A Retrospective on ORCA: Open Resource Control Architecture [1]

Jeff Chase and Ilya Baldin

Technical Report CS-2014-XXX
Department of Computer Science
Duke University
December 2014

Abstract

ORCA is an extensible platform for building infrastructure servers based on a foundational leasing abstraction. These servers include Aggregate Managers for diverse resource providers and stateful controllers for dynamic slices. ORCA also defines a brokering architecture and control framework to link these servers together into a federated multi-domain deployment. This chapter reviews the architectural principles of ORCA and outlines how they enabled and influenced the design of the ExoGENI Racks deployment, which is built on the ORCA platform. It also sets ORCA in context with the GENI architecture as it has evolved.

1 Introduction

The Open Resource Control Architecture (ORCA) is a development platform and control framework for federated infrastructure services. ORCA has been used to build elements of GENI, most notably the ExoGENI deployment [3]. In ExoGENI the ORCA software mediates between GENI user tools and the various infrastructure services (IaaS) that run a collection of OpenFlow-enabled cloud sites with dynamic layer-2 (L2) circuit connectivity.

ORCA is based on the SHARP resource peering architecture [10], which was conceived in 2002 for federation in PlanetLab and related systems [4] as they emerged. ORCA incorporates the Shirako resource leasing toolkit [15] and its plug-in extension APIs. This research was driven by a vision similar to GENI: a network of federated resource providers enabling users and experimenters to build custom *slices* that combine diverse resources for computing, networking, and storage.

When construction of GENI began in 2008, ORCA was selected as a candidate control framework along with three established network testbeds: PlanetLab, Emulab, and ORBIT. ORCA was the only candidate framework that had been conceived and designed from the ground up as a platform for secure federation, rather than to support a centrally operated testbed. At that time ORCA was research software with no production deployment, and so it was more speculative than the other control framework candidates. We had used it for early experiments with elastic cloud computing [15, 17, 20, 5, 16], but we were just beginning to apply it to network resources [6, 1].

The GENI project was therefore an opportunity to test whether we had gotten ORCA's architecture and abstractions right, by using it to build and deploy a multi-domain networked IaaS system. The crucial test was to show that ORCA could support advanced network services, beginning with RENCI's multi-layer network testbed (the Breakable Experimental Network—BEN) in the Research Triangle region.

During the GENI development phase (2008-2012), participants in the GENI *Cluster D* group, led by RENCI, built software to control various infrastructures and link them into a federated system under the ORCA control framework. For example, the Kansei group built KanseiGenie [18], an ORCA-enabled wireless testbed. The RENCI team built various software elements later used in ExoGENI. They include a control system and circuit API for the BEN network; modules to link ORCA with off-the-shelf cloud managers; storage

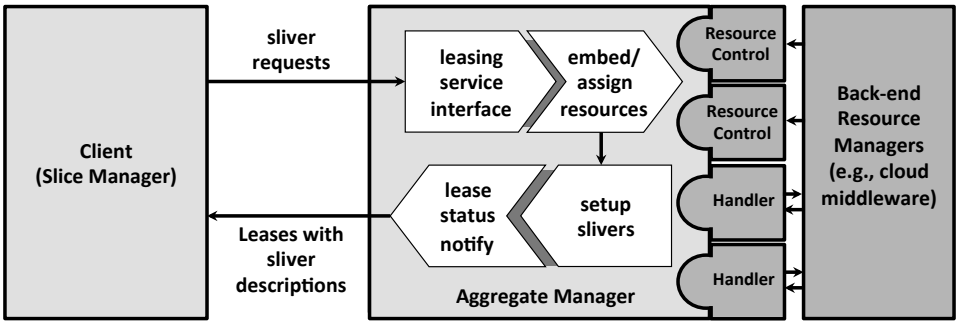

Figure 1: Structure of an ORCA aggregate. An Aggregate Manager (AM) issues sliver leases (§2.1, §2.3) to clients called Slice Managers (SMs). The AMs are built from a generic leasing server core in the ORCA platform (light shade). The core invokes plug-in extension modules (dark shade) for resource-specific functions of each aggregate (§2.2). These extensions may invoke standard APIs of off-the-shelf IaaS resource managers, e.g., OpenStack. Their resource control functions are driven by logical descriptions of the managed resources (§2.4). ExoGENI uses this structure for diverse aggregates, including network providers (§2.5).

provisioning using commercial network storage appliances; VLAN-sliced access control for OpenFlow-enabled network dataplanes; adapters for various third-party services; and a front-end control interface that tracked the GENI API standards as they emerged.

Through this phase ORCA served as a common framework to organize these efforts and link them together. This was possible because ORCA was designed as an *orchestration platform* for diverse resource managers at the back end and customizable access methods at the front end, rather than as a standalone testbed itself.

ORCA was based on the premise that much of the code for controlling resources in a system like GENI would be independent of the specific resources, control policies, and access methods. The first step was to write the common code once as a generic *toolkit*, keeping it free of assumptions about the specific resources and policies. The second step was to plug in software adapters to connect the toolkit to separately developed IaaS resource managers, which were advancing rapidly outside of the GENI effort. Finally, we used a powerful logic-based *resource description language* (NDL-OWL) to represent resources and configurations declaratively [2, 1]. Descriptions in the language drive policies and algorithms to co-schedule compute and network resources and interconnect them according to their properties and dependencies.

This approach enabled us to demonstrate key objectives of GENI—automated embedding and end-to-end assembly (*stitching*) of L2 virtual network topologies spanning multiple providers—by early 2010, well before GENI had defined protocols to enable these functions. ORCA already defined a protocol and federation structure similar to what was ultimately adopted in GENI; we essentially just used that structure to link together third-party back-end resource managers as they appeared, and control them through their existing APIs. This philosophy carried through to the ExoGENI effort when it was funded in 2012: the *exo* prefix refers in part to the idea of incorporating resources and software from outside of GENI and exposing their power through GENI APIs.

The remainder of this chapter outlines the ORCA system in more detail, illustrating with examples from ExoGENI. §2 gives an overview of ORCA's abstractions and extension mechanisms, and the role of logic-based resource descriptions. §3 summarizes ORCA's structure for orchestrating providers based on *broker* and *controller* services. §4 sets ORCA in context with the GENI architecture as it has evolved.

