed, or get access to serial console. By basing its infrastructure largely on OpenStack [8], a commodity open source Infrastructure-as-a-Service implementation, Chameleon demonstrated that mainstream cloud technology can be used for supporting Computer Science systems experimentation. At the same time, Chameleon extended the concepts underlying infrastructure clouds by systematizing the concept of an allocatable resource, extending it beyond handling node reservations to encompass other resources, and emphasized generalized time management of cloud resources in support of interactive and co-scheduled resource use, critical in experimentation.

In this paper, we introduce the concept of an *allocatable resource* as entity defining isolation and thus potential for exclusive usage on cloud resources; we discuss its properties and implementation for different types of resources. We then describe an architecture for a generic allocatable resources management service, as well as its implementation as the OpenStack Blazar [9] service (originally called Climate, since its inception in 2013 until mid-2014) which has been accepted as a top level OpenStack component since the fall of 2017. Blazar's implementation is adaptable to the management of diverse resources so that the service can be used in configurable setting both in conjunction with other OpenStack components (such as Nova [10] and Neutron [11]), and on its own by developing independent plugins for resources managed by services outside of OpenStack. Finally, we analyze our experiences with allocatable

resources on Chameleon demonstrating the value of advance reservations where resources are supply-constrained as well as the importance of incentives for their management.

This paper is organized as follows. In Section 2 we introduce the concept of allocatable resource and discuss its properties. In Section 3 we describe the architecture for a generic allocatable resources management service followed by a discussion of implementation of the Blazar OpenStack service in Section 4. In Section 5 we provide insights gained from allocatable resource usage on Chameleon. We describe related work in Section 6 and conclude.

II. Allocatable Resources

We define an *allocatable resource* as a well-defined object within a system that the system's clients can automatically allocate for exclusive, metered usage, delimited by well-defined time events. We will call the temporary exclusive ownership of such resources a lease. Leases can be atomic (associated with one resource only) or complex (associated with multiple resources).

We discuss below the properties of allocatable resources:

Well-defined: It is essential that the description of an allocatable resource can distinguish between any resources that can be considered different within the system. For example, if a cloud instance maps to multiple architectures, the instance itself is an allocatable resource but its deployment on a particular architecture is not. The allocatable resource description is different than descriptions that a client may input while interacting with the system which could be expressed in terms of constraint such as "node with memory of at least 2GB per core"; in this case, generality simply facilitates interactions, ultimately resolving the generic description to a specific allocatable resource mapping. This is particularly important in systems supporting experimentation where claims are made in the context of a well-defined model.

Exclusive usage: This property implies the ability to define a unit of isolation between users. Historically, roughly two definitions of this isolation were considered useful: system isolation, which presents to the user an independent system, and performance isolation which ensures that the allocatable resources present consistent performance. One of the most enabling examples of system isolation are virtual machines (VMs) [12] which emulate an individual computer system. Containers [13–15], similarly provide system isolation though of a lesser degree (e.g., unlike VMs containers may share a kernel). The GENI project defined the concept of a slice [16] which encompasses a set of connected L2 circuits and the compute resources connected to them and thus defines an isolated networking environment. System isolation does not necessarily provide performance isolation, i.e. assurance that a system will be associated with a well defined quantum of resource such as guaranteed bandwidth. This is generally hard to provide in shared environments, and thus systems that require it (e.g., platforms supporting Computer Science experimentation) often resort to defining allocatable resources at coarse grain to avoid sharing. For example, to provide performance isolation Chameleon defines compute allocatable resources as physical nodes, rather than parts of a node (which would provide finer-grain sharing but is hard to implement). The implementation of isolation is typically associated with a certain cost/overhead. For example, hypervisor hosting VMs will require resources to implement its function, or bare metal nodes have to be restored to default state between users which imposes an overhead on the length of a lease. Allocatable resource is thus whatever remains after the overhead has been consumed.

