ORCA Control Framework Architecture and Internals

Jeff Chase
Computer Science Department
Duke University
{chase}@cs.duke.edu

November 13, 2009

Abstract

This document incorporates material from a variety of previously published works with various authors, including papers and theses from David Irwin, Laura Grit, and Aydan Yumerefendi. It supersedes related material from those sources.

1 Introduction

Orca is a software framework and open-source platform to manage a programmatically controllable shared substrate, which may include servers, storage, networks, or other components. This class of systems is often called cloud computing or utility computing.

The Orca software is deployed as a control framework for a prototype GENI facility. We see GENI as an ambitious futuristic vision of cloud networks as a platform for research in network science and engineering.

An Orca deployment is a dynamic collection of interacting control servers (*actors*) that work together to provision and configure resources for each guest according to the policies of the participants. The actors represent various stakeholders in the shared infrastructure: substrate providers, resource consumers (e.g., GENI experimenters), and brokering intermediaries that coordinate and federate substrate providers and offer their resources to a set of consumers.

Orca is based on the foundational abstraction of *resource leasing*. A lease is a contract involving a resource consumer, a resource provider, and one more brokering intermediaries. Each actor may manage large numbers of independent leases involving different participants.

1.1 Name Games: Architecture and Platform

Orca is actually the name for a software architecture and an umbrella project to develop that architecture and explore various research questions. The Orca platform is an evolving set of software components that implement and extend the Orca architecture. Many of these components are separately named, and papers have been written about them using these names at various points in our research. Orca project software includes the Shirako leasing core [11]; the Automat [14] control portal and related components; a new implementation of the SHARP framework [9] for accountable lease contracts and brokering; Cluster-on-Demand (COD [5]), a back-end resource manager for shared clusters; and driver modules to interface the system to various virtualization technologies (e.g., Xen) and guest environments (e.g., cluster/grid middleware, Web services, workload generators, etc.).

There is some confusion about when to use the name Orca rather than the names of the various elements. This is understandable given that the usage has changed with time and the umbrella name Orca was introduced relatively late. We generally use "Orca" to talk about the overall platform or architectural principles extracted from the overall project, and now use the other names only to make statements about specific implementations. The distinction is important because we expect these implementations to evolve over time. Also, we envision that in the future there may be multiple implementations that interoperate using architecturally defined protocol interfaces (e.g., over SOAP or XML-RPC), and these implementations could vary in many ways from our current Java-based prototypes. Orca is protocol-centric, while Shirako/COD is bound to Java.

1.2 Slices, Slivers, and Guests

Cloud computing systems, hosting utilities, and GENI are instances of the *host/guest model*. We need a simple term because the host/guest model dominates this era of distributed computing in the same way that the client/server model dominated the previous era.

Orca and its predecessors are designed for the class of host/guest systems based on an "infrastructure as a service" model, such as GENI. Resources from a hosting substrate are partitioned and allocated ("sliced and diced") into private isolated *slices* on behalf of various consumers or applications. A slice gives its owner control over some combination of virtualized substrate resources assigned to the slice.

Following GENI terminology, substrates consist of collections of physical *components*. *Substrate providers* control the allocation and use of collections of components (*aggregates*). The GENI substrate may include virtual servers, storage, programmable network elements, networked sensors, mobile/wireless platforms, configurable instrumentation, and other programmable infrastructure components attached to the cloud network.

A *sliver* is the smallest unit of some resource that is independently programmable and/or independently controllable in some resource-specific fashion. Each sliver is granted from a single substrate provider. Substrate providers might employ various virtualization technologies to instantiate slivers on their components.

The component and sliver abstractions are intended to be quite general (see Section 2.3). To make these vague abstractions more concrete, consider the simple example of virtual cloud computing, i.e., a virtual machine hosting service. The substrate provider maintains clusters of servers, and uses virtual machine technology to instantiate virtual machines (VMs) on them. A VM is a simple and familiar form of sliver, allocated from a simple and familiar form of component: a virtualizable computer.

Slices may combine multiple slivers from multiple substrate providers. In essence, a slice is a grouping mechanism for slivers. Slices are built-to-order for some specific application or purpose, embodied in software that runs within the slice. A key goal of GENI is to enable researchers to experiment with radically different forms of networking by running experimental systems within their isolated slices.

The software or system (or experiment) that inhabits a slice is a *guest*. The term *guest* is generalized from virtual machine systems, which refer to the image instance running in a domain (VM) on a host. Since the goal of the project is to generalize virtual slicing to diverse aggregations of substrate components, it is reasonable to use the same term for the inhabitant of the slice. *Terminology note.* The term *guest* is not yet accepted in GENI, which limits its focus to guests that are "experiments". We are trying to hold on to a more general term because control framework software generalizes to other kinds of guests.

In general, a guest is a distributed software environment running within collection of slivers configured to order, possibly from different substrate providers. Some guests will be long-running services that require different amounts of resources at different stages of execution. The guests may range from virtual desktops to complex experiments to dynamic instantiations of distributed applications and network services, such as robust peer-to-peer services, content delivery networks, multi-tier Web services, or job management systems (grids).

1.3 ORCA Principles

Orca grew out of foundational research on controlled resource sharing and automated configuration for networked substrates, rather than the needs of any specific deployment. It is based on a general model for dynamic, brokered leasing that strives to be substrate-neutral, policy-neutral, and guest-neutral.

The design of Orca reflects principles that emerged from a thread of research in minimalist operating system design a decade ago [8, 2]. The system kernel should concern itself narrowly with physical resource management and physical resource abstractions, decoupled from the specific programming abstractions (e.g., processes, threads, files, jobs) that are the building blocks of guest applications. It should expose interfaces to suitably privileged components to control resource allocation policy. Allocation decisions must be made visible to the hosted software to permit robust adaptation.

The Orca model seeks to extend these ideas to a diverse networked substrate with open, flexible, secure, robust, and decentralized control. It is based on a few key principles:

- *Sustainable structure and autonomy.* All participants have the ability to quantify and control what they contribute to a system and what they obtain from it through time. The structure protects their autonomy to exercise this control according to local policies.
- *Negotiated contracts with varying degrees of assurance including but not limited to strong isolation (reservations).* The control plane provides a means to form *contracts* for specific resource allocations at specific times. However, different providers and virtualization technologies may offer varying degrees of isolation. What is important is that the contracts are explicit about the assurances they offer, and over what periods of time.
- *Neutrality for guests and resources.* The architecture provides extensible programmatic interfaces to instantiate, configure, and control a wide range of guest software environments on a wide range of resources through an elemental leasing abstraction (the "wasp waist" of the architecture).

- *Policy neutrality with a balance of local autonomy and global coordination.* Orca must be free of resource allocation policy: needs and capabilities evolve with time. Consumers determine their policies for requesting resources, and providers determine arbitration policy for the resource pools under their control. Providers may delegate control of resources to other entities, who may subdivide their holdings according to their own needs via brokering intermediaries that represent their policies. The system has no central point of trust or control.

Orca derives from the SHARP resource peering model [9]. SHARP emphasizes exchanges of contracts among authenticated autonomous entities, who are accountable for their commitments. These stakeholders are represented by actors in the resource control plane. Actors are presumed to be independent, self-interested, and strategic.

Resource leases are contracts for access to resources, following the principles above: they are specific about the assurances being made by both parties, and they are active over mutually agreed intervals of time. Resource providers can retain varying degrees of scheduling control over their resources by adjusting the terms and attributes of the contracts they enter into. Each lease has an agreed termination time: leases make scheduling possible.

The contract mechanism enables providers to delegate varying degrees of control over their resources to brokers that represent the resources and policies of a community. The *resource delegation* mechanisms enable a continuum of deployment choices balancing provider autonomy with coordination across multiple providers. At one end of the continuum, providers maintain complete control over their resources, and consumers negotiate resource access with each provider separately. At the other end of the continuum, a set of providers federate their resources under a coordinated *facility* with common resource arbitration and authorization policies. Many other points are possible within the continuum. For example, providers may expose and withdraw portions of their resources in different ways at different times to different facilities, according to their local policies. These are deployment choices, not architectural choices.

While slices for experiments in GENI are often spoken of as static entities provisioned from a declarative description or a Web interface, much of our research emphasizes *dynamic guests* that monitor changing conditions and adapt as their needs change. In particular, Orca was designed to support long-running hosted services that must maintain service quality across flash crowds, resource faults or stutters, and other changes. To this end, it exports programmatic, service-oriented interfaces for self-managing guests to negotiate for resources and configure them on-the-fly. (See Section 8.) While these APIs are designed to enable feedback control policies [3, 14, 4, 13, 12, 7, 10], they can be used to bolt on experimenter tools in GENI.

2 Overview

2.1 Actors

There are three basic actor roles in the architecture, representing the providers, consumers, and intermediaries respectively. Figure 1 depicts these actor roles and their interactions. There can be many instances of each actor type, e.g., representing different substrate providers or resource consumers.