2 Overview of the ORCA Platform

ORCA and GENI embody the key concepts of slices, slivers, and aggregates derived from their common heritage in PlanetLab. An *aggregate* is a resource provider: to a client, it appears as a hosting site or

domain that can allocate and provision resources such as machines, networks, and storage volumes. An Aggregate Manager (AM) is a service that implements the aggregate's resource provider API. A *sliver* is any virtual resource instance that is provisioned from a single AM and is named and managed independently of other slivers. Slivers have a lifecycle and operational states, which a requester may query or transition (e.g., **shutdown**, **restart**). A *slice* is a logical container for a set of slivers that are used for some common purpose. Each sliver is a member of exactly one slice, which exists at the time the sliver is created and never changes through the life of the sliver.

Client tools call the AMs to allocate and control slivers across multiple aggregates, and link them to form end-to-end environments (slices) for experiments or applications. In ORCA we refer to a client of the AM interface as a Slice Manager (SM, see §3). Each request from an SM to an AM operates on one or more slivers of exactly one slice. The slices are built to order with suitable end-to-end connectivity according to the needs of each experiment. The slice abstraction serves as a basis for organizing user activity: loosely, a slice is a unit of activity that can be enabled, disabled, authorized, accounted, and/or contained.

Figure 1 illustrates an ORCA aggregate and the elements involved in issuing a sliver lease to a client SM. ORCA bases all resource management on the abstraction of *resource leases* (§2.1). The core leasing engine is generic: ORCA factors support for specific resources and policies into replaceable *extension modules* that plug into the core (§2.2). The extensions produce and/or consume declarative *descriptions* that represent information needed to manage the resources (§2.4). The common leasing abstractions and plug-in APIs facilitate implementation of AMs for diverse resources (§2.5).

2.1 Resource Leases

Resource leases are explicit representations of resource commitments granted or obtained through time. The lease abstraction is a powerful basis for negotiating and arbitrating control over shared networked resources. GENI ultimately adopted an equivalent leasing model in the version 3.0 API in 2012.

A lease is a promise from an aggregate to provide one or more slivers for a slice over an interval of time (the *term*). Each sliver is in the scope of exactly one lease. A lease is materalized as a machine-readable document specifying the slice, sliver(s), and term. Each lease references a logical description (§2.4) of the slivers and the nature of the access that is promised. Leases are authenticated: ORCA leases are signed by the issuing AM. Lease contracts may be renewed (*extended*) or *vacated* prior to expiration, by mutual agreement of the slice owner and AM. If an SM abandons a sliver, e.g., due to a failure, then the resources are freed when the lease expires.

SHARP introduced a two-step leasing API in which the client first obtains an approval to allocate the resources (a *ticket*), and then redeems the ticket to claim the resources and provision (instantiate) the slivers. A ticket is a weaker promise than a lease: it specifies the promised resources abstractly. The AM assigns (binds) concrete resources to fill the ticket only when it is redeemed. In ORCA the tickets may be issued by *brokers* outside of the aggregates (§3.2).

By *separating allocation and provisioning* in this way, the leasing API enables a client to obtain promises for resources at multiple AMs cheaply, and then move to the redeem step only if it succeeds in collecting a resource bundle (a set of tickets) matching its needs. The two-step API is a building block for grouped leases and *atomic co-allocation*—the ability to request a set of slivers such that either the entire request succeeds or it aborts and no sliver is provisioned. The AM may commit resources cheaply in advance, and then consider current conditions in determining how to provision the resources if and when they are needed.

From the perspective of the AMs, leases provide a means to control the terms under which their resources are used. The resource promises may take a number of forms expressible in the logic, ranging along a continuum of assurances ranging from a hard physical (e.g., bare metal) reservation to a weak promise of best-effort service over the term. By placing a time bound on the commitments, leases enable AMs to make other promises for the same resources in the future (*advance reservations*).

From the perspective of the SMs, leases make all resource allotments explicit and visible, enabling them to

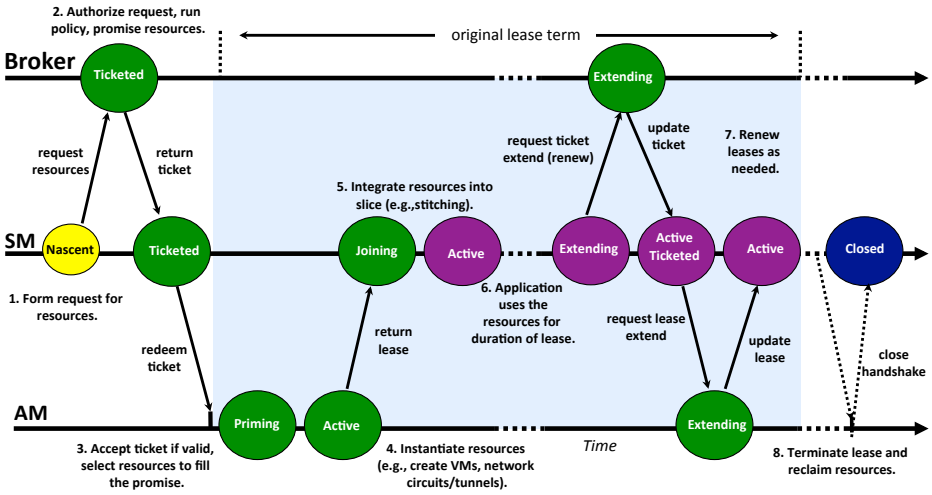

Figure 2: Lease states and transitions. Interacting state machines representing a single ORCA lease at three servers: the SM that requested the resource, the broker that issued the ticket, and the provider (AM) that sets up the slivers, issues the lease, and tears down the resource when the lease expires. Each state machine passes through various intermediate states, triggering policy actions, local provisioning actions, and application launch. This figure is adapted from [15].

reason about their assurances and the expected performance of the slice. Since the SM may lease slivers independently, it can modify the slice by adding new slivers and/or releasing old slivers, enabling *elastic* slices that grow or shrink according to need and/or resource availability. Various research uses of the ORCA software experimented with elastic slice controllers (§3.3), building on our early work in adaptive resource management for hosting centers [8, 9].