Time-bounded, metered, automatic allocation: Resources are allocatable if their availability can be bounded by well-defined time events. The most general implementation of this functionality allows clients to select specific time events between which their lease will take place; this is often referred to as advance reservations [17]. We note that *on-demand* availability is a special case of advance reservations where the start time defaults to the time at which the request is made. Resources are available only on an on-availability basis [18], e.g., at a time that cannot be reliably bounded or constrained by the client, are not allocatable by a client (though they may be allocatable by the provider as is the case in e.g., batch systems). The clients should also be able to change the placement of those events in time throughout the lifetime of a lease, whether inactive or active (i.e., without or with allocated resources). The usage thus described should be monitored, metered, and potentially limited according to those measures; the most common example of this is the specific dollar amounts that users pay under different cost models in commercial clouds, but also applies to allocations and policy constraints on usage in clouds operated within non-monetary economies such as academic clouds. Allocations that do not conform to policy/metering requirements (such as a credit or allocation limit) should not be admitted into the system. Finally, the requirement for automatic allocation is essential to ensure that a system managing allocatable resources will scale.

A desirable characteristic of a system managing allocatable resources is to provide an availability calendar: it allows users to assess the availability of a resource at any given time, though only an actual lease request can provide the transactional guarantee of a resource availability. Still, unless the transactional volume in a system is very high for a specific type of resource, the availability calendar can be an effective additional tool in resource management.

Of the properties described above, the isolation units are typically set by a system designer who selects implementation suitable to the system's objectives. Providing well-defined descriptions of those isolation units and managing them in a way that satisfies the remaining conditions is the

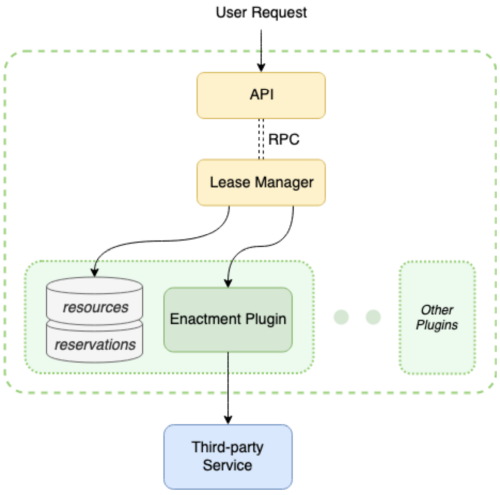

Figure 1. Architecture of allocatable resource management system

objective of an allocatable lease manager described in the rest of this paper.

III. Architecture

The architecture of an allocatable resources management system consists of three components: (1) service interfaces together with a server that can interpret it, such as e.g., HTTP-based API (Lease Interfaces), (2) functionality generic to the management of all leases (Lease Manager), and (3) functionality specific to the management of particular resources (Enactment/Resource Plugins), potentially working with third-party services to implement their functionality. We describe those components below and illustrate relationships between them in Figure 1: one lease manager can handle leases for multiple types of resources, each via a collaboration with (possibly) third party enactment services.

A. Service Interfaces

An allocatable resource manager has to support two broad categories of functions: (1) inventory management, allowing system operators to populate and manage databases of allocatable resources (such as physical nodes or floating IPs), and (2) lease management, allowing clients of the service to make and manage leases on those allocatable resources. We describe them below in turn.

The inventory management is achieved via four basic CRUD operations (create, read, update, and delete). The `create_resource` operation will generally validate provided values, fetch information about the resource and create a record in the table or database based on which resource availability is managed. The `show_resource_details` and `list_resources` functions respectively, return a specific resource description or a list of all resource descriptions of the same type. The `update_resource` allows for modifications to the original resource descriptions. Finally, the `delete_resource` operation deletes the representation of a resource in the table/database and makes it unreservable.

---

create_resource(values) $\rightarrow$ *resource*
show_resource_details(resource_id) $\rightarrow$ *resource*
list_resources() $\rightarrow$ *list of resources*
update_resource(values) $\rightarrow$ *resource*
delete_resource(resource_id)

The second type of interface allows clients to create and manage leases; the operations are as follows:

---

create_lease(constraints) $\rightarrow$ *lease*
update_lease(lease_id, values) $\rightarrow$ *lease*
delete_lease(lease_id)
show_lease_details(lease_id) $\rightarrow$ *lease*
list_leases() $\rightarrow$ *list of leases*

The `create_lease` operation is called when a lease is originally requested. The constraints argument may contain both temporal constraints (start and end time) and resource description. The resource description may include multiple units of resources of different types (e.g., multiple nodes of one type, a floating IP, and a VLAN) or provide a partial description of a resource (e.g., "a node with at least 2GB per core"), or even be omitted if reasonable defaults can be set (e.g., if the only allocatable resource type are compute nodes it may default to the most common node type); a lease is created if the constraints can be satisfied based on the existing state of the system, a lease record is persisted, and a lease ID is returned; otherwise the system will return an error. Throughout the lease lifecycle, the client can use the `update_lease` operation to update a lease by changing its resource or temporal constraints subject to resource availability and policies – or use the `show_lease_details` and `list_leases` operations to display relevant information.