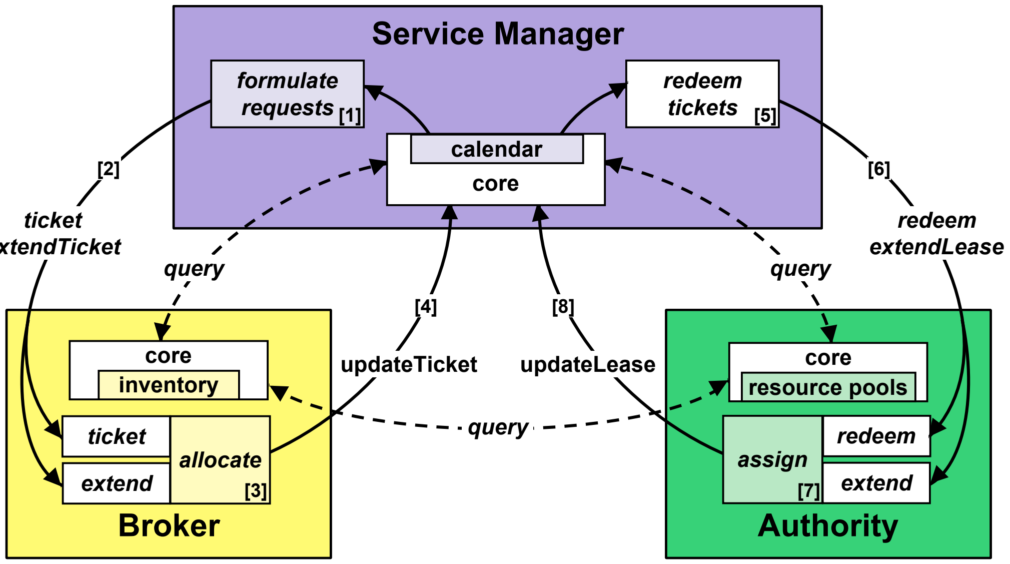

Figure 1: Leasing interactions and extension points for the leasing system. Colored boxes within each actor represent the controller policy module plugins to the leasing core. The arrows illustrate the leasing protocols. A service manager (slice controller) formulates its requests [1] and issues one or more `ticket` or `extendTicket` requests to the broker [2]. The broker determines the allocation of its inventory to requests [3] and returns tickets via `updateTicket` [4]. Service managers redeem their tickets for leases by sending `redeem` or `extendLease` messages to the authority [5] and [6]. The authority assigns resources to requests [7] and sends an `updateLease` to the service manager to return leases or signal changes to the lease status [8]. A `query` request returns an attribute list for an actor. On lease state changes the core also upcalls event handler plugins (not shown) registered for each lease or resource type.

Authority or Aggregate Manager (AM). An *authority* actor controls access to some subset of the substrate components. It corresponds directly to the *aggregate manager (AM)* in GENI. Typically, an authority controls some set of infrastructure resources in a particular site, autonomous system, transit domain, administrative domain, or component aggregate comprising a set of servers, storage units, network elements, or other components under common ownership and control.

Terminology note. The term *site* or *site authority* (e.g., a cluster site or hosting center) is often used to refer to a substrate authority/AM, as a result of our roots in virtual cloud computing. For network substrates we are using the term *domain authority* more often when it is appropriate.

Slice/Service Manager (SM) or Slice Controller. This actor is responsible for creating, configuring, and adapting one or more slices. It runs on behalf of the slice owners to build each slice to meet the needs of a guest that inhabits the slice.

Terminology note. This actor was originally called a *service manager* in SHARP (and in the Shirako code) because the guest was presumed to be a service. As GENI has developed, we have adopted the term *slice controller* because the actor's role is to control the slice, rather than the guest itself, and because in GENI the guest is an experiment rather than a service. *Slice Manager* (also SM) is also OK since the "controller" is properly speaking a plugin module to the actor itself. (See Section 4.1.)

Broker. A broker mediates resource discovery and arbitration by controlling the scheduling of resources at one or more substrate providers over time. It may be viewed as a service that runs within a GENI *clearinghouse.* A key principle in Orca is that the broker can have specific allocation power delegated to it by one or more substrate authorities, i.e., the substrate providers “promise” to abide by allocation decisions made by the broker with respect to their delegated substrate. This power enables the broker to arbitrate resources and coordinate allocation across multiple substrate providers, as a basis for federation and scheduling of complex slices across multiple substrate aggregates. Brokers exercise this power by issuing *tickets* that are redeemable for leases.

We emphasize that there may be many instances of each actor role participating in an Orca control plane at any given time. Actors may join and depart. There is no inherent central point in the architecture. *Terminology note:* we use the term “actor” to refer to both the code for an actor and a running actor instance. The distinction is analogous to class and instance in object-oriented languages. For example, if we refer to “the broker” in discussing the code, it does not mean that there can be only one broker in the system.

2.2 Identity

Actors represent stakeholders in the shared infrastructure, such as contributing resource providers or experimenters using a GENI testbed facility.

Each stakeholder is associated with a principal. Principals possess asymmetric keypairs, and are bound to identities in the real world.

Each actor possesses a private key for each principal it represents. Typically each actor acts on behalf of a single principal who creates the actor. *Note.* In the code, each actor is its own principal, and maintains exactly one keypair, which is configured or minted when the actor is created. The intent is that the actor’s public key can be endorsed by the user principal on whose behalf the actor is acting, if the actor does not use the principal’s keys directly.

Actors become aware of principals by receiving signed endorsements of their public keys from other principals. An X.509 certificate is one form of endorsement. Each actor is initialized with one or more principals whom it trusts to issue endorsements, including a *root* principal that manages that actor.

GENI note. We believe that identity management, key binding to real-world principals, and accountability of those principals should be handled by external identity providers (IdPs) and related services, and left out of GENI.

2.3 Resource Model

Substrate providers may choose how much substrate detail to expose to the broker through their delegations and advertisements, and over what intervals of time. The architecture presumes that providers can represent their substrate in sufficient detail to enable users to match and select against the resource inventories, and to enable brokers to arbitrate in a useful way among contending requests.

As stated previously, a slice or lease is exposed to the guest and its SM as some set of “slivers”, which are instantiate on “components” managed by the AM. In essence, what the AM advertises to the broker is an ability and willingness to create slivers of various kinds over various periods of time.

The component and sliver abstractions can cover a lot of very different resource scenarios. For example, a "component" could itself be a complex aggregate with lots of internal moving parts and control interfaces. "Sliver" is an abstraction whose concrete meaning for each specific substrate depends on the nature of the substrate, the virtualization technologies used to slice-and-dice the substrate, and what the AM chooses to expose about the substrate. A "sliver" might span multiple components, or it might only use a part of a component based on some internal mapping or virtualization. It is up to a substrate provider (AM owner or integrator) to choose the right level of abstraction to expose for components and slivers on a particular kind of substrate.

Let us try to define these terms more precisely.

A *component* is any piece of substrate that is exposed to the AM as an element that is named, controlled, configured, slivered, and programmed independently of other components. In general, components are always hidden behind an AM interface, and are not exposed outside of the AM unless the AM specifically chooses to expose them. For example, an AM might allow slices to allocate physical components directly by defining a static one-to-one mapping of slivers to components.

A *sliver* is any virtualized resource or piece of a slice that is exposed to the guest and its SM as an element that is named, controlled, configured, allocated, and programmed independently of other slivers. Slivers instantiated at the same substrate provider (authority/AM) are grouped into leases according to type and interval of validity.

Orca views the substrate as a set of collections (*pools*) of typed resource elements. Each authority/AM maintains pools of typed components, and *advertises* or *delegates* to the broker pools of typed slivers that it can instantiate on those components. Each delegation or advertisement is valid over a specified interval of time.

The resource *type* captures what is known about a component or sliver and what it can do. Resource types have attributes. These type attributes are common to all units of the resource type, and are presumed not to change. Resource type attributes are advertised by the authority/AM in its delegations to the broker.

One use of resource type attributes is to specify the degree of assurance or isolation promised by the substrate provider for slivers of that type (e.g., best effort, minimum share, exclusive access).

Resource discovery is a process of searching and matching resource types and pools based on their attributes, so that potential users of resources may locate and select resources that match their needs. The Orca code base currently supports a weak form of resource discovery: SM may `query` a broker's inventory for a list of all resource types in the broker's *inventory*, and the attributes of those types. It is possible to install more sophisticated matching policies by extending the `query` implementation in the broker policy module.

Section 4.2 discusses resource types in more detail. Section 3.4 discusses the mapping of the resource model onto diverse substrates in the GENI context.

2.4 Leases and Tickets

A *lease* is a contract assuring the holder access to some set of resources (a *resource set*) for a defined renewable time period (the *term*). Each lease is bound to a single slice, and covers resources (slivers) of a single type from a single authority/AM. The holder of a lease may request to renew (extend) the lease before it expires.

Each lease is initiated by a request from an SM to a broker for u logical units (slivers) of resource of type r. If approved, the broker issues a *ticket* for the u units from some specific pool, and the SM redeems the ticket at the AM for the pool to obtain a lease for u specific units.

In general, the AM is free to determine how to map slivers onto its components, and which specific units of a type to assign to fill a ticket. However, the SM may pass additional attributes with the `redeem` to specify the request more fully, and this may drive and/or constrain the authority's assignment policy.

GENI note: GENI participants often speak of slivers as allocated one at a time through some sliver interface. In ORCA a ticket or lease may cover many slivers of the same type from the same AM.

2.5 Objects and Naming

The ORCA actor protocol operates on five kinds of objects: slices, leases, slivers, pools, and principals. There are various other objects internal to each actor or container, e.g., objects representing the actors themselves and plugins installed in the actors.

Each object is named by a unique identifier, which is an RFC 4122 GUID. The GUID names the object at every actor that has a local record of the logical object.

Each object has an attached property list of named attributes. The properties are passed to actors and plugins that operate on the object, which may add to the property list or modify it in well-defined ways. Properties are discussed further in Section 2.11.

Each actor maintains state pertaining to the leases it knows about. We use the term *lease object* to refer to the state object for such a contract within an actor at all stages of the lease lifecycle, even if a contract is not yet in force or has expired. Each lease is initiated by an SM, and must be approved (ticketed) by a broker and granted by an AM.

The GUID for an object is assigned by the actor that creates it.

- The GUID for a lease is selected by the SM that requested it. The lease properties are a union of the resource type properties, derived by the broker from the containing resource pool, and configuration properties specified by the requesting SM.
- The GUID for a sliver is assigned by the granting AM and is returned in any lease covering the sliver.
- The GUID for a slice is assigned by the SM that wishes to create a slice for the purpose of grouping its leases. Creating a slice is not a privileged operation. The creating SM may also attach properties to the slice.

For convenience, slices and some other objects also have user-assigned symbolic names. There is a single flat symbolic name space and uniqueness of symbolic names is not assured. Protocol exchanges identify objects by GUID, and not by symbolic name.