2.2 Extension Modules

ORCA is based on a generic reusable leasing engine with dependencies factored into stackable plug-in extension modules [15]. The core engine tracks lease objects through time in calendar structures that are accessible to the extensions. For example, an AM combines the leasing engine with two types of extensions that are specific to the resources in the aggregate:

- `ResourceControl`. The AM core upcalls a `ResourceControl` policy module periodically with batches of recently received sliver requests. Its purpose is to `assign` the requests onto the available resources. The batching interval is a configurable parameter. It may defer or reject each request, or approve it with an optional binding to a resource set selected from a pool of matching resources. The module may query the attributes of the requester, the state of the resources and calendar, other pending requests, and the history of the request stream.
- `Handler`. The AM core upcalls a `Handler` module to `setup` a sliver after approval, or `teardown` a sliver after a lease is closed (expired, cancelled, or vacated). Resource handlers perform any configuration actions needed to implement slivers on the back-end infrastructure. The handler API includes a `probe` method to poll the current status of a sliver, and a `modify` method to adjust its properties.

An ORCA AM may serve multiple types of slivers by combining multiple instances of these modules, which are indexed and selectable by sliver type. Each upcall runs on its own thread and is permitted to block, e.g., for configuration actions in the handler, which may take seconds or minutes to complete. The extensions post their results asynchronously through lease objects that are shared with the leasing core.

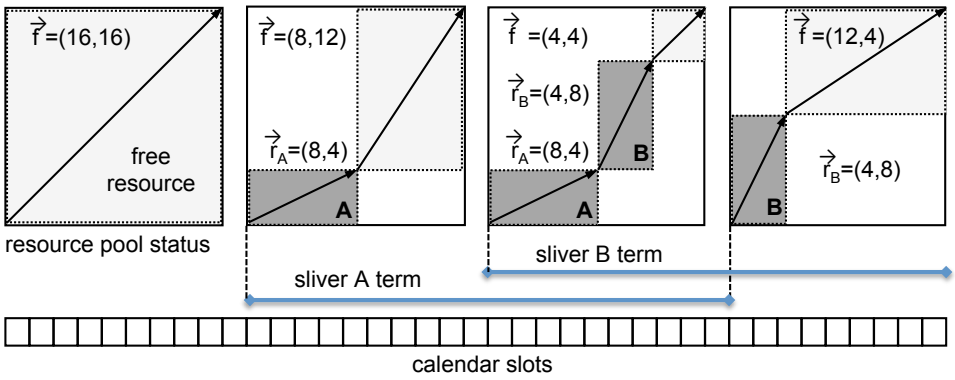

Figure 3: A simple example of resource algebra and sliver allocation. These slivers represent a virtual resource with two dimensions, e.g., virtual machines with specified quantities of memory and CPU power. They are allocated and released from a pool of bounded capacity. Free capacity in the pool is given by vector addition and subtraction as slivers are allocated and released.

2.3 Leasing Engine

The ORCA platform views each lease as a set of interacting state machines on the servers that are aware of it. The lease state machines have well-defined states and state transition rules specific to each type of server. Figure 2 illustrates typical states, transitions, and actions.

The core engine within each server serializes state machine transitions and commits them to stable storage. After a transition commits, it may trigger asynchronous actions including notifications to other servers, upcalls to extension modules, and various other maintenance activities.

Lease state transitions and their actions are driven by the passage of time (e.g., sliver `setup` at the start of the term and `teardown` at the end of the term), changes in status of the underlying resources (e.g., failure), decisions by policy modules, and various API calls.

Cross-server interactions in the leasing system are asyncronous and resilient. After a failure or server restart, the core recovers lease objects and restarts pending actions. The extensions may store data blobs and property lists on the lease objects. The core upcalls each extension with the recovered lease objects before restarting any actions. The servers and extensions are responsible for suppressing any effects from duplicate actions that completed before a failure but were restarted or reissued on recovery.

2.4 Resource Descriptions

The ORCA platform makes it possible to build new AMs quickly by implementing the `Handler` and `ResourceControl` modules. Since the leasing core and protocols are generic, there must be some means to represent resource-specific information needed by these modules. This is achieved with a data-centric API in which simple API requests and responses (`ticket/redeem/renew/close`) have attached *descriptions* that carry this content. The descriptions contain statements in a declarative language that describe the resources and their attributes and relationships.

The description language must be sufficiently powerful to describe the *resource service* that the aggregate provides: what kinds of slivers, sizes and other options, constraints on the capacity of its resource pools, and interconnection capabilities for slivers from those pools. It must also be able to describe resources at multiple levels of abstraction and detail. In particular, clients describe their *sliver requests* abstractly, while the aggregate's descriptions of the *provisioned slivers* are more concrete and detailed. GENI refers to these cases as *advertisement*, *request*, and *manifest* respectively.

In ORCA the descriptions are processed only by the resource-specific parts of the code, i.e., by the extension modules. The core ignores the descriptions and is agnostic to their language. An ORCA resource description is a set of arbitrary strings, each indexed by a key: it is a property list. ORCA defines standard labels for distinct property lists exchanged in the protocols, corresponding to the advertisement, request, and manifest cases.

To support meaningful resource controls, the description language must enable a *resource algebra* of operators to split and merge sliver sets and resource pools. Given descriptions of a resource pool (an advertisement) and a set of slivers, a resource algebra can determine if it is feasible to draw the sliver set from the pool, and if so to generate a new description of the resources remaining in the pool. Another operator determines the effect of releasing slivers back to a pool.

The original SHARP/Shirako lease manager [15] used in ORCA described pools as quantities of interchangeable slivers of a given type. A later version added resource controls using an algebra for multi-dimensional slivers expressed as vectors [12], e.g., virtual machines with rated CPU power, memory size and storage capacity, and IOPS. Figure 3 depicts a simple example of resource algebra with vectors.

In the GENI project we addressed the challenge of how to represent complex network topologies and sliver sets that form virtual networks over those topologies. For this purpose we adopted a *declarative logic language* for resource descriptions. Logical descriptions expose useful resource properties and relationships for inference using a generic reasoning engine according to declarative rules. In addition to their expressive power, logical descriptions have the benefit that it is semantically sound to split and combine them, because they are sets of independent statements in which all objects have names that are globally unique and stable. For example, a logical slice description is simply a concatenation of individual sliver descriptions, each of which can be processed independently of the others. Statements may reference objects outside of the description, e.g., to represent relationships among objects.