While most reservations are terminated by the lease manager when their end time comes around, some can be terminated directly by the user using the `delete_lease` operation (e.g., if the work is completed sooner than originally expected); both actions trigger a call to a resource plugin operation implementing resource deallocation and graceful shutdown.

B. Lease Manager

The lease manager component provides an interface to resource and lease databases and/or tables, manages system events (such as reservation start and stop), and calls out to enactment plug-ins to allocate and deallocate resources for active reservations. This component also provides constraints management and lease monitoring throughout their lifecycle.

Resources in the inventory are stored in the resource database as individual records. A resource record consists of its unique ID, its type, and then a set of key/value metadata pairs that describe the resource in more detail, e.g. a node might store its rack position and CPU architecture, while a VLAN might store its 802.1Q tag. Leases are stored in the lease database as a lease record, with a unique ID, start time and end time, and one or more reservation records consisting of a resource type and the set of constraints specified by the user for that resource type. It is important to persist the original constraints so that additional resources satisfying them may be substituted later, or so that the user may adjust the constraints later. Separating the concept of a lease and a resource reservation provides the flexibility for one lease to cover multiple types of resources at the same time, e.g. a user can reserve both a set of nodes and a public IP address by simply associating resource records with a given lease's reservation record. In addition to the lease and reservation records, a set of lease lifecycle event records representing each phase of enactment (lease start, before lease end, lease end) are stored for each lease.

The resource assignment on lease creation may be early (final mapping to specific resources created at the time of reservation) or late (final mapping to specific resources created by the time the lease becomes active); the former leads to a simpler implementation, the latter provides more flexibility and dynamicity in optimizing assignments for various queries and adapting to resource changes. In either case at creation time the database query should return a non-empty list of possible options satisfying the constraints or the lease will not be accepted; it is thus important that the resource database supports efficient querying over an arbitrary set of key/value pairs (resource metadata).

Unless an iterative negotiation style interaction with the client [17] is desired and supported, a selection function is then applied to pick a specific option. Depending on the timing of resource assignment this function may optimize constraint management across leases or optimize administrative processes. For example, in our original implementation the selection function would pick the first item off the list; this led to significant churn on nodes that the resource query returned first and thus uneven hardware wear; we subsequently modified the selection function to pick a random resource which resulted in more uniform assignments across resources. In general, the selection function can be used to optimize other qualities like power usage. Once a unique resource is identified, records are persisted in the database and a resource reservation ID is returned.

Lease management may involve management for either adaptation or optimization. For example, the system may dynamically monitor resource inventory for its health status. Unhealthy resources are marked as such, and any leases that contain that resource (active or pending) enter a special "degraded" state. Based on policies and configuration, the lease manager may automatically try to fix the lease (both active and pending) by finding another resource that matches the original constraint stored in the database, by disassociating the resource from the lease. If no replacement resources are found, the lease remains in the degraded state.

The lifecycle events associated with a lease set up during lease creation are periodically checked and triggered at appropriate times. Most of the events delegate to resource plugins described in the next section to implement resource-specific functions. For example, once the reservation is ready to start, an event is triggered that causes the manager to call the internal `on_start` operation implemented by the enactment plug-in; as a result of this action the reservation status changes from pending to active.

C. Resource plugins

The Lease Manager handles only functionality related to managing resource reservations and assumes that enactment, i.e., a method for allowing reservation owners to access their reserved resources while their reservation is active, is implemented by resource-specific enactment services. The main assumption we make about those services is that they can separate reservable resources from a pool of (potentially) on-demand resources, making them usable only when obtained through the Lease Manager. For example, when including floating IPs as an allocatable resource via our system, the operators must ensure that reservable floating IPs are not included in their subnet's allocation pools, which prevents them from being allocated to users directly via Neutron – but then the Lease Manager, using privileged service credentials, can call out to Neutron to allocate floating IPs into a specific project (in this case the project owning the reservation) and remove them from the project when requested (i.e. when the reservation ends).

Enactment plugins allow the resource manager to support leases for different types of cloud resources, managed by different services (e.g., compute resources managed by OpenStack Nova [10] and network resources by Neutron [11]). To interface with these services, each resource type requires resource-specific enactment plugins, ensuring separation of concerns.