Note: the code currently assumes that GUIDs are indeed unique, as they should be if RFC 4122 is respected. If there is concern about accidental or malicious GUID collisions, then we could mint a secure identifier by extending the GUID with the public key of the principal that creates the object.

2.6 Authorization policy

Every operation is requested on behalf of some principal (the subject) and operates on an object. The authorization policy approves or denies each requested operation based on the subject, the object, and the nature of the operation.

An actor determines what rights to assign to a principal based on security assertions attached to endorsements it receives for that principal's public key. There are three forms of security assertions of interest.

- **Security attributes.** A *security attribute* is a property of the principal that may be queried by an authorization policy. Example: "This principal is one of Chase's students." Attributes are a general basis for Attribute-Based Access Control (ABAC). A primary role of Shibboleth Identity Providers (IdPs) is to certify attributes for an authenticated identity.
- **Delegation commands.** A principal may delegate a subset of its rights to another principal by issuing an endorsement specifying the rights to be delegated. A delegation chain rooted in a trust anchor is proof that the endorsed principal has specific rights. This form of capability is often called a *credential*.
- **Resource contracts.** Resource server actors (brokers and authorities) may delegate specific rights to specific resources at specific times to a specific principal by issuing resource contracts (tickets or leases) to that principal. This class of endorsements includes tickets and leases in SHARP-derived systems such as ORCA.

ORCA as an architecture supports flexible authorization policies in the resource servers (authorities or AMs and brokers), based on external endorsing trust anchors such IdPs, Slice Authorities, Management Authorities, GENI facility management, and so on. An actor may receive endorsements, credentials, and delegations attached to a request, or it might fetch them on demand using some form of distributed storage and recovery service. The code has support for signed tickets and leases and a pluggable authorization policy upcalled on requests to objects on resource servers (brokers and authorities/AMs). ORCA provides a bare-bones authorization policy based on the following simple ACL rules.

- Slices are owned by principals. Leases are held by principals with rights to the containing slice. The holder of a lease has rights to all slivers instantiated as a result of redeeming the lease.
- The holder of these rights to objects may delegate them to other principals for the duration of the object's existence. Revocation is not supported.
- The actor root is empowered to take any operations on the actor. For example, it may issue endorsements for identity providers that the actor should trust to issue X.509 certificates for principals, or brokers that an authority should trust to manage its resources, or that a slice controller should trust to handle requests for resources.
- The actor root may delegate some or all of its rights to some other principal, such as GENI operations (GMOC). An authority root may also grant specific rights to brokers it chooses to trust, for example, the right to suspend or shut down slivers instantiated as a result of

accepting tickets issued by the broker. The broker in turn may delegate those rights. For example, a broker associated with GENI could delegate those rights to GMOC.

Note. Implementation of this policy incomplete. A language like SAML should be used for security assertions, but the current implementation uses more ad hoc representations. Revocation is an issue.

2.7 Actor Interfaces

Protocol messages include resource requests, resource delegations/advertisements, contracts, generic queries and responses, and configuration commands. The issuer digitally signs these elements, or the messages that contain them.

Many protocol messages have *property lists* attached pertaining to objects named in the message. Property lists provide a generic mechanism to extend the actor information flow without changing the basic message formats (e.g., WSDL). Properties are discussed further in Section 2.11.

The protocol messages can be grouped as follows:

- **Lease actions.** The basic inter-actor protocol operations in Figure 1 are operations on lease objects named by GUIDs. These include asynchronous notifications of changes to the lease status. The leasing protocols are fundamentally *asynchronous*: the response to a request for resources is delivered as a notification when the request is granted (*ticket issue*) or denied, and a second notification when instantiation completes (*lease grant*) or fails.
- **Generic queries.** In addition, any actor may request general information from another using a `query`. The query interface is extensible: each `query` passes a property list as an argument and returns a property list as a result.
- **Management interface.** The *management API* is a lower-level interface to query and modify state within an actor. It is intended to support external management and control tools, such as a Web portal interface, rather than interaction among stakeholders (actors). See Section 2.12.
- **Sliver control.** Specific substrate providers may export additional interfaces to control slivers. For example, it may be useful to have interfaces to reset or restart individual slivers. For complex slices that link different slivers of different types and/or from different providers, additional interfaces may be needed to *stitch* or link them together in the slice data plane.
- **Other.** Additional operations are needed to exchange, discover, and query identity credentials and endorsements. There should be some kind of generic publish/subscribe framework for instrumentation streams.

Todo: make a summary figure for the query/ticket/redeem sequence parallel to the one for ProtoGENI.

2.8 Actor State Registries

Each actor maintains persistent state in a local repository (database). The actor repository includes records for several kinds of objects.

- **Lease registry.** Each actor maintains a calendar-indexed registry of leases that are active or pending and are known to the actor. For example, a broker maintains a lease object for each pending ticket request or unexpired ticket issued by the broker. Lease object state includes any signed tickets or leases pertaining to the contract.
- **Slice registry.** Each actor maintains a registry of slices that are bound to some lease object in the lease registry. A slice record includes its GUID, its symbolic name, a list of properties, and references to principals associated with the slice. An actor may discard slice records with no active or pending tickets or leases.
- **Principal registry.** Each actor maintains a registry of known principals. The registry record for a principal includes its public key, a property list of security attributes, the endorsing entity and expiration time for each attribute, and associated endorsement certificates received by the actor.

Note that there is no global registry of objects in the system: each actor maintains state for only the objects that it participates in managing. Note also that the SM actors are first-class stateful entities with persistent registries.

2.9 The Leasing Core and Plugins

We use the Shirako toolkit to build ORCA actors in the Java environment. Shirako may be thought of as a distributed lease manager: it implements common logic for a set of protocols and conventions for actors to negotiate and coordinate lease contracts, and maintain their state registries.

Each Shirako-based actor is a multi-threaded server that is written in Java and runs within a Java Virtual Machine. Actors communicate using an asynchronous peer-to-peer messaging model through a replaceable stub layer.

Shirako is designed as a common leasing core with generic actor shells and lease state machines, together with plugin APIs for guest-specific, resource-specific, or policy-specific components. Shirako is dynamically extensible through *extension packages* that hook *plugins* to these APIs. Section 4.1 discusses plugins and their interfaces in more detail.

The core can be viewed as a library linked into each actor implementation. The plugins are used to customize the actor for some specific needs, and interface it to outside elements (e.g., a particular substrate or management tools).

The core orchestrates the workflow to acquire and manage leased resources by upcalling the plugins when they are needed to take some decision or action.

The core is driven by clocked lease state machines that initiate actions as needed to maintain timed lease contracts. For example, leases expire if they are not renewed (extended), but lease renewal ("meter feeding") is automated in the SM.

All inter-actor protocol messages are sent and received by the leasing core. Code layered above the core can observe and control the process through various plugin APIs. Each actor has at least one *policy controller* plugin. The policy controllers use downcall APIs on a local lease store to track their resource holdings and mark lease objects to drive the flow of requests and configuration actions. Section 4 gives an overview of the internal plugin interfaces and plugin-related software, including calendar scheduling, resource control policies, and substrate-specific management.

2.10 Actor Containers

Multiple actors may inhabit the same JVM (*container*) and interact through local procedure calls. Actors in the same container share a per-container database and a common actor registry and management tools.

Each container has a keypair and GUID. A container administrator (also called the container owner, operator, or root) can control the actors within the container. In particular, the container administrator may act as the actor administrator (also called the actor owner, operator, or root) for any actor in the container. It may delegate actor ownership to other principals on a per-actor basis.

Various tools, classes, and methods define actor configuration and how actors associate with each other. This may be accomplished through an automated policy, or the actor root may do it manually, e.g., through a Web portal interface. *Todo:* we need a global registry for resource control servers, so the SMs can find them.

2.11 Property Lists

The lease protocol interactions in Figure 1 must carry whatever information is needed to guide resource management and configuration. For example, a slice manager/controller (SM) may need to pass specific requirements for configuring a resource to an authority/AM. Similarly, an authority must pass information about each resource unit back to the SM so that the guest can use the resource effectively.

For example, consider the IP address of a newly leased guest node. In principle, it could be either the AM or the SM that assigns the address, depending on how network address space is managed in a deployment. Whichever actor manages the address space, it must pass the address to the other actor. As another example, an SM requests resources, but these requests are subject to an arbitration policy at the broker and an assignment policy at the authority, which may require detailed information about the guest's preferences.

Rather than trying to "screw down" these various design choices, we instead factored any resource-specific and policy-specific information out of the generic leasing protocols. Controllers may interact by exchanging *property lists* piggybacked on the interaction protocols, in a manner similar to proven extensible protocols such as HTTP and WebDAV. The property lists are sets of $[key, value]$ string pairs.

Controller policy modules and handlers may attach property lists as attributes to requests, tickets, and leases. They flow from one actor to another in each of the exchanges shown in Figure 1, and they are accessible to the plugins in each actor. Property strings can encode arbitrary descriptions in arbitrary languages and formats. But the properties are not interpreted by the leasing core: their meaning is a convention among the controller plugins and handlers.

This use of property lists allows the protocols to accommodate a variety of information exchanged between actors, in keeping with the design principles of guest neutrality, resource neutrality, and policy neutrality. This design choice makes the platform extensible and open to innovation, at the cost of compromising interoperability among plugins that do not "speak the same language".

2.12 Management Interfaces

The leasing core exports *management APIs* to configure, query, and manage actors within a container. These APIs support external management tools such as the external Web portal engine described below. Some kinds of extensions (plugins) may define their own management interfaces (e.g., see Section 8).

For GENI we have contracted to define and implement extensions to these APIs to enable richer and more powerful management tools.

An important part of this effort is to "remote" these interfaces so that external tools can invoke them over a network with appropriate authentication and authorization (e.g., SOAP with WS-Security). Currently only one part of the API may be invoked through SOAP: a simple management interfac for the container itself. More complete support for remote invocation would enable seamless communications and a unified view of actors in multiple containers through a single management portal. These generalized management interfaces can serve other purposes as well (e.g., fault injection, or a GENI meta-authority, such as GMOC facility operations).