To this end, the RENCI team augmented the Network Description Language [13] with a description logic ontology in the OWL semantic web language. We called the resulting language NDL-OWL [2]. We used NDL-OWL to describe infrastructures orchestrated with ORCA: BEN and other networks and their attached edge resources, including virtual machine services. For example, NDL-OWL enables us to enforce semantic constraints on resource requests, express path selection and topology embedding as SPARQL queries over NDL-OWL graph models, check compatibility of candidate interconnections (e.g., for end-to-end VLAN tag stitching, §3.4), and generate sequences of handler actions to instantiate slivers automatically from the descriptions. These capabilities are implemented in extension modules with no changes to the ORCA core.

2.5 Building Aggregates with ORCA

We used ORCA and NDL-OWL to build a collection of Aggregate Managers (AMs) for back-end resource managers from other parties. In ExoGENI, these include two off-the-shelf managers for cloud sites: OpenStack for Linux/KVM virtual machines and xCAT for bare-metal provisioning. These systems expose local APIs to allocate and instantiate resources. Each AM runs one or more `Handler` modules that invoke these back-end control APIs.

We augmented the cloud aggregates with additional back-end software to function as sites in a *networked IaaS federation* under a separate NSF SDCI project beginning in 2010. The added software includes a caching proxy for VM images retrieved by URL, an OpenFlow access control proxy (FlowVisor) to enable slices to control their virtual networks, and OpenStack extensions for dynamic attachment of VM instances to external L2 circuits. We also added handlers to invoke storage provisioning APIs of third-party storage appliances. These elements are independent of the cloud manager: the AM handlers orchestrate their operation.

In the GENI project, we implemented AMs for network management. Most notably, the control software for BEN and its circuit service were implemented natively using ORCA in 2009; the AM handlers issue direct commands to the vendor-defined APIs on the BEN network elements.

Later we implemented AMs to provide *dynamic circuit service* for a network of cloud sites under ORCA

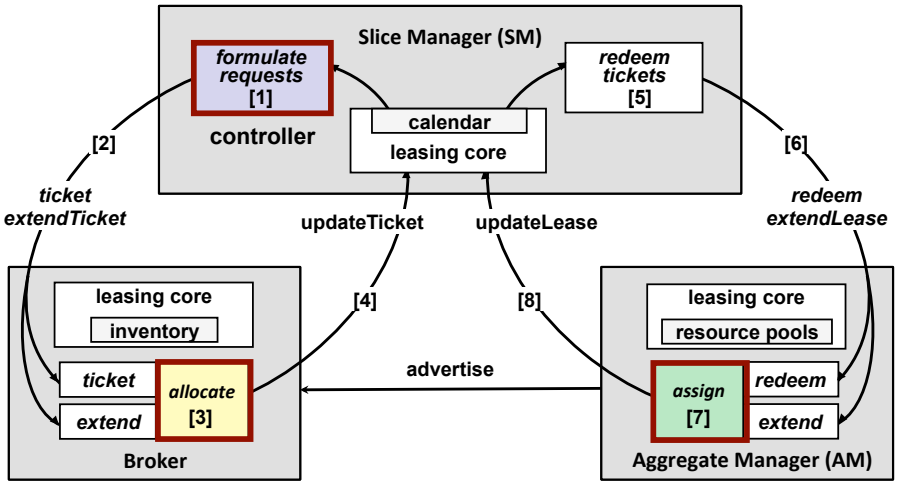

Figure 4: Interacting servers in the ORCA resource control plane. The figure shows the sequence of steps and protocol messages to provision a sliver, passing through policy modules in each server. This figure is adapted from [12].

control. The circuit AMs proxy third-party L2 circuit services from national-footprint backbone providers, including NLR's now-defunct Sherpa service and the OSCARS services offered by ESNet and Internet2. For these systems the AM handler calls the circuit API under its own identity; the circuit provider does not know the identities of the GENI users on their networks. In effect, the provider implicitly delegates to the AM the responsibility to authorize user access, maintain user contacts and billing if applicable, and provide a kill switch. This approach was easy to implement without changing the circuit providers: to the provider the AM is indistinguishable from any other client.

Finally, for ExoGENI we implemented an AM to control *exchange points*—RENCI-owned switches installed at peering centers where multiple transit providers come together (e.g., Starlight). These switches implement VLAN tag translation; the exchange AM uses this capability to stitch circuits from different providers into a logical end-to-end circuit. The handler for the exchange AM issues direct commands to the vendor API on the switches, similarly to the BEN control software.

3 Orchestration and Cross-Aggregate Resource Control

Section 2 described how we can build diverse aggregates with ORCA by plugging resource-specific and policy-specific extension modules into a common leasing core, and accessing their resources via common leasing protocols. This approach can apply to any aggregate whose resources are logically describable. It helps to deliver on a key goal of GENI: support for diverse virtual resources from multiple providers (aggregates).

But GENI's vision of a provider federation goes beyond that: it is also necessary to *coordinate* functions across aggregates in the federation. ORCA defines two kinds of coordinating servers that are not present in the GENI architecture: brokers and controllers. The *brokers* (§3.2) issue tickets for slivers on aggregates: they facilitate resource discovery and cross-aggregate resource control. The *controllers* (§3.3) manage slices: each slice is bound to exactly one controller, which receives notifications for all events in the slice and issues sliver operations on the slice. In general, ORCA controllers run on behalf of users to manage their slices: they have no special privilege.

These servers play an important role in ExoGENI. In particular, the ExoGENI controllers manage topology embedding and slice assembly (§3.4). They also implement the GENI API and proxy requests from GENI

users for ExoGENI resources (§3.5). Proxying requires them to check authorization for GENI users, so the ExoGENI AMs are configured to trust these special controllers to perform this function.

3.1 ORCA Resource Control Plane

Brokers and controllers are built using the same leasing platform as the AMs. The ORCA toolkit design recognized that these servers have key structural elements in common: a dataset of slices and slivers; timer-driven actions on lease objects organized in a calendar; similar threading and communication models; and lease state machines with a common structure for plug-in extension modules.

Figure 4 illustrates the three types of interacting servers in the ORCA framework and their policy modules. The AMs advertise their resource pools to one or more brokers. Controllers run as extension modules within generic Slice Manager (SM) servers. The SM controllers select and sequence resource requests for the slice to the brokers and AMs. The brokers run policy modules—similar to the AM `ResourceControl` extensions—to allocate slivers and issue tickets against an inventory of advertisements. The AMs provision slivers according to the tickets and other directives passed with each request. The protocol messages carry resource descriptions produced and consumed by the policy modules. All messages are signed to ensure a verifiable delegation path.