The create, update, and delete inventory management operations as shown in Figure 1 contain an almost direct pass through to their plugin implementation. They contain either custom-made tools for generating resource meta-data or interface with a service that holds that information (e.g., it might fetch compute host information from OpenStack Nova or services configuring it for Nova). Especially when adapting resource management services that were not originally implemented to work with reservation systems, this part of the plugin may also implement a method separating reservable resources from the main pool of on-demand resources (managed by a service like Nova), making them usable only when reserved. In an OpenStack installation this

would result in dividing the pool of nodes into reservable nodes and nodes available via on-demand only as before.

While creating and updating leases is handled entirely as a generic reservation, the allocatable resource manager plugins implement functions dealing with allocating and deallocating actual resources to a lease. Those operations are as follows:

```
on_start(resource_reservation_id)
before_end(resource_reservation_id)
on_end(resource_reservation_id)
update_reservation(resource_reservation_id, values)
```

The `on_start` and `on_end` functions are called respectively when a reservation (lease of a specific resource) starts and ends and handle resource allocation and deallocation. In addition, `on_end` is also called when a lease is deleted, to trigger the end of an active reservation or perform required cleanup for pending reservations. The `before_end` function can trigger an action at a configurable time before the end of a reservation. For example, it can be used to snapshot instances running on compute hosts before they are terminated at the end of their reservations.

While updating a pending reservation can be handled entirely via generic service logistics implementation, once a reservation becomes active (i.e., is associated with allocated resources) updating a reservation may trigger a call to a plugin `update_reservation` function (e.g. adding more compute nodes to an existing reservation).

IV. Implementation

In the context of the Chameleon project we defined three types of allocatable resources: heterogeneous bare metal machines, isolated network segments (VLANs) and public IP addresses on the Chameleon testbed [6]. The compute nodes are well-described by the Chameleon Resource Discovery [19], down to serial numbers of individual components. We chose to provide bare metal nodes as allocatable resources in order to provide both system and performance isolation; the sole ownership of the node ensures that users can run performance tests without interference by others. In contrast, the isolation property for network allocatable resources (VLANs) provides only system isolation; this is because we do not currently have a reliable implementation ensuring performance isolation for networks. The IP addresses are allocated from a pre-assigned pool. For all allocatable resources, Chameleon provides a resource calendar that facilitates planning.

While the implementation of individual allocatable resources varies, the ability to allocate, meter, and enforce usage is implemented via the same service. The Lease Manager is based on a separate OpenStack service called Blazar [9], to which we are actively contributing. We additionally integrated or implemented separate resource plugins for each use-case we required: bare metal node reservation (via OpenStack Nova, the compute instance provisioning service), VLAN 802.1Q tag reservation, and public IP reservation (via OpenStack Neutron, the network provisioning service).

A. Blazar: Allocatable Resource Manager

The Blazar system consists of two components: an API component, which provides the lease interfaces over an authenticated HTTP/JSON interface, and a manager component, which provides lease, reservation, and resource lifecycle management, as well as the delegation to various resource plugins for enactment. The API and manager components communicate over an RPC interface, where an AMQP bus serves as the transport layer. Authentication to the HTTP/JSON interfaces is performed via OpenStack's Keystone [20] authentication service. The interfaces are exposed to end-users over the Internet on a TLS-encrypted connection, which is terminated by a proxy running HAProxy.

The manager component handles user requests and translates them into actions against the backing resource and reservation databases. Blazar does not lazy-assign resources; when a user creates a lease, specific resources are selected and assigned to the lease, making them unreservable by other users for that time period.

We use the SQLAlchemy library [21] to create a thin object-relational mapping (ORM) layer that the manager uses to interact with database entities. Each resource type has three database tables associated with it: a *resources* table, which stores the resource records, a *reservations* table, which stores a set of constraints specific to a reservation for the resource and any parameters needed for enactment of the reservation, and an *allocations* table, which stores associations between the first two tables once resources are allocated to a reservation. An optional fourth table called *extra capabilities* can be used to store arbitrary key/value pairs that further describe a resource. Users can leverage these *extra capabilities*, combined with some attributes standard to the resource (and stored in the resources table), to filter the resource inventory via their reservation constraints. For the lease management, we have three tables: *leases*, which stores the lease records, *events*, which stores the lease lifecycle event records, and *reservations*, which serves as a general table for all reservations across all resource types. A record in a resource's reservations table is associated with a record in the general reservations table. This separation is necessary due to a specific enactment plugin sometimes needing additional parameters stored at lease creation time, e.g. which network to assign a public IP from.