For the Automat project, we extended the management framework for *facility level* control. Our original work (SHARP and Shirako/COD) conceived actors as independent entities that control their own resources and destiny. Automat introduces the notion of a *facility* or "center" with an operating authority that owns the resources and delegates them to actors, which may be instantiated dynamically as the granularity of extensible policy control. More concretely, the substrate is viewed as owned by the container and its operator, who can assign and revoke control over partitions of substrate to specific authority actors. Actors can be instantiated dynamically within the container. The purpose in Automat is to enable autonomic computing researchers to experiment with hosting center management policies on a subset of the substrate resources in a facility.

GENI note. For a container with multiple authority actors, this idea of common container-wide substrate and operator is similar to the notion of a *Management Authority* in the SFA proposed for GENI. We treat it as non-essential in GENI discussions because we see no reason for it to be visible to other actors in the control framework: it is an implementation detail of the substrate provider.

2.13 Web Portal

In the current implementation, a Web portal provides a common GUI interface to all of the actors in a single container. The portal is written in the Velocity template scripting language. The Apache Velocity interpreter is written in Java, and Velocity templates may invoke other Java objects in the same container. The portal runs alongside the actors within the container, and the portal templates invoke the local management APIs directly.

Users log into the portal and obtain a GUI for the local actors that they are authorized to control. The different actor roles present different interfaces through different tabs in the portal. Thus the same portal may be used by end users (e.g., experimenters), broker operators, and/or substrate providers.

Containers provide standard "container-managed security" mechanisms for a user to authenticate to the Web portal with a password. For example, standard Web application servers such as Tomcat can accept user principal credentials endorsed by a locally trusted identity provider such as an institution's Kerberos or Shibboleth service. The code distribution comes configured with a local password file with a single root account.

The portal is called "Automat" because many of its features were added to support dynamic actor instantiation and dynamic extensibility needed for the Automat project. The portal allows users to upload and deploy guests and controllers, subject them to programmed test scenarios, and record and display selected measures as the experiment unfolds. The portal also includes a container administrator interface for the root operator (accessed through the the "center" tab that appears for users with operator privileges). The container root has administrative rights over the substrate controlled from the container, and over all actors within the container.

Views and extensibility. The portal itself is extensible. To support interactivity in the Automat testbed, a controller may define an optional *view* as a Web portal plugin, enabling users to monitor and interact with guests and controllers during an experiment. For example, the controller plugins can implement a graphical interface to set attributes or parameters for the guest, e.g., to modulate a workload generator. View plugins typically are server-side Web templates (e.g., in Velocity) that run at the Web portal, which could run in a different JVM from the actor. They may also include applets or other client-side code.

Todo: revisit how templates and views get and display info from resources and relationships among resources. We would like to take a maximally abstract view of resources, and there are some virtual cloud computing assumptions built in here and there.

Authentication and authorization. We plan to extend the ORCA software to: (1) enable actors to obtain Shibboleth-provided security attributes for user identities logging in through the portal, and (2) identify and prototype an appropriate set of user attributes as a basis for authorization decisions that consider the attributes and their sources. We will also plan to enable credential delegations using new support for "proxy authentication" in Shibboleth. The purpose is to allow an actor (e.g., an AM) to validate that a requesting peer (e.g., a slice controller or a portal) is acting on behalf of some user, and to obtain that user's credentials. The user must have authenticated to an IdP that recognizes the requesting peer as a legitimate service and that is trusted by the actor to make assertions about the user.

3 Deployment and Integration

This section gives an overview of deployment considerations, and ways for user tool developers (Section 3.2) and substrate providers (Section 3.3) to integrate with ORCA. Section 3.4 discusses issues relevant to diverse substrates and GENI.

In general, integration involves deploying one or more Shirako actors and selecting, modifying, or developing plugins for those actors. The plugins may include code to glue or bolt other software such as portal interfaces or substrate management services onto the common Shirako leasing engine, which offers common scaffolding for most of the actor internals. Section 4.1 discusses plugins and their interfaces in more detail.

Of course, Orca actors are not required to be built with Shirako, or even built in Java. The protocol interfaces in the GENI deployment are SOAP with WS-Security, with basic interfaces and formats defined in WSDL. In principle, actors could be implemented in a wide range of programming environments. However, Shirako is extensible in ways that would affect external protocol interfaces and wire formats, and we intend to take advantage of that as the project unfolds. In particular, the key protocol messages carry generic property lists with additional directives and descriptions, which may be in languages such as NDL-OWL (or RSpec). This means that development of non-Shirako actors for the GENI-ORCA deployment would require an exercise in nailing down a common data model and implementing it separately in multiple programming environments. The data model we use today is still evolving, and the WSDL does not capture all of the syntax or semantics (e.g., lists of properties and their meanings).

The remainder of this section, and the rest of this document, assume that you are using actors based on the Shirako leasing engine in the Java environment.

For integrators who are programming directly to Shirako plugin APIs, the software provides various tools, XML formats, programmatic interfaces, Web portal interfaces and name spaces to manage plugin elements and register them for upcalls on specific actors, slices, leases, or resource types. Defining easy ways to register and configure plugins and actor relationships has proven to be tricky, and it has not received full attention because it is not a "research topic". This area is one of the steeper parts of the learning curve for team members and integrators.

GENI Note. For GENI, the project is creeping toward a standard set of GENI-friendly plugins that support standard GENI interfaces and languages to attach external substrate providers and experiment control tools. That could remove the need for other GENI integrators to deal directly with the Shirako interfaces or Java programming. It will also provide useful reference plugins for integrators who choose program directly to the Shirako plugin interfaces.

3.1 Actor Deployment

Developers of user tools and guest software, and some advanced users, will deploy one or more slice controller (SM) actors. Substrate providers will deploy one or more authority (AM) actors to interface their substrate inventory to the control plane.

All actors are long-lived entities with persistent state stored in a database. They must run in a Java-capable environment, i.e., with adequate memory and non-broken versions of the JVM and related tools, often installed over the broken versions helpfully included with popular Linux distributions. They act as both clients and servers for other actors in the control plane, so they must establish and accept connections and handle cryptographic keys. In addition, AMs interact with their substrate components, so they must connect to the substrate management plane, which may be a private network. SMs interact with their slivers, so they must connect to the slice data plane, which may be a private network. A surprisingly large share of complications and problems trace back to these basic needs.

In general, users should not have to deal with actor deployment. An institution, lab, or facility provider can host SMs within a hosted slice control service. The user can still extend the functionality by defining or selecting the plugins for an SM. For example, the Automat Web software portal supports dynamic instantiation of programmable slice controllers for experiments in feedback controlled autonomic computing. For GENI context we expect to use a similar approach to host SMs with standard plugins that interface to various off-the-shelf experiment tools running outside of the Java environment. Even so, an advanced user or customer might build their own SM with extensions programmed directly the the plugin APIs, and run it on their own machine for maximum flexibility.

To expand on all that, an actor deployment must consider the following issues:

- **Persistence.** All actors are stateful and must maintain persistent state in a recoverable database. We are currently using a per-container SQL database to meet the storage needs of actors in the container.
- **Continuous operation.** Actors should be continuously active while a slice is operating. If an SM allows its leases to expire, then its resources may be deallocated. If an authority fails, then existing leases will remain active beyond their expiration times, and ticketed guests may be denied access to their resources. If a broker fails, then it may be impossible for guests to renew their tickets, and an aggressive authority may close down unrenewed leases.
- **Connectivity to/from control plane.** The SM must be reachable for external invocations (e.g., via SOAP) from other control plane actors. In particular, the resource control servers (brokers and resource authority servers or AMs) contact the SM to notify it of changes to its resource status, including issued tickets and leases. This could be a problem if the SM is behind a firewall. Resource control servers must be visible to their clients.
- **Connectivity to guest.** The guest slivers must be reachable to the SM, or at least to any external controller that manages the guest. For example, when a new virtual machine is instantiated, the controller might connect to it to launch software. This is problematic if the slivers have private IP addresses or are behind a firewall.
- **Identity and authentication**. All actors must possess suitable keypairs bound to the principals they represent. All actors must know the addresses for other actors they contact, and the public keys of all actors they interact with. The SM must possess a suitable keypair to act on behalf of a user and to authenticate to any sliver interfaces in the slice, e.g., for an ssh login to a VM as root.

There is more to say about how we resolve these issues in specific deployments, e.g., at Duke and RENCI. For example, at Duke, for many years all slices and actors have run on the same VLAN segment, but with different subnets: we run all actors in a special subnet with routes to all the slice subnets, and each node within a slice is installed with a route to the control subnet.

3.2 Integrating User Control Tools

User control tools exist to help users be more productive. Users and/or their guest software use control tools to discover and request resources, launch guests on those resources, and monitor and control the guests. There are three styles of integration for user control tools:

- **Direct integration.** Program the user control tool functionality into an SM actor as a set of plugins.
- **Indirect integration.** Some SM plugins may send messages to invoke interfaces outside of the container, or expose ports and interfaces to tools running outside of the container. To the extent that these interfaces are standardized, we can provide off-the-shelf plugins for integration. For example, the Orca team is currently building a reference slice controller that exports an SFA sliver interface. That would enable tools built for ProtoGENI to obtain resources through an Orca control plane without writing any new code.
- **No integration.** It is important to consider whether any integration is needed for a particular tool. For example, some tools may have elements built into the image that is running on a programmable sliver, and have communication bindings or keys to contact an external tool. Such tools could be independent of the control framework. Common tools such as *ssh* may be used manually if the user knows basic binding information for a sliver (hostname or IP address and key).

One example of a user control tool is a Web portal with a GUI. The Orca software release includes an extensible portal. But it is also possible to integrate a different front-end portal through the SM plugin interfaces, directly or indirectly. For richer portal functionality that includes management of the substrate, remoteable management APIs may be helpful.