An ORCA deployment may combine many instances of each kind of server. ORCA was conceived as a *resource control plane* in which multiple instances of these servers interact in a deployment that evolves over time. The ORCA toolkit combines a platform to build these servers and a control framework to link them together. These linkages (e.g. delegations from AMs to brokers) are driven by configuration files or administrative commands, or programmatically from an extension module.

Since all agreements are explicit about their terms, this structure creates a foundation for resource management based on peering, bartering, or economic exchange [10, 14, 11]. An AM may advertise its resources to multiple brokers, and a broker may issue tickets for multiple AMs. The AM ultimately chooses whether or not to honor any ticket issued by a broker, and it may hold a broker accountable for any oversubscription of an AM's advertisements [10].

3.2 Brokers

Brokers address the need for cross-aggregate resource control. They can arbitrate or schedule access to resources by multiple users across multiple aggregates, or impose limits on user resource usage across aggregates using the same broker. The broker policy has access to the attributes of the requester and target slice, and it may maintain a history of requests to track resource usage by each entity through time.

Brokers also offer basic support for co-allocation across multiple aggregates, including advance reservations. This support is a key motivation for the two-step ticket/lease protocol in SHARP and ORCA. In particular, an ORCA broker may receive resource delegations from multiple aggregates and issue co-allocated tickets against them. Because all allocation state is kept local to the broker, the co-allocation protocol commits locally at the broker in a single step.

The key elements that enable brokers are the separation of allocation and provisioning (§2.1) and the "resource algebra" of declarative resource descriptions (§2.4). Given the algebra, the processing for ticket allocation can migrate outside of the AMs to generic brokers operating from the AM's logical description. The AM must trust the broker to perform this function on its delegated resource pools, but it can always validate the decisions of its brokers because the tickets they issue are redeemed back to the AM for the provisioning step.

The logical resource descriptions enable advertisements at multiple levels of abstraction. In practice, most AMs advertise *service descriptions* rather than their internal structure. For example, ExoGENI network AMs hide their topology details: they advertise only their edge interconnection points with other domains and the bandwidth available at each point. Abstract advertisements offer more flexibility and autonomy to the AMs,

who may rearrange their infrastructures or adjust sliver bindings to respond to demands and local conditions at the time of service.

Each AM chooses how to advertise its resources in order to balance the risks of ticket rejection or underutilization of its resources. For example, an unfortunate side effect of abstract advertisements is that brokers may issue ticket sets that are not feasible on the actual infrastructure, particularly during periods of high utilization. Ticket rejection by the AM is undesirable and disruptive, but it is unavoidable in the general case, e.g., during outages. An AM may hold unadvertised capacity in reserve to mask problems, but this choice leaves some of its resources unused. Alternatively, an AM may advertise redundantly to multiple brokers. This choice reduces the risk of wasting resources due to broker failure at the cost of a higher risk of overcommitment and ticket rejection.

To summarize, the flexible delegation model enables a continuum of deployment choices that balance the local autonomy of resource providers with the need for global coordination through the brokering system. Resource contracts and logical descriptions enable AMs to delegate varying degrees of control over their resources to brokers that implement policies of a community. At one end of the continuum, each AM retains all control for itself, effectively acting as its own broker, and consumers negotiate resource access with each provider separately. At the other end of the continuum a set of AMs federate using common policies implemented in a central brokering service. These are deployment choices, not architectural choices. In ExoGENI all AMs advertise to a central broker, but each cloud site also serves some resources locally.

3.3 Controllers

The slice controllers in ORCA match the Software-Defined Networking and Infrastructure (SDN/SDI) paradigm that is popular today. Like SDN controllers they control the structure of a virtual network (a slice) spanning a set of low-level infrastructure elements. They issue commands to define and evolve the slice, and receive and respond to asynchronous notifications of events affecting the slice. Like other ORCA servers, the controller/SM is stateful: it maintains a database recording the status of each slice, including active tickets and leases and any pending requests. It is the only control element with a global view of the slice.

One simple function of the controller is to automate sliver renewal ("meter feeding") as leases expire, to avoid burdening a user. The controller may allow leases to lapse or inject new lease requests acccording to a policy. In the early work, the controller and SM were conceived as the locus of automated adaptation policy for elastic slices and elastic services running within those slices [15, 17, 14, 20, 5, 16]. (This is why the SM was called *Service Manager* in the early papers.) For example, a 2006 paper [17] describes the deployment of elastic grid instances over a network of virtual machine providers, orchestrated by a "grid resource oversight controller" (GROC). The grid instances grow and shrink according to their observed load.

To assist the controller in orchestrating complex slices, the ORCA leasing engine can enforce a specified sequencing of lease setup actions issued from the SM. The controller registers dependencies by constructing a DAG across the lease objects, and the core issues the lease actions in a partial order according to the DAG. This structure was developed for controllers that orchestrate complex hosted services [15], such as the GROC, but it has also proved useful to automate stitching of network connectivity within slices that span aggregates linked by L2 circuit networks [2], as described in §3.4 below.

In particular, the leasing engine redeems and instantiates each lease before any of its successors in the DAG. Suppose an object has been ticketed but no lease has been received for a redeem predecessor: then the engine transitions the ticketed object into a **blocked** state, and does not fire the redeem action until the predecessor lease arrives, indicating that its setup is complete. The core upcalls the controller before transitioning out of the **blocked** state. This upcall API allows the controller to manipulate properties on the object before the action fires. For example, the controller might propagate attributes of the predecessor (such as an IP address or VLAN tag returned in the lease) to the successor, for use as an argument to a configuration action.