The manager queries the events table every few seconds and triggers any unexecuted events whose time has come via plugins described below.

B. Nova/Ironic Plugin: Nodes as Resources

The Nova plugin implements reservation of bare metal nodes. In OpenStack, bare metal provisioning is a combined effort between the Nova and Ironic [22] systems. When an operator adds a bare metal node to the inventory, the operator does so by specifying an Ironic node UUID. The plugin retrieves specs such as how many CPUs are on the node from Nova. These attributes are then mirrored in the resource database. Operators can add additional metadata to the node, e.g. rack placement or CPU vendor information, which is stored in the *extra capabilities* table for this resource type.

At lease start, the plugin moves the reserved nodes to a special Nova host group. Users must present a valid reservation ID to Nova when launching an instance, and Nova schedules their instance on one of the nodes in this host group. Before the lease ends, the plugin will send a notification email to the email address tied to the user's OpenStack account. This is important because when a lease ends, the plugin will instruct Nova to terminate all running instances on the bare metal nodes, and users may want to ensure their data is moved off the node beforehand. The plugin cleans up the host group after instance termination. Any BIOS or firmware settings are reset as part of instance termination; this is performed by Ironic.

C. Neutron Plugin: VLANs as Resources

One of the networking enactment plugins Chameleon uses is the network segment plugin, which allows users to reserve a VLAN 801.2Q tag. The Chameleon testbed infrastructure resides on host institution networks both at TACC and at the University of Chicago, and initially relied upon switches provided by the host institution. For this reason, only a limited number of 801.2Q tags were provided to Chameleon, and demand for isolated networks could exceed capacity. Additionally, network slices are built by Chameleon users using special *stitchable VLANs* extending to the nearest stitchport [23], and they are few in number (e.g., only 10 are available at the University of Chicago site). To utilize the plugin, an operator adds networks to the resource inventory by specifying their 801.2Q tag. Additionally, operators configure Neutron to no longer allow users to create networks with a specific 801.2Q tag, as only the resource plugin should be allowed to perform this action.

During lease creation, the VLAN resource plugin will instruct Neutron to create a new OpenStack network with a given 801.2Q tag. The network is associated with the users account, and will appear in their dashboard for use, though they wont be able to modify it. When the lease ends, the network is simply deleted, making sure to first unhook the network from any running instances.

D. Neutron Plugin: IPs as Resources

The second networking enactment plugins is responsible for managing the IPv4 addresses allocated to Chameleon on the public Internet. Metering public IPs is important as, in our experience, users would often allocate more public IPs to their account than needed, or likewise forget to release them when finished. Over time, this can deplete the pool of available IP addresses. To utilize this plugin, an operator adds IP addresses to the resource inventory by specifying their IPv4 address and Neutron network UUID. Normally, Neutron provides an interface that allows users to request an IP on a given network out of an allocation pool in an on-demand fashion. To properly implement IPv4 addresses as an allocatable resource, this interface must be disabled, which can effectively be accomplished by configuring the Neutron network to have an empty on-demand IP allocation pool.

During lease creation, the IP resource plugin will instruct Neutron to allocate a new Floating IP on the network. The Floating IP is then associated with the user's account, and will appear in their dashboard for use. When the lease ends, the Floating IP is deleted after ensuring it is no longer assigned to any running instance. It is not currently possible to prevent a user from deleting this Floating IP, but in this event, the resource plugin simply does not attempt to delete the IP at lease termination. This enactment plugin was contributed by NTT.

V. Analysis of Lease Usage on Chameleon

To understand how users were using leases we analyzed the usage data from the Chameleon testbed between 2015-07-17 and 2019-04-11. The usage is broken down over all types of node resources on the Chameleon testbed described in detail at [24]. We gradually added resources to the Chameleon testbed (e.g., the Skylake nodes were added slightly more than a year ago) so for each resource the relevant usage is shown from the time it was added. We also removed all maintenance leases as well as all operations leases from the pool to focus exclusively on user behavior. The usage data was collected from OpenStack Blazar (reservation service) and Nova (compute service) databases, and all the DevOps data (data belongs to the internal development and maintenance projects) was excluded.