Advanced users may prefer to use programmable tools or scriptable command line tools. Such tools may include functionality that is specific to a particular kind of guest, e.g., to instantiate the many pieces of a complex guest and coordinate staging or workflow among them. The Orca slice controller plugin interfaces were designed in part to meet this need, and have been used directly for sequenced staging ant stitching of complex environments such as Globus grid deployments [13]. (See Section 4.6.) But where useful functionality exists outside of Orca, it can be integrated using the indirect approach.

GENI Note. For example, the GENI services WG decided to separate and distinguish the functions of controlling a guest experiment (e.g., launching and monitoring processes, etc.) and controlling the slice itself (e.g., interacting with other control plane actors to allocate, configure, and stitch resources). GENI uses the terms *experiment controller* and *experiment control tools* for software that controls the experiment itself. Sophisticated experiment control tools are evolving for GENI, such as GUSH. In Orca, these tools can be integrated using the indirect approach: they work in tandem with the SM actor, but are separate from it. For example, the GUSH experiment controller is written in C++ and interacts with an SM actor through an XMLRPC protocol interface.

For GENI, it is an open question how to integrate the various portals with functionality specific to different testbeds that might be federated through GENI (*portal mashing*).

3.3 Integrating Substrate

A substrate authority (AM) is responsible for controlling some portion of the substrate and interfacing it to the control plane. We make the following assumptions about the back-end AM implementations:

- **Component control.** Authority/AM actors and their operators have direct or indirect control of the substrate components in their domains. It must have sufficient control to make promises about allocation of the substrate to the broker and to setup and teardown slivers and slices on the substrate. A typical authority would have physical control of the substrate components over a secure and isolated management network.
- **Sliver control.** Each guest obtains exclusive control over each leased sliver for the duration of its lease. Depending on the nature of the sliver, this control may involve interfaces exported by the authority, by an associated component manager or sliver server, and/or by the sliver itself (e.g., secure passwordless root login to a virtual appliance).
- **Sliver isolation.** Any sliver created on a component by the component's controlling authority is isolated from other slivers hosted on the same component, at least to the extent advertised in the lease. For example, a typical expectation is that slivers on the same component inhabit isolated name spaces and are weakly performance-isolated through proportional-share resource scheduling.
- **Sliver termination.** An authority is able to unilaterally terminate slivers that are bound to its substrate components, and/or reclaim resources from them.
- **Slice isolation.** Authorities impose suitable control over the traffic traversing their networks, e.g., through gateways to the external Internet.

Orca enables a substrate provider to outsource identity, resource arbitration and authorization, and calendar scheduling to a broker. A ticket is in essence an endorsement of a principal by the broker and an assertion of specific resource rights granted by the broker according to its arbitration and authorization policies. By delegating resources to the broker, the authority/AM consents to the broker's policies, and agrees to try to honor tickets issued by the broker. If the ticket is valid, then the authority/AM need not concern itself with the real-world identity of the customer.

We emphasize that the authority/AM retains ultimate control over its resources.

- A substrate provider may choose to operate its own broker, and refuse to delegate to anyone else.
- An AM might apply additional authorization checks based on additional knowledge about principals authorized by the broker.
- An AM has full control over mapping/embedding/assignment, i.e., how to assign resources of its physical substrate to slivers ticketed by the broker.
- An AM might expose only a subset of its substrate to the broker, retaining some surplus "headroom" for rolling maintenance or local demands.
- A provider might expose its substrate at a high level of abstraction or with weak assurance, and retain more scheduling control for itself.
- An authority might refuse to honor any ticket issued by the broker for any reason, or break its lease contracts unilaterally while they are active. This behavior is unexpected and unhelpful, but it is within the provider's control.

Although the resource delegation and brokering features of Orca do not force any provider to do anything, the value that any specific deployment gains from these features depends on the ability and willingness of providers to make useful commitments.

An AM needs code to represent its substrate in one or more pools from which slivers may be allocated. The pools are grouped by type, decorated with resource attributes, and exported as delegations to a selected broker. The broker claims the delegation, if it chooses to accept it, and incorporates the pool into its resource inventory. The broker issues tickets for slivers from the pool according to its policies. The AM selects or supplies a `ResourceControl` policy for the resource type to map ticketed slivers onto components in the pool, and *resource handler* scripts to instantiate each slivers on a selected component. If slivers span multiple components, then it may be necessary to supply an alternative `IConcreteSet` representation as well, rather than using existing implementations (e.g., `NodeGroup`) included in the release. Of course, a substrate provider might choose to expose a complex substrate to the AM as a single monolithic component, and handle the mapping internally.

These things can be done by writing plugins, or through a Web portal. If the existing node/group model (see Section 6) works for your substrate, then it may be that all you need are some handler scripts to setup and teardown slivers, and perhaps some node drivers to go with them (see Section 4.1).

3.3.1 Images and Programmability

An Orca system can host a wide range of guests (or experiments). What is needed is to package the guest software as a set of executable images for various programmable slivers in the slice, together with wrappers that can launch the guest and control it. These wrappers include configuration code and a dynamic controller built to the plugin interfaces, or external standard control tools that integrate indirectly with a generic SM.

Packages and appliances. Good mechanisms and formats exist for packaging of user-installable guests. Package managers are now mature. VM images are a convenient vehicle for delivering *virtual appliances* with prepackaged software components for a specific purpose or application (e.g., OVF, rPath, AMI), containing the application together with a snapshot of a file tree and a bootable operating system image, which may be customized to the application. The appliance model is well-suited to virtualized infrastructure, and it can reduce integration costs for software producers and simplify configuration management of software systems. Tools for building and maintaining appliances and images exist outside of Orca and are compatible with it.

Image manager. It is a separate and interesting problem to maintain a collection of shared images or appliances to load into slivers. The Orca software release includes an interface to a replaceable external image manager. Each image and image server (collection) is named by a GUID. The SM may specify the image to run on a per-lease basis; all slivers of the same lease load the same image.

Image staging. Since image staging is substrate-specific, it is the responsibility of plugins to manage image staging and installation of user-supplied images. The DOME testbed allows users to supply images through its portal, and prestages those images to its nodes before any lease specifying those images can launch. In our VM deployments we manage images through a shared file server that supports fast cloning, i.e., NetApp or ZFS. For large clusters, it would be useful to have hierarchical or peer-to-peer staging implemented in node drivers (see Section 4.1.2).

Image functions. The guest may require that certain functions are present on the image. For example, the Keymaster approach [6] for key exchange between a sliver guest and its controller requires a minimal Keymaster program installed on the image. For feedback-controlled slice resizing [12], we have used a Hyperic HQ monitoring agent installed on the image, or post-installed as a package by the slice controller when the sliver instantiates.

Guest packages. A practical Orca system must bridge the gap between the static packaging and dynamic control after a guest is instantiated. We refer to a complete bundle of elements for a guest, including software images and controller elements, as a *guest package*. A practical system should support workable standards and facilities for dynamically installable and upgradeable guest packages. It it is an open question to what degree these control features will be incorporated into industry standards for packaging appliances.

3.4 GENI Discussion

Here is a summary of some high-level issues relevant to GENI integration.

3.4.1 Standardization of Types and Properties

An important challenge is to manage dependencies among composable controller elements and other plugins, and to ensure consistent configurations across the control plane. For example, the set of properties and their meanings is an important part of the specification for any AM or broker, and slice controllers and handlers must know how to use the property sets for the providers they use. Similarly, controllers are responsible for passing the right properties among the handlers and other controller elements they interact with—each policy must know the set of properties required to correctly execute the handlers and drivers for the guest it controls. Controller elements may also make assumptions about what is on the image. Standards are needed, and our approach does not preclude them, but it does not dictate them either. `Query` can help (see Section 4.7), but it is not sufficient.

Name space for resource types. One issue is that there is no standard space of type codes and no standard classification of possible substrate resources into types. This is both a strength and a weakness. It leaves flexibility for a substrate provider to classify their resources at whatever granularity is appropriate, e.g., to partition resources in a data center into sets based on location or slivering method. However, some mechanism is needed to ensure that interchangeable resources are assigned the same type at different providers, or to maintain a standard directory of types, or to prevent collisions in the type name space.

Name space for properties. A related issue is that there is no standard space of properties. This document gives several examples of useful properties that have been implemented. But they are not universally available across all plugin implementations, and the names are not always used consistently.

Interpretation of properties. The value of a property may be a detailed document in some higher-level specification language such as RSpec or NDL-OWL (see Section 4.2). These representation languages are still evolving, and we see their development as one of the most significant long-term tasks (and contributions) for GENI. It is an open question whether these languages should be powerful enough to represent anything that needs to be said, or whether other aspects (such as statements about time) should be factored into other properties. We think it is important to retain more general property lists in the protocols. Standards for the property name space are needed, but no such standards exist.

3.4.2 Broker Allocation Power

A key long-term issue is how to represent complex resources at the "right" level of abstraction for a broker, to balance the needs of effective coordinated allocation with provider autonomy.

For example, in the GENI Spiral 1 demo for BEN substrate, the SM requests "a VLAN embedded in the BEN network" from the broker, and then passes a detailed NDL-OWL specification of desired services and connectivity along with the ticket to the BEN authority. In this approach, the broker does not have a sufficiently detailed view of the substrate to know that all of the tickets it issues are simultaneously satisfiable. Under contention, this solution would force some users to retry rejected tickets. Ideally, the broker could understand the substrate at a sufficient level of detail to schedule the resource efficiently with low probability of rejection.

The challenge is to decouple allocation and arbitration functions from management of the underlying substrate, so that they may run somewhere else. This challenge is not unique to ORCA: it is inherent in the problem of coordinating scheduling of diverse resources,

3.4.3 Limitations of the Resource Model

We frequently discuss various potential weaknesses of the resource model that might be relevant in GENI.