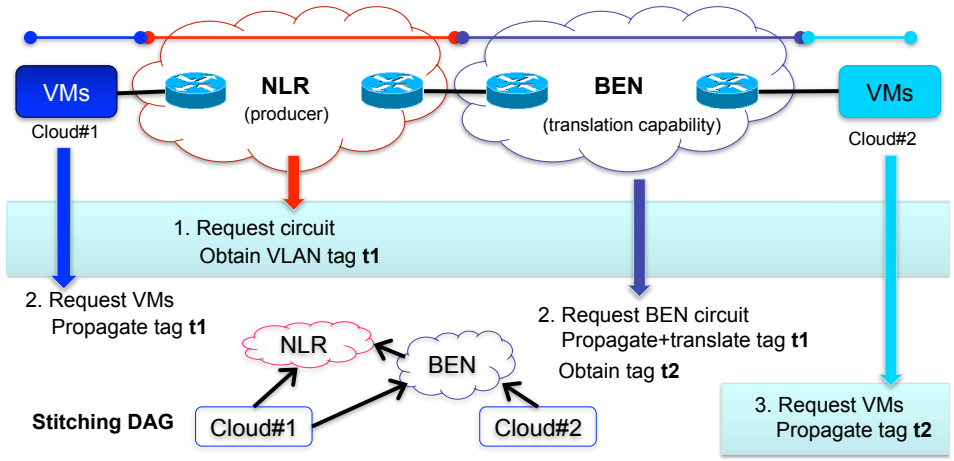

Figure 5: Dependency DAG and stitching workflow for an end-to-end L2 circuit scenario. NLR/Sherpa chooses the VLAN tag at both ends of its circuits, BEN has tag translation capability, and the edge cloud sites can accept tags chosen by the adjacent provider. The connection point descriptions yield a stitching workflow DAG. The controller traverses the DAG to assemble the circuit with a partial order of steps.

3.4 Automated Stitching and Topology Mapping

ExoGENI illustrates how controllers and their dependency DAGs are useful to plan and orchestrate complex slices. In particular, the controllers automate end-to-end circuit stitching by building a dependency DAG based on logical resource descriptions.

The controller first obtains the description for each edge connection point between domains traversed by links in a slice. The descriptions specify the properties of each connection point. In particular, they describe whether each domain can produce and/or consume "label" values (e.g., VLAN tags) that name an attachment of a virtual link to an interface at the connection point. A domain with translation capability can act as either a label producer or consumer as needed.

Stitching a slice involves making decisions about which domains will produce and which will consume labels, a process that is constrained by the topology of the slice and the capabilities of the domains. Based on this information, the controller generates a *stitching workflow DAG* that encodes the flow of labels and dependencies among the slivers that it requests from these domains. A producer must produce before a consumer can consume. The controller traverses the DAG, instantiating resources and propagating labels to their successors as the labels become available.

Figure 5 illustrates with a hypothetical scenario. The NLR circuit service is a producer: its circuits are compatible for stitching only to adjacent domains with consumer or tag translation capability. The resulting DAG instantiates the NLR circuit first, and obtains the tag from the sliver manifest returned by the domain's AM. Once the tag is known, the controller propagates it by firing an action on the successor slivers at the attachment point, passing the tag as a parameter. Each domain signs any labels that it produces, so that downstream AMs can verify their authenticity. A common broker or other federation authority may function as a trust anchor. In extreme cases in which VLAN tag negotation is required, e.g. among adjacent "producer" domains, it is possible to configure the broker policy module to allocate a tag from a common pool of values.

The ExoGENI controllers also handle inter-domain topology mapping (embedding) for complex slices [19]. A controller breaks a requested topology down into a set of paths, and implements a shortest-feasible-path algorithm to plan each path, considering compatibility of adjacent providers in the candidate paths as described above. To plan topologies, the controller uses a `query` interface on the brokers to obtain and cache the complete advertisements of the candidate network domains. It then performs path computation against these logical models in order to plan its sliver requests. If a path traverses multiple adjacent producer domains,

it may be necessary to bridge them by routing the path through an exchange point that can translate the tags. After the controller determines the inter-domain path, the domains select their own intra-domain paths internally at sliver instantiation time.

Topology embedding is expensive, so it is convenient to perform it in the SM controllers. The SMs and controllers are easy to scale because they act on behalf of slices: as the number of slices grows it is easy to add more SMs and assign new slices to the new SMs.

3.5 GENI Proxy Controller

ExoGENI runs special GENI controllers that offer standard GENI interfaces to GENI users. The GENI controllers run a converter to translate the GENI request specification (RSpec) into NDL-OWL. The converter also checks the request for compliance with a set of semantic constraints, which are specified declaratively in NDL-OWL. If a request is valid, it acts as a proxy to orchestrate fulfillment of the request by issuing ORCA operations to ExoGENI brokers and AMs. This approach enables suitably authorized GENI users to access ExoGENI resources and link them into their GENI slices.

ExoGENI's proxy structure was designed to support GENI standards easily as they emerged, without losing any significant capability. In particular, a global GENI controller exposes the entire ExoGENI federation as a single GENI aggregate. This approach enables GENI users to create complete virtual topologies spanning multiple ExoGENI aggregates, without relying on GENI stitching standards.

4 Reflections on GENI and ORCA

This section offers some thoughts and opinions on the GENI-ORCA experience. We believe that the ORCA architecture has held up well through the GENI process. We built and deployed ExoGENI as a set of extension modules with few changes to the ORCA core. Although the ORCA software itself is not used outside of ExoGENI, the rest of GENI has ultimately adopted similar solutions in all areas of overlap.

In particular, the latest GENI API standard is similar to the ORCA protocol, with per-sliver leases, separate `allocate` and `provision` steps, dynamic stitching, abstract aggregates with no exposed components, elastic slices with adjustable sliver sets, and a decoupled authorization system. Beyond these commonalities, GENI omits orchestration features from ORCA that could help meet goals that are still incomplete. It also adopts policies for user identity and authorization, which are outside the scope of the ORCA architecture.

The remaining differences lie in the data representations carried with the protocol—the languages for resource descriptions and for the credentials that support a request. In particular, GENI uses a resource description language (RSpec) that is not logic-based. RSpec may prove to be more human-friendly than NDL-OWL, but it is decidedly less powerful, and it rests on weaker foundations.

These differences are primarily interoperability issues rather than architectural issues or restrictions of the protocol itself. The version 3.0 GENI API is open to alternative credential formats including (potentially) broker-issued tickets, by mutual agreement of the client and server. In principle the protocol is open to alternative resource description languages as well.

4.1 Platforms vs. Products + Protocols

In retrospect, ORCA's toolkit orientation set us apart from the GENI project's initial focus on standardizing protocols to enable existing network testbeds to interoperate. Many of our colleagues in the project understood ORCA as another testbed provider, rather than as a platform to federate and orchestrate diverse providers. They focused on the infrastructure that ORCA supported, which at that time was limited to Xen virtual machine services [15]. There was less interest in the toolkit itself, in part because ORCA used a different

language (Java) and tooling than the other GENI clusters. Our focus on the toolkit suggested a rigid development model and a relaxed approach to protocol standards, which are essential for interoperability.