We first asked to what extent Chameleon users took advantage of the fact that testbed resources are allocatable rather than merely using resources that happened to be available when the user started the experiment. To assess that we counted the number of advance reservations used for each type of allocatable node resources on Chameleon (reservations for floating IP addresses and VLANs were introduced very recently and have not yet generated reliable usage information). We considered the lead time with which each reservation was made and mapped them into four categories: (1) on-demand, (2) up to a day in advance (reservations with short lead time), (3) up to a week in advance, and (4) more than a week in advance (reservations with long lead time). The results are shown in Figure 2

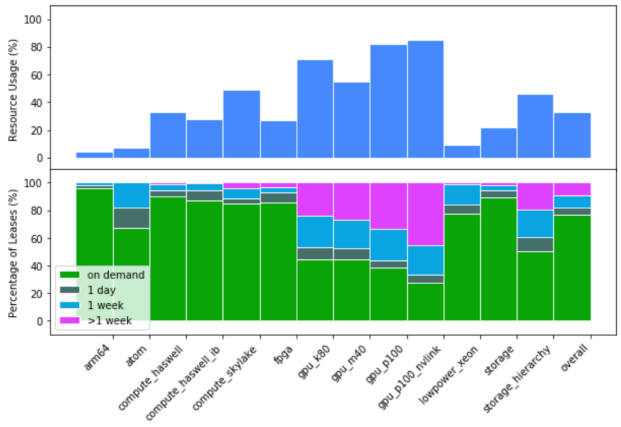

Figure 2. The relationship between reservation lead time and resource usage. Upper: percent usage of resources for various node types; Lower: percentage of leases that falls into four different reservation lead time categories for various node types.

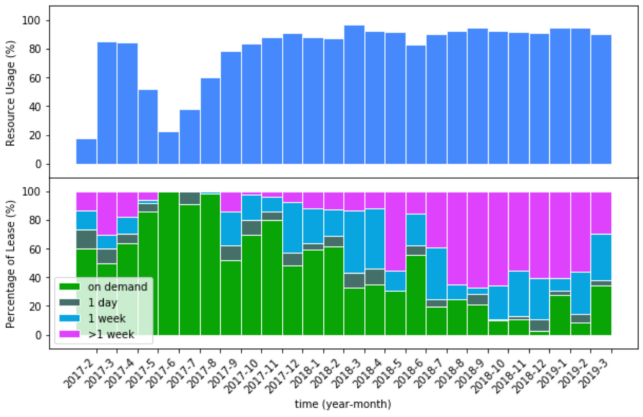

Figure 3. The relationship between reservation lead time and resource usage of GPU_P100 nodes. Upper: percent usage of GPU_P100 by month; Lower: percentage of leases for reservation lead time of GPU_P100.

(lower) where each category is shown as percentage of the total number of leases for that specific resource.

The results show that the most reservations and with the most lead time were made for the GPU resources, a resource type that is very much in demand, closely followed by storage hierarchy nodes, also a very desirable resource group. In terms of reservations with long lead time, the FPGA and Skylake nodes are also notable. We then overlaid the graph with a graph of resource usage Figure 2 (upper) which shows strong correlation between resource usage measured as a total time a resource has been leased divided by total time available across all resources in that type (times when resources were in maintenance were excluded). The superimposed graphs show a strong correlation between the number of advance reservations and resource popularity: clearly, as resources are harder to obtain users are increasingly more motivated to allocate resources in advance.

We then asked how much of an incentive to make a reservation is the need to obtain a multi-node lease. We asked what percentage of all advance reservations on Haswell and Skylake resources (i.e., larger clusters created specifically for multi-node experimentation) were made for multi-node leases. It turns out that on Skylakes 53.17% of advance reservations were made for leases of more than one node (in that 35.61% for reservations of more than 8 nodes) and on Haswells 40.78% were made for leases of more than one node (in that 20.80% for reservations of more than 8 nodes). This again reflects the difference in usage of the resource (there are more Haswell nodes and Skylakes are newer and thus more popular).

A similar relationship is illustrated in Figure 3 below which compares utilization of the GPU_P100 nodes over time: when the cluster got introduced in early 2017 and over the summer of that year the utilization was relatively low and users were almost always able to find resources on-demand; as the resource became better known however users had to increasingly use advance reservations with longer lead times; the recent reversal of this trend reflects new policies that were introduced recently to prevent individual users from creating excessive numbers of leases.