- It is designed for allocatable resources: sets of units of a common type. It is not clear how the model applies to, for example, acquiring rights to install filtering or redirection rules into a shared rule base, as in OpenFlow.
- It is designed for resources with static properties. More precisely, the properties that drive resource selection and arbitration are presumed to be static for the duration of an advertisement/delegation. It is not clear how to select resources based on a highly dynamic property, such as location in a mobile testbed, or density of occupation or use.
- It is designed to manage collections of interchangeable resource units that are independently programmable. It is not clear how well it extends to large numbers of resources that are "one-of-a-kind" by virtue of their location or position in a topology.

In each case, there are ways to fit the resources to the model, but some other model might be a more natural fit.

3.4.4 Simplifications for Early Spirals

The following are simplifications we made for GENI Spirals 1 and 2. These are either temporary shortcuts or hardened ideological positions, depending on context. In other words, we continue to be skeptical that these missing features are really necessary or "right".

No Slice Authority. In the SFA proposal for GENI, slice creation is controlled by privileged entities called Slice Authorities. They are responsible for approving slices, assigning slice GUIDs and names, and binding slices to principal identities and security attributes. We see the functions of the Slice Authority as subsumed by an identity provider that endorses security attributes of the slice owner (experimenter). The SM runs on behalf of the slice owner and has no inherent privilege beyond that role, so it is distinct from the Slice Authority. The SM creates slices: slice creation itself is not a privileged operation, since slices are just a grouping mechanism for leases.

No Management Authority. In GENI, aggregate managers control substrate components on behalf of one or more *Management Authority* entities representing the resource owners. For our purposes, it is necessary to assume only that each AM is suitably empowered to make good on its advertisements. The substrate resources might or might not be under its direct physical control.

No "lone wolf" components. Some GENI leaders believe that individual components should be exposed directly to the control plane in some lightweight way independent of any controlling aggregates. ORCA does not bother with this case. All components are members of aggregates, and each aggregate is under the control of an authority/AM. Rather than viewing aggregate as a special case of component, we view "lone-wolf" components as a special case of aggregates, i.e., an aggregate of one.

No Clearinghouse Federation. We view delegation and brokering as the preferred mechanisms to federate multiple providers into a unified facility, if the providers are willing to make any substantive commitment to the facility at all. We have not defined mechanisms to federate or coordinate independent brokers. Of course, SMs may combine resources from multiple brokers into a single slice. We see Clearinghouse federation as little more than an agreement among brokers to use common identity providers and overlapping authorization policies. But there is no mechanism for federated brokers to co-schedule their resources, unless they delegate to a common broker.

Note that delegation does not require providers to relinquish access or control of their resources, although they must trust their broker(s) to enforce the agreed authorization/arbitration policy, whatever it might be. The policy may include priority for specific user communities at specific providers, pre-negotiated maintenance windows, or other needs commonly cited as reasons for providers to avoid making any commitments beyond best effort. Of course, even that minimal level of commitment can be represented in a useful delegation as well.

No SSL. Actors digitally sign their communications. We believe that state-changing commands or actions must be signed so that actions are non-repudiable and actors can be made *accountable* for their actions. SSL alone is not sufficient. Given that we are concerned with message integrity and authenticity, and not privacy, SSL is not necessary either.

3.4.5 Discussion

Todo: revise and extend. MA and SA are useful, but have only a weak link to the control plane. MA and SA combine the roles of identity provider and accountability service. They endorse a public key and act as a contact point to an administrative entity that is empowered to address problems originating with the principal, and/or coerce or punish the principal. As an accountability service, they might hold identities in escrow.

If necessary, we plan to conform by building a Slice Authority into the broker. The broker would assign the slice GUID, assign attributes derived from the requester's identity, and return a signed certificate to be attached to the slice property list.

Implementation: At each broker, on initial request of a new slice, assign a GUID to the slice, and attach a slice property whose value is an X.509 cert containing the requester's public key and the slice GUID. The SM should use this GUID as the GUID of the slice. The SM passes the slice certification property through on all `ticket` and `redeem` requests for this slice. Brokers and authorities check the certification when they first learn of a slice, and save it with the slice.

4 Plugins and Properties

This section gives an overview of the extension interfaces (plugins), plugin interactions, and the control flow and data flow pertaining to each lease.

4.1 Overview of Plugins

The leasing engine is designed to be neutral to resource types, guest software environments, and resource management policy. In general, all connections to the outside world—substrate, user control tools, portals, etc., pass through plugin extension modules. All inter-actor protocol messages are sent by the leasing core, rather than plugin code, so Shirako integrators are more concerned with the plugin APIs and management APIs than with the protocol messages themselves.

Plugins are registered with the actor core and are upcalled from the leasing engine on various events. There are four plugin interfaces of primary interest to integrators.

- **Controllers.** Each actor invokes a *policy controller* module in response to periodic clock ticks. Clocked controllers can monitor lease status or external conditions and take autonomous action to respond to changes. Shirako provides APIs for policy controllers to iterate collections of leases, and monitor and generate events on leases. Any calendar-based state is encapsulated in the controllers. Controllers may also create threads to receive instrumentation streams and/or commands from an external source.
- **ResourceControl.** At an authority/AM, the mapping (also called binding or embedding) of slivers onto components is controlled by an *assignment* or *resource control* policy. The policy is implemented in a plugin module implementing the `IResourceControl` interface. `ResourceControl` is indexed and selectable by resource type, so requests for slivers of different types may have different policies, even within the same AM.
- **Resource handlers.** The authority/AM actor upcalls a *handler* interface to `setup` and `teardown` each sliver. Resource handlers perform any substrate-specific configuration actions needed to implement slivers. The handler interface includes a `probe` method to poll the current status of a sliver, and `modify` to adjust attributes of a sliver.
- **Guest handlers.** The SM leasing engine upcalls a *handler* interface on each sliver to `join` it to a slice and cleanup before `leave` from the slice. Guest handlers are intended for guest-specific actions such as installing layered software packages or user keys within a sliver, launching experiment tasks, and registering roles and relationships for different slivers in the slice (*contextualization*). Of course, some slivers might not be programmable or user-customizable after setup: such slivers do not need a guest handler.

Strictly speaking, Shirako plugins are Java classes that implement the defined upcall interfaces. These classes may also use various support interfaces exported by the core. Of course, more advanced functionality can be layered above these basic interfaces.

Integrators of *user control tools* should focus on the SM-side controller (slice controller) interface and guest handlers. Any custom control logic in an SM is implemented in slice controller modules that generate requests for resources according to the needs of each guest. Guest handlers serve as notification that a sliver has been instantiated and ready for use, or that it will soon be terminated. As described in Section 3.2, some integrators may be able to use off-the-shelf plugins with standard external interfaces such as the SFA sliver interface.

Integrators of *substrate* should focus on resource handlers and perhaps `IResourceControl` and related interfaces (e.g., `IConcreteSet`). As described in Section 3.3, some integrators may be able to use off-the-shelf plugins with standard external interfaces to setup and teardown slivers on their components.

4.1.1 Handler Invocation and Scripting

The underlying handler machinery is the same for resource handlers in the authority/AM and guest handlers in the SM.

Handlers are registered and selected by resource type. Each handler invocation executes in an independent thread, so handlers may block for slow configuration actions. Handlers are invoked through a class called `Config`, which can invoke an interpreter for a handler scripting language. Current handlers within the source release are writing in the *Ant* scripting language. Other Java-compatible scripting languages (e.g., Groovy or Python) could also be invoked as handlers from the actor's Java core, but we do not yet have shim interfaces for those.

Shirako provides a mechanism to sequence handler invocation by defining predecessor relationships among leases, and a mechanism to "stitch" related leases by passing linkage tokens (e.g., VLAN tags) down the tree (See Section 4.6).

For resource handlers, the current code presumes that each sliver is hosted on a single component; in this case handlers are logically associated with components and registered by component type. This issue is discussed in more detail in Section 6.

Guest handlers can invoke software interfaces *within* a sliver, but they have no access to the underlying components, since they run within the SM rather than the authority.

4.1.2 Drivers

Controllers and handlers often invoke external programs or scripts that run directly on a controlled component. For example, there is a resource handler to setup and teardown virtual machines on host servers running the Xen hypervisor. This handler works by executing scripts of Xen-related commands in the control domain (domain-0) on a named host server. The handler runs within the authority actor JVM, while the control domain is itself a Linux virtual machine running on the host server, which is a component out in the substrate. Thus the handler needs some kind of remote execution facility to launch command scripts on a remote node over the substrate's management network.

The Orca source includes a framework for dynamically installable *driver* packages that run under a *node agent* installed on a remote node. For resource handlers, the "node" is a control domain or management node for one or more components. For guest handlers, the "node" is most likely the sliver itself. In either case the same node agent software can be used: it assumes only that the node has an IP address that is reachable from the actor where its invoking handler is running. It is also necessary to conduct a secure pre-exchange of keys so that the communication is authenticated in both directions.

The node agent and drivers are *optional.* If the remote node has some other useful service interface, then there may be no need for drivers. For example, we can hope that in the future all interesting substrates will provide useful remote management interfaces for a controlling authority may use. Surprisingly few of them offer that today. Handlers are always free to use bare *ssh* or some other remote execution facility.

A *driver* is a packaged set of actions that run under the node agent to perform configuration actions that are specific to a component or guest. The node agent and drivers are written in Java. The node agent exports an interface (currently SOAP/WS-Security) for requests from an authorized actor on the network, whose public key is preregistered with the node agent. The node agent accepts authenticated, authorized requests to install, upgrade, and invoke drivers.

Shirako includes drivers for several types of resources and applications used in our testbed, e.g., Xen hypervisors, volume cloning on local storage and various file servers (Network Appliance filers and Sun ZFS servers).

Drivers must respect various conventions (e.g., idempotence) to ensure robust behavior in failure/recovery scenarios. Drivers are discussed in Section 6.5.

4.2 The Four Kinds of Properties

The various actors drive the behavior of the plugins by means of the property lists passed through the leasing protocols and core. All plugins inhale and exhale property lists. The core coordinates the flow of property lists among the plugins, but does not generate or interpret them.