Even so, the ORCA toolkit accelerated development by using generic ORCA servers to "wrap" existing back-end systems and call them through their existing APIs. The result looked much like the structure ultimately adopted in GENI (§4.2), but using the ORCA protocols rather than the GENI standards. The ORCA experience suggests that the lengthy GENI development phase could have been shortened by focusing on wrappers and adapters in the early spirals, rather than the protocols.

Moreover, if the wrappers are standardized, then it is possible to change the protocols later by upgrading the wrappers. We found that it is easier to stabilize the plug-in APIs for the toolkit than the protocols themselves. For example, ORCA uses an RPC system (Axis SOAP) that has never served us well and is slated for replacement. The GENI standards use XMLRPC, which is now seen as defunct.

Although it is always difficult to standardize protocols, interoperability in a system like ORCA or GENI is less about protocols than about data: machine-readable descriptions of the principals and resources. In both systems the protocols are relatively simple, but the messages carry declarative resource descriptions and credentials, which may be quite complex. Our standards for these languages will determine the power and flexibility of the systems that we build (§4.4).

4.2 Federation

The key differences in control framework architectures relate to their approaches to federating the aggregates. In general, the aggregates themselves are IaaS or PaaS services similar to those being pursued by the larger research community and in industry. The problem for GENI is to connect them.

The GENI framework takes a simple approach to federation: it provides a common hierarchical name space (URNs) and central authorities to authorize and monitor user activity. GENI leaves orchestration to users and their tools and it does not address cross-aggregate resource control (§4.3). Even so, the GENI community invested substantial time to understand the design alternatives for federation, reconcile terminologies, and specify a solution. Various architectures were proposed for GENI to factor identity management and authorization functions out of the standalone testbeds and into federation authorities, but a workable convergence did not emerge until 2012.

The GENI solution—so far—embodies a design principle also used in ORCA. The AMs do not interact directly; instead, they merely delegate certain functions for identity management, authorization, and resource management to common coordinator servers. The coordinators issue signed statements certifying that clients and their requests comply with federation policy. The AM checks these statements before accepting a request.

For example, the GENI Clearinghouse authorities approve users, authorize slices, and issue credentials binding attributes to users and slices. User tools pass their credentials to the AMs. These mechanisms provide the common means for each AM to verify user identity and access rights for the GENI user community. The coordinators in ORCA/ExoGENI include brokers and the GENI controllers, which are trusted by the AMs to check the GENI credentials of requests entering the ORCA/ExoGENI enclave.

As originally conceived, ORCA AMs delegate these user authorization functions to the brokers: if a community broker issues a ticket for a request, the AM accepts the user bearing the ticket. It is the responsibility of the broker to authorize each user request in its own way before granting a ticket. More precisely, the ORCA architecture left the model for user identity and authorization unspecified. It is fully compatible with GENI's choices in this area, which we contributed to and embrace. However, these choices should remain easily replaceable in any given deployment (§4.5).

4.3 Orchestration

GENI has not specified any coordinator functions beyond checking user credentials. In particular, GENI has not adopted brokers or any form of third-party ticketing to enable cross-aggregate resource management.

Although the sponsor (NSF) has voiced a desire to control user resource usage across multiple aggregates, GENI has defined no alternative mechanism for this purpose. Importantly, AMs and controllers are not sufficiently powerful to meet this need without some structure equivalent to brokers. The local policy of any AM may schedule or limit allocation of its own resources, but it has no knowledge or control over allocations on other aggregates. Similarly, any limit that an unprivileged controller imposes on resource usage by a slice is voluntary, because the controller acts as an agent of the slice and its owners.

Slice controllers are also not part of the GENI architecture. GENI was conceived as a set of protocols and service interfaces: the client software to invoke these interfaces was viewed as out of scope, notwithstanding a 2009 vote in the Tools Working Group. Instead, the idea was that a standard AM API would encourage an ecosystem of user tools to grow organically from the community. To the extent that computation is needed to orchestrate cross-aggregate requests for a given slice—such as topology mapping—those functions were conceived as central services provided to the tools through new service APIs. We believe that it is more flexible and scalable to provide these functions within the tools. ExoGENI shows that it is possible to do so given sufficiently powerful resource descriptions and a platform for building the tools.

Over time it became clear that the GENI clients are stateful. In particular, tools must maintain state to implement timer-driven sliver renewal, multi-step atomic co-allocation, and "tree" stitching across aggregates. Later in the project, the GENI Project Office developed an extensible tool called `omni` and a Web portal to proxy requests from stateless tools into GENI. These clients have steadily incorporated more state and functionality. It seems likely that they will continue to evolve in the direction of controllers.

The ORCA view is that a federated infrastructure control plane is a "tripod": all three server types—aggregate/AM, controller/SM, and broker—are needed for a complete system. The factoring of roles across these servers is fundamental. The AMs represent the resource providers, and are the only servers with full visibility and control of their infrastructures. The SMs represent the resource consumers, and are the only servers with full visibility and control over their slices. The brokers and other authorities (e.g., the GENI clearinghouse) mediate interactions between the SMs and AMs: they are the only servers that can represent policy spanning multiple aggregates.

4.4 Description Languages

Our experience with ORCA and GENI deepened our view that the key problems in federated infrastructure are largely problems of description. This understanding is a significant outcome of the GENI experience.

GENI differs from other infrastructure services primarily in its emphasis on diverse infrastructure, rich interconnection, and deep programmability. It follows that the central challenges for GENI are in describing "interesting" resources and in processing those descriptions to manage them.

The early development phase of GENI was marked by an epic debate on `dev@geni.net` about whether a common framework for diverse resource providers is even possible. It is perhaps still an open question, but if the answer is yes, then the path to get there involves automated processing of rich resource descriptions. To incorporate a new resource service into an existing system, we must first to describe the service and its resources in a way that enables generic software to reason about the space of possible configurations and combinations.

For example, the ORCA experience shows that it is easy to incorporate current cloud systems and third-party transit network providers as GENI aggregates through an adaptation layer if we can describe their resources logically. Powerful logical descriptions also enable the various coordinator functions (§4.3) in ORCA/ExoGENI. One lesson of this experience is that AM advertisements do not in general describe the infrastructure *substrate*, as the GENI community has understood them, but instead describe infrastructure *services*, which are even

more challenging to represent and process. For example, AMs may proxy or "resell" resources from other providers whose substrate they do not control, or they may offer various mutually exclusive options for virtual sliver sets within the constraints of a given substrate resource pool.