While the capability to make a resource lease supports the critical requirement for temporary resource ownership, in the absence of incentives (such as payment) for users to limit their usage it also creates the potential for misuse. While Chameleon has policies that limit this misuse – usage is charged against allocations, leases are limited to 7 days, and there are limits on how many leases a user can have open – the effectiveness of these policies in practice is a matter of interest. To evaluate it, we use a simple measure: since a user cannot do much with a lease unless s/he deploys an instance, we defined a lease in-use metric, which reflects the percentage of time a lease has an instance deployed on it, and expressed it as the ratio of time spent with instance deployed over the total time of reservation, both calculated across all the nodes in a lease.

Figure 4 shows the lease-in-use metric for all Chameleon leases. We can see that by this metric 38.86% of leases were fully used, 62.87% of leases were more than 80% used, but 18.65% of the leases were used less than 20% and a substantial 15.1% never had an instance deployed on them. While some discontinuity in lease usage is legitimate (due e.g., to changing configuration of resources), there are at least two examples of simply not sharing fairly: users make the reservation to “hold the resource in readiness” in which case the beginning of the lease is not used (we found that 5.8% of all leases were not used for at least a day) or they may forget to release resources that are no longer needed (we found that 5.7% of all leases were not used for at least

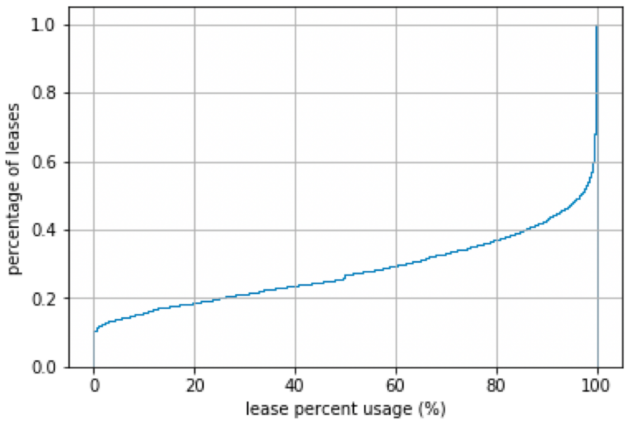

Figure 4. The cumulative distribution function (CDF) of lease percent usage

a day before they expired). On the other hand, we also found that 19.15% of all leases released resources early. Clearly, the ability to reserve resources has to come with incentives promoting reasonable sharing (as it does in e.g., commercial clouds when the incentives are monetary).

VI. Related Work

Various ways of allocating resources to provide isolation for experiments have been described in the context of various experimental testbeds [2–4] but to our knowledge this is the first work giving a systematic description of them as "units of allocation" and relating them to similar concepts enabling exclusive usage in infrastructure clouds.

Many approaches to advance reservations have been proposed, including our own work [17], [25]. Here, we consider advance reservations over multiple resource types and present the experiences of their use overtime. OpenPEX [26] as well as [27] describe negotiation protocols that allows users and providers to come to an agreement when the original request cannot be precisely satisfied; their focus is on negotiation and is orthogonal to our approach. Reservations of network resources are described by Charbonneau et al. discussed a number of architectures for supporting advance reservation of network resources, including OSCARS, DRAC, EnLIGHTened, G-Lambda, and PHOSPHORUS [28].

Finally, in terms of implementation the work on advance reservations in OpenStack has been originally initiated by [29] to investigate the impact of advanced reservations for energy-aware provisioning of bare-metal cloud resources. We build on this work, and extended it to fit the needs of an experimental testbed as well as generalize it to manage multiple types of resources, including networking.

VII. Conclusion

In the course of designing an experimental testbed for Computer Science research we developed the abstraction of allocatable resource, which allows clients to provision well-defined, isolated resources available between client-controlled time events. This paper describes the architecture of a service managing such allocatable resources. We implemented this architecture as the Blazar component within the widely used OpenStack system, originally to manage allocatable compute resources, then extending it to manage floating IPs and VLANs. The presented architecture is adaptable and can be extended further to manage container deployments, IoT devices, wireless networks, or concepts relating to other domains.

An analysis of three and a half years of usage of Blazar on the Chameleon testbed shows that advance reservations are an effective tool for allocating resources especially in the presence of resource scarcity. At the same time, we note that a close management of user incentives is needed to ensure that the managed resources are being made good use of and shared fairly within the community.