There are four kinds of properties, which are handled differently within the leasing system:

- *Request properties* describe a request for resources. To request a lease for resources, the slice controller creates a lease object, attaches request properties, and invokes a method to initiate a ticket request. The request properties are passed to the broker's policy controller, which arbitrates among pending requests received by the broker.

- *Resource properties* specify the attributes of resource types. An AM advertises the properties in its initial delegation of a pool of resources to the broker. The broker attaches the resource properties to each ticket drawn from the pool, and may add new resource properties to return with the ticket. The properties are readable to all downstream plugins in the SM and AM.

- *Configuration properties* direct how the authority instantiates or configures resources for a lease. The slice controller sets these properties on the lease object before the lease is ticketed or before it is redeemed or extended. The core passes the configuration properties through to the AM at `redeem` or `extend` time, where they are readable to the AM plugins, including the assignment policy (`IResourceControl`) and resource handlers.

- *Unit properties* define additional attributes for each logical resource unit (sliver) in a lease. AM plugins attach unit properties to each sliver named in the lease. The core transmits them to the SM with each lease and then through to the SM plugins, including the guest handler for each sliver.

To reduce the overhead to transmit, store, and manipulate lease properties, the messaging stubs (proxies) transmit only the property set associated with each protocol operation, and receivers ignore any superfluous property sets. The slice itself also includes a property set inherited as configuration properties by all leases in the slice.

The value of a property may be a detailed document in some higher-level specification language such as RSpec or NDL-OWL. It is useful to note that the property lists correspond to various levels of abstraction that a resource specification language must support. RSpec documents speak of "request RSpec" for a resource request, "ticket RSpec" for a partially materialized request that has been approved, and "manifest" for a fully materialized request, what we call a lease with unit properties for each instantiated sliver. The specification language must also capture the physical substrate. The AM exposes the substrate at its chosen level of abstraction through the resource properties in its advertisement for a resource pool. The broker passes these properties with any ticket drawn from the pool.

In general, we expect that resource properties are static properties of a resource type derived from the AM's advertisement. Actors may cache and reuse these static attributes freely. However, the system needs more flexibility for properties to reflect scheduling or sizing choices made by the broker or authority. To this end, the broker or authority policy may add or modify designated resource properties on a per-ticket basis. These are often called *subtype* properties, and they represent virtual sliver attributes that may take different values on different instances of the type (e.g., sizing attributes for a virtual machine such as the share of CPU power available to it). If the broker leaves them unspecified, then the AM's assignment policy may supply them. For example, a broker may specify that a VM sliver requires a certain allotment of memory and network bandwidth, but leave its CPU allotment undefined. In this case, the authority may allocate the sliver a reserved share of its choosing and change it at any time depending on other needs (a so-called "best-effort" share). No actor may modify resource properties that arrive fully specified from an upstream actor, since they represent (transitively) promises made by that actor.

Section 6 gives some examples of resource properties for a virtual cloud computing system.

4.3 Lease Objects

Shirako actors retain and cache lease state in memory. Each actor has a local lease object to represent each lease. There are different lease classes with interfaces tailored to the different actor roles. *Note:* on the SM side, the view of lease objects goes through classes implementing the interfaces `IClientReservation` and descendents, and in particular the class `CodReservation` and descendents.

There are a few state elements that are common to all leases. These are defined for direct use by the core rather than incorporated into the properties:

- **State.** Each lease object has local state variables that represent the actor's view of the lease state. At any given time, a lease is in one of a fixed set of discrete states defined by a finite state machine (FSM). The lease state machines govern all functions of the leasing engine.

- **Term.** The interval of time over which the lease is valid. In practice there are multiple term variables used during negotiation. When a slice controller formulates a resource request, the term indicates the preferred start time and duration of the lease. The start time may be in the future for an advance reservation; the lease's term can also be modified to alter the start time, and duration (e.g., on lease extension) before the next request or before a ticket is granted.

- **Type.** Resources are typed, and each lease pertains to resources of a single type. An actor may query the space of available resource types on a broker. The resource type does not change during the lifetime of a lease. The type field selects the downstream plugins that configure resources for the lease.

- **Unit count.** Each lease pertains to a block of units (slivers) with identical resource attributes, e.g., a cluster of virtual machines with identical size.

4.4 Lease Negotiation and Lifecycle

Expiration and unilateral reclaim ("abort"). Leases provide a mutually agreed upon termination time that mitigates the need for an explicit notification from the authority to the SM at termination time. As long as actors loosely synchronize their clocks, an SM need only invoke its `leave` handlers for each lease prior to termination time.

An authority may unilaterally destroy resources at the end of a lease to make them available for reallocation if the SM does not renew the lease by presenting a fresh ticket from the broker in an `extendLease`. The decision of when to close expired leases is a policy choice. An authority is free to implement an aggressive policy or a relaxed policy that reclaims resources only when another guest requests them, or any other policy along that continuum.

An authority is empowered to unilaterally close a lease on its resources at any time. However, closing a lease prior to expiration amounts to breaking a promise that the AM made to the SM holding the lease and to the broker that issued the ticket.

Renew/extend. The SM may request to extend/renew leases before they expire. Support for lease renewals helps to maintain the continuity of resource assignments when both parties agree to extend the original contract. Extends also free the holder from the risk of a forced migration to a new resource assignment—assuming the renew request is honored. For extend/renewal operations, servers may reuse cached configuration properties, bypassing the cost to retransmit them or process them at the receiver. The protocol to extend a lease involves the same pattern of exchanges as to initiate a new lease (see Figure 1). The core obtains a new ticket from the broker, and marks the request as extending an existing ticket named by a unique ID.

Automated meter feeding. The SM core renews any lease automatically at the appropriate time if it is marked as *renewable*. Each automatic extend requests the broker and AM to extend the contract for the agreed duration of the previous lease interval. The slice controller may register an event handler to receive an upcall before the SM core issues an `extendTicket` call to the broker. The controller may then choose to modify request attributes, such as the lease term, or even mark the request as nonrenewable so that its term is not extended.

Request Properties: passed in a ticket request	
`elasticSize`	Broker may allocate less than the requested resource units
`min/max`	Defines constraints on the requested units, size, or time (e.g., `elasticSize.min` prevents the broker from allocating too few resources)
`deferrable`	Broker may choose a later start time than requested
`elasticTime`	Broker may approve a shorter term than requested
`atomicId`	ID for an atomic request group—set on all requests that are members of a request group for broker coallocation
`atomicCount`	Count of number of requests in this atomic request group.

Table 1: Example properties for a ticket request to a broker, which may be considered at the discretion of the broker's policy controller.

Flex. A slice controller may request changes to a lease at renewal time by flexing designated subtype request properties or the number of resource units. For extends that shrink the number of units it is desirable to provide some mechanism for the slice controller to select the specific victim slivers to relinquish, as discussed below.

Close/Vacate. Slice controllers may require a way to vacate their leases and release their resources for use by other guests. The SM or authority may `close` a lease at any time, which releases the resources at the authority/AM.

GENI note: We have contracted to add support for a slice controller to `vacate` a closed lease, which would cancel the issued ticket and release the resources at the broker. The proposed solution is secure, but it relies on the slice controller to initiate. Of course, if the slice controller fails, then the resources are lost to the broker for the term of the lease, even if the authority initiates a close.

Intent to renew. It would be useful to have some way to negotiate for a broker to promise a good faith intent to renew the lease over some specified time interval. In essence, this approach decouples the expected lifetime of the contract from the renewal term, which enables the system to reclaim orphaned resources for use by another consumer if the holder fails (similar to garbage collection in .NET, RMI, and other distributed object systems derived from SRC Network Objects).

4.5 Request Property Examples

Request properties. Table 1 lists some example request properties. These are common conventions that should be supported by a range of controllers. *Todo: do it.* These examples provide simple ways for the slice controllers to specify the degree of flexibility that the broker has to fill the request at a time and manner of its choosing. The request properties may also specify constraints on the request. For example, a slice controller could combine requests into a single *request group* to indicate to the broker that it must fill the requests *atomically* (co-schedule them). To define a request group, a consumer sets an ID on each request (`atomicId` in Table 1). To ensure that requests are not allocated until they all arrive at a broker, a consumer may also set a count for the number of requests that are associated with that ID. The broker fills all the requests as a unit when it has received them all. Properties to guide colocation are also desirable, but we have not defined any.

4.6 Lease Groups and Stitching

Shirako provides a grouping primitive to sequence guest handler invocations. This is useful for complex guests with configuration dependencies among their components. Since the slice controller specifies properties on a per-lease basis, it is useful to obtain separate leases to support development of guests that require a diversity of resources and configurations. Configuration dependencies among leases may impose a partial order on configuration actions—either within the AM (*setup*) or within the SM (*join*), or both.

The dependencies are represented by a restricted form of DAG: each lease has at most one *redeem predecessor* and at most one *join predecessor*.

- If a redeem predecessor exists for a ticketed lease object and the SM has not yet received a lease for the predecessor, then it transitions the request into a blocked state, and does not redeem the ticket until the predecessor lease arrives, indicating that its `setup` is complete.
- If a join predecessor exists for a newly issued lease, the service manager holds the lease in a blocked state and does not fire its *join* handler until the join predecessor is active.

In both cases, the controller may register for an upcall from the core before transitioning out of the blocked state and firing the handler. The upcall gives the controller an opportunity to manipulate properties on the lease before the handler fires, or to impose more complex trigger conditions. For example, the controller may query properties of the predecessor and modify properties of the successor.

These features enable the slice controller to impose a partial order on configuration actions, and propagate information among related leases. For example, the GridEngine guest example (SGE) instantiates one VM as a cluster master and another as a central file server for an SGE job service. It instantiates these services first, then instantiates a dynamic set of worker nodes, passing the IP addresses of the master and file server to the workers as they initialize. The IP addresses are retrieved from the unit properties of the predecessor leases.

These features are also useful for "stitching" slivers in the slice data plane. For example, the July 2009 ORCA-BEN demo stood up a complex slice with VMs at multiple sites linked by a VLAN over the BEN network. It was necessary to instantiate the VLAN before attaching the VMs to it. The VLAN tag was returned as a unit property in the VLAN lease, and then transferred as a configuration property to successor leases for the VMs.