4.5 Logical Trust

Our ongoing research focuses on declarative representations for trust and authorization, as well as for resources. The ORCA team at Duke was deeply involved with development of the GENI trust and authorization system (2010-2013). In recent work we have shown how to specify trust structure and policy alternatives for a GENI deployment concisely and precisely in SAFE declarative trust logic [7].

The GENI architecture can be built as a set of autonomous services (e.g., the AMs) linked by a declarative trust structure. The various coordinator roles and trust relationships are captured in declarative policy rules rather than as procedures or assumptions that are "baked in" to the software.

SAFE is a platform to facilitate this approach to building secure networked systems. Statements in the logic are embedded in certified credentials generated by the platform. A generic compliance checker validates credentials according to policy rules, which are also expressed in the logic. A SAFE script wrapper makes it easy to build and manage credentials using embedded templates, and check received credentials for compliance with a logical security policy.

It takes about 150 lines of scripted SAFE logic to represent the GENI trust structure, authorization rules, credential templates, and credential flow. The SAFE approach has potential to accelerate development and deployment of systems like GENI. It balances low implementation cost with flexibility for deployments to accommodate diverse policies of their members, evolve their structures and policies over time, and federate with other deployments.

These are the same goals that motivated the design of ORCA. Note that ORCA and SAFE are complementary. For example, they may be used together to build GENI.

5 Conclusion

We embarked on the ORCA project more than a decade ago to build a foundation for networked infrastructure (IaaS) services. Our goal was to pin down a "thin waist" that was simple and deployable, but also sufficiently flexible and powerful to support basic resource control for a wide range of infrastructures and sharing models. By *resource control* we simply mean an ability to manage and dispense resources in measured and metered quantities according to policies.

Our solution was to develop ORCA. The GENI project brought a welcome opportunity to put ORCA to the test. There followed a six-year effort to realize the GENI vision and to build and deploy ExoGENI on the ORCA platform. This chapter summarizes the principles and elements of ORCA and discusses which were successful in GENI and which were not.

6 Acknowledgements

We thank RENCI, NSF, IBM, and the GENI Project Office (GPO) at BBN for their support. Many colleagues at GPO and other GENI projects have contributed to our efforts and/or influenced our thinking. Various students and colleagues have collaborated at various stages of the project, as represented by their authorship of the works cited below. Many have also contributed substantially to the ORCA core code base, including Aydan Yumerefendi, David Irwin, and Laura Grit, and also Varun Marupadi, Victor Orlikowski, Matt Sayler, and Ken Yocum.

References

[1] I. Baldine, Y. Xin, D. Evans, C. Heerman, J. Chase, V. Marupadi, and A. Yumerefendi. The Missing Link: Putting the Network in Networked Cloud Computing. In *International Conference on the Virtual Computing Initiative*, May 2009.

[2] I. Baldine, Y. Xin, A. Mandal, C. Heerman, J. Chase, V. Marupadi, A. Yumerefendi, and D. Irwin. Autonomic cloud network orchestration: A GENI perspective. In *GLOBECOM Workshops: 2nd IEEE International Workshop on Management of Emerging Networks and Services (MENS 2010)*, December 2010.

[3] I. Baldine, Y. Xin, A. Mandal, P. Ruth, A. Yumerefendi, and J. Chase. ExoGENI: A multi-domain infrastructure-as-a-service testbed. In *TridentCom: International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, June 2012.

[4] R. Braynard, D. Kostić, A. Rodriguez, J. Chase, and A. Vahdat. Opus: an Overlay Peer Utility Service. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.

[5] J. Chase, I. Constandache, A. Demberel, L. Grit, V. Marupadi, M. Sayler, and A. Yumerefendi. Controlling Dynamic Guests in a Virtual Computing Utility. In *International Conference on the Virtual Computing Initiative (ACM Digital Library)*, May 2008.

[6] J. Chase, L. Grit, D. Irwin, V. Marupadi, P. Shivam, and A. Yumerefendi. Beyond Virtual Data Centers: Toward an Open Resource Control Architecture. In *Selected Papers from the International Conference on the Virtual Computing Initiative (ACM Digital Library)*, May 2007.

[7] J. Chase and V. Thummala. A Guided Tour of SAFE GENI. Technical Report CS-2014-002, Department of Computer Science, Duke University, June 2014.

[8] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.

[9] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the Twelfth International Symposium on High Performance Distributed Computing (HPDC)*, June 2003.

[10] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[11] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual Machine Hosting for Networked Clusters: Building the Foundations for "Autonomic" Orchestration. In *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November 2006.

[12] L. E. Grit. *Extensible Resource Management for Networked Virtual Computing.* PhD thesis, Duke University Department of Computer Science, December 2007.

[13] J. Ham, F. Dijkstra, P. Grosso, R. Pol, A. Toonk, and C. Laat. A Distributed Topology Information System for Optical Networks Based on the Semantic Web. *Journal of Optical Switching and Networking*, 5(2-3), June 2008.

[14] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-Recharging Virtual Currency. In *Proceedings of the Third Workshop on Economics of Peer-to-Peer Systems (P2P-ECON)*, August 2005.

[15] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proceedings of the USENIX Technical Conference*, June 2006.

[16] H. Lim, S. Babu, and J. Chase. Automated Control for Elastic Storage. In *IEEE International Conference on Autonomic Computing (ICAC)*, June 2010.

[17] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi, and J. Chase. Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control. In *Supercomputing (SC06)*, November 2006.

[18] M. Sridharan, W. Zeng, W. Leal, X. Ju, R. Ramanath, H. Zhang, and A. Arora. From Kansei to KanseiGenie: Architecture of Federated, Programmable Wireless Sensor Fabrics. *Proceedings of the ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, May 2010.

[19] Y. Xin, I. Baldine, A. Mandal, C. Heerman, J. Chase, and A. Yumerefendi. Embedding virtual topologies in networked clouds. In *6th International Conference on Future Internet Technologies (CFI 2011)*, June 2011.

[20] A. Yumerefendi, P. Shivam, D. Irwin, P. Gunda, L. Grit, A. Demberel, J. Chase, and S. Babu. Toward an Autonomic Computing Testbed. In *Workshop on Hot Topics in Autonomic Computing (HotAC)*, June 2007.