Acknowledgment

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. We would like to thank the OpenStack community whose work we leverage, in particular contributions from NTT and other organisations to the OpenStack Blazar project. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

References

[1] P. Mell, T. Grance *et al.*, "The NIST Definition of Cloud Computing," 2011.

[2] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab *et al.*, "Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed," *The International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[3] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A Federated Testbed for Innovative Network Experiments," *Computer Networks*, vol. 61, pp. 5–23, 2014.

[4] D. Johnson, T. Stack, R. Fish, D. M. Flickinger, L. Stoller, R. Ricci, and J. Lepreau, "Mobile Emulab: A Robotic Wireless and Sensor Network Testbed," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 2006, pp. 1–12.

[5] G. Von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Voeckler, R. J. Figueiredo, J. Fortes *et al.*, "Design of the Futuregrid Experiment Management Framework," in *2010 Gateway Computing Environments Workshop (GCE)*. IEEE, 2010, pp. 1–10.

[6] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Mambretti, P. Rad, and P. Ruth, "Chameleon: A Scalable Production Testbed for Computer Science Research," in *Contemporary High Performance Computing: From Petascale toward Exascale*, 1st ed., ser. Chapman Hall/CRC Computational Science, J. Vetter, Ed. Boca Raton, FL: CRC Press, 2018, vol. 3, ch. 5.

[7] Chameleon. [Online]. Available: https://www.chameleoncloud.org

[8] Openstack. [Online]. Available: https://www.openstack.org/

[9] Openstack Blazar. [Online]. Available: https://docs.openstack.org/blazar

[10] Openstack Nova. [Online]. Available: https://docs.openstack.org/nova

[11] Openstack Neutron. [Online]. Available: https://docs.openstack.org/neutron

[12] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, 2005.

[13] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*. James Turnbull, 2014.

[14] Docker. [Online]. Available: https://www.docker.com/

[15] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific Containers for Mobility of Compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[16] R. McGeer, M. Berman, C. Elliott, and R. Ricci, *The GENI book*. Springer, 2016.

[17] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web Services Agreement Specification (WS-Agreement)," in *Open grid forum*, vol. 128, no. 1, 2007, p. 216.

[18] F. Liu, K. Keahey, P. Riteau, and J. Weissman, "Dynamically Negotiating Capacity between On-demand and Batch Clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 38.

[19] Chameleon Resource Discovery. [Online]. Available: https://www.chameleoncloud.org/hardware/

[20] Openstack Keystone. [Online]. Available: https://docs.openstack.org/keystone

[21] SQLAlchemy. [Online]. Available: https://www.sqlalchemy.org/

[22] Openstack Ironic. [Online]. Available: https://docs.openstack.org/ironic

[23] I. Baldin, J. Chase, Y. Xin, A. Mandal, P. Ruth, C. Castillo, V. Orlikowski, C. Heermann, and J. Mills, "ExoGENI: A Multi-domain Infrastructure-as-a-Service Testbed," in *The GENI Book*. Springer, 2016, pp. 279–315.

[24] Chameleon Hardware Discovery. [Online]. Available: https://www.chameleoncloud.org/hardware/

[25] B. Sotomayor, K. Keahey, and I. Foster, "Combining Batch Execution and Leasing using Virtual Machines," in *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008, pp. 87–96.

[26] S. Venugopal, J. Broberg, and R. Buyya, "Openpex: An Open Provisioning and Execution System for Virtual Machines," in *17th International Conference on Advanced Computing and Communications (ADCOMŠ09). Citeseer*, 2009.

[27] J. Chung, R. Kettimuthu, N. Pho, R. Clark, and H. Owen, "Orchestrating Intercontinental Advance Reservations with Software-defined Exchanges," *Future Generation Computer Systems*, vol. 95, pp. 534–547, 2019.

[28] N. Charbonneau, V. M. Vokkarane, C. Guok, and I. Monga, "Advance Reservation Frameworks in Hybrid IP-WDM Networks," *IEEE Communications Magazine*, vol. 49, no. 5, pp. 132–139, 2011.

[29] M. D. de Assunçõo, L. Lefevre, and F. Rossigneux, "On the Impact of Advance Reservations for Energy-aware Provisioning of Bare-metal Cloud Resources," in *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE, 2016, pp. 238–242.