We have determined that it is awkward to shoehorn complex configuration scenarios into the current "single pass" configuration model for the ORCA slice controller. For GENI, we have contracted to enable more powerful slice controllers by enhancing these features to orchestrate end-to-end slice configuration, including multi-pass cross-domain stitching. That requires extending the sequencing graph on the slice controller, adding new pre-redeem `link` or post-redeem `configure` methods to the lease interface on the authority/AM. In addition, we need to validate linkage labels that are passed across aggregate boundaries, e.g., by signing them.

Query Property	Description
`inventory`	Snapshot of the current resources under the broker's control so that the guest controller knows what types of resources it may request and if those resources will satisfy its requirements. The broker can choose the level of transparency it wishes to disclose about its resources; exposing more information allows the guest controller to better formulate its requests.
`partitionable`	Describes if the broker allows requests for resource slivers (e.g., virtual machines with specified CPU, memory, and bandwidth).
`callInterval`	The length of time between allocations on the broker. Depending on the broker's allocation policy, it may only perform them at regular intervals (e.g., scheduled auctions).
`advanceTime`	Defines how early a request needs to arrive at a broker to ensure that the broker will satisfy that request for a given start time. For example, a broker may run an auction for a specific start time a day in advance.

Table 2: Example `query` properties for brokers.

4.7 Queries

The Orca actor interface defines a generic `query` method. Actors use queries to learn about some other actor, the resources it manages, the services it provides, and/or the plugins it is using. The `query` takes a property list as input and generates a property list as output. It is the responsibility of the actor's policy controller to respond to queries.

The `query` is a basic hook for resource discovery. As an example, before making a request to a broker, an SM may wish to know more about the broker's inventory or policy (e.g., resource types and availability or how frequently the broker allocates resources). To formulate a request, a service manager may also query the request properties the broker's policy accepts and understands; for example, some brokers may understand an `elasticSize` property, whereas others may not. Table 2 provides examples of queries an SM may make to a broker to gain information about its policy.

4.8 Packaging and Configuration

Todo: revise and extend. Controllers, views, and guests are packaged in *extension packages*, which are uploadable through the portal. An extension package is an archive consisting of Java libraries, presentation templates (e.g., Velocity scripts), guest installation files, etc. The contents of each extension package are described in a package descriptor file, which enumerates the controllers and guests in the package. The system validates the package and registers all controller modules listed in the descriptor file, making their classes available through a custom classloader. The portal keeps track of registered controller modules and offers menus to instantiate and configure controllers. Extension packages may use classes defined in previously loaded packages.

5 Shirako Internals and the Lease State Machines

This section outlines the implementation of the leasing engine.

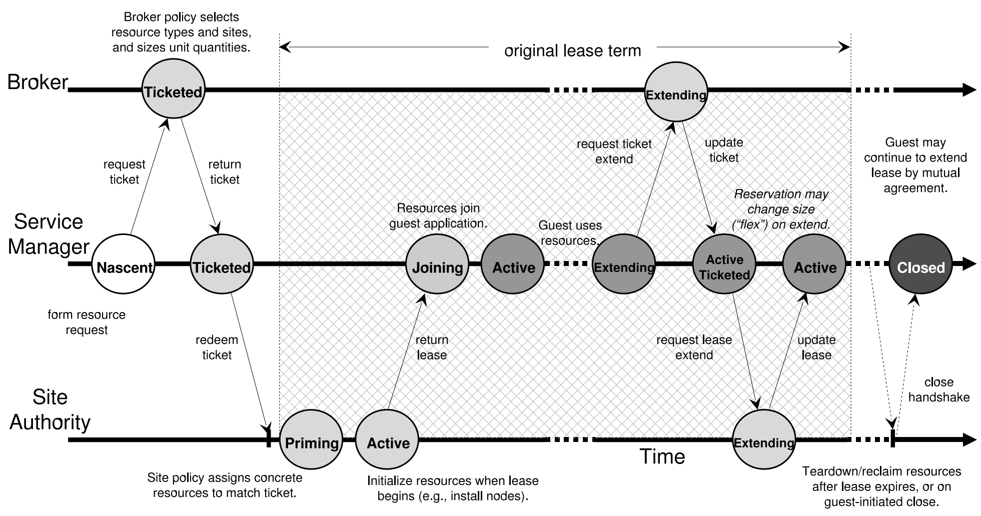
Broker policy selects resource types and sites, and sizes unit quantities.
original lease term
Broker
Ticketed
Extending
request ticket
return ticket
request ticket extend
update ticket
Reservation may change size ("flex") on extend.
Guest may continue to extend lease by mutual agreement.
Resources join guest application.
Guest uses resources.
Service Manager
Nascent
Ticketed
Joining
Active
Extending
Active Ticketed
Active
Closed
form resource request
redeem ticket
return lease
request lease extend
update lease
close handshake
Site Authority
Priming
Active
Time
Extending
Site policy assigns concrete resources to match ticket.
Initialize resources when lease begins (e.g., install nodes).
Teardown/reclaim resources after lease expires, or on guest-initiated close.

Figure 2: Interacting lease state machines across three actors. A lease progresses through an ordered sequence of states until it is active; the rate of progress may be limited by delays imposed in the policy modules or by latencies to configure resources. Failures lead to retries or to error states reported back to the service manager. Once the lease is active, the service manager may initiate transitions through a cycle of states to extend the lease. Termination involves a close handshake.

The state for a brokered lease spans three interacting state machines, one in each of the three actors involved in the lease: the SM that requests the resources, the broker that approves the request, and the authority that assigns and configures them. Each actor maintains a local lease object representing the contract throughout its lifetime, indexed by the contract's GUID.

Each lease object behaves as a finite state machine (FSM). The FSMs transition in response to timing events, arriving inter-actor messages such as requests or lease/ticket updates, clock ticks, changes in resource status or demand, and actions by plugin modules controlling resource management policy for each actor. The state transitions in each local FSM are driven by the logic of its specific role (SM, broker, authority). Figure 2 illustrates state transitions for a resource lease lifecycle.

5.1 Time

Some state transitions are triggered by timer events, since leases activate and expire at specified times. For instance, a slice controller may schedule to shutdown the guest before the end of the lease. Because of the importance of time in the lease management, actor clocks should be loosely synchronized using a time service such as NTP. While the state machines are robust to timing errors, unsynchronized clocks can lead to anomalies from the perspective of one or more actors: requests for leases at a given start time may be rejected because they arrive too late, or they may activate later than expected, or expire earlier than expected. One drawback of leases is that managers may "cheat" by manipulating their clocks; accountable clock synchronization is an open problem.

When control of a resource passes from one lease to another, we charge setup time to the controlling lease, and teardown time to the successor. Each holder is compensated fairly for the charge because it does not pay its own teardown costs, and teardown delays are bounded. This design choice greatly simplifies policy: brokers may allocate each resource to contiguous lease terms, with no need to "mind the gap" and account for transfer costs. Similarly, service managers are free to vacate their leases just before expiration without concern for the authority-side teardown time. Of course, each guest is still responsible for completing its *leave* operations before the lease expires: the authority is empowered to unilaterally initiate *teardown* whether the guest is ready or not.

Actors are clocked externally to eliminate any dependency on absolute time. Time-related state transitions are driven by an internal clock that advances in response to external `tick` calls.

5.2 The Kernel

The package `orca.shirako.kernel` contains the leasing core. It consists of a set of classes that are instantiated within each actor. Requests enter the actor core through a `KernelWrapper` object, which validates the request (see below) and then invokes the actor's `Kernel` object to execute the request. The kernel is also invoked internally to handle clock ticks and request acknowledgements.

The kernel polls a list of lease objects for state changes in response to clock ticks. In particular, it issues `probe` calls to resynchronize the state of the internal lease object to reflect any changes to the resources bound to the lease. *Note:* the kernel's queue of lease objects to poll currently includes all lease objects. Actors really should receive a larger set of notifications ("interrupts") to inform the actor of changes to the underlying resources. Only leases with pending notifications should be placed on the queue, to reduce polling overhead.

The actor `Kernel` is common to all actor roles. Fundamentally, the kernel maintains a set of slices; each slice contains a set of lease objects (referred to throughout the code as *reservations*); each lease has an associated `ResourceSet`. Each reservation has an independent state machine (FSM). All events go through the kernel, which drives FSM transitions, logs all state changes needed for recoverability, and upcalls plugin modules to make policy decisions.

Internally, the kernel invokes several abstract plugin interfaces, each of which has multiple implementations specific to the actor role and/or the nature of the resources being managed. Integrators would typically not use these interfaces directly, but all integrator-supplied plugin functionality fits underneath them somewhere, and in principle they are all replaceable. The key interfaces invoked by the kernel are:

- `IPolicy` represents the actor's policy controller, which is responsible for mapping resources onto leases (sometimes called "mapper"). In the resource servers (broker and authority roles) the policy has an obvious provisioning function (arbitration and assignment). The same upcalls are used on the SM side to enable a policy to, for example, bid from a local budget of virtual currency for each outgoing request. The SM-side hooks for a cyberinfrastructure economy are not used by any plugins currently in the pool, but the terminology derived from that orientation (e.g., bidding, auctions) are present at many points in the code.
- `IKernelReservation` is the kernel's interface to a lease object.
- `ShirakoPlugin` is an uber-plugin that controls the underlying resource representations, including replaceable modules for the actor's messaging stubs and interface to the actor's database.

Todo: Need a fig to map these plugins and how control flows from the kernel.

5.3 Lease Objects and Reservation Classes

Some parts of the core are specific to a given actor role. In particular, there is a specific lease state machine class for each actor role in a given lease. These lease objects or "reservation objects" implement `IKernelReservation`, which is all the kernel knows about them, but also have other interfaces for use by the actor policy controllers and other code.

Perhaps confusingly, a single actor may maintain different leases with different roles. This is because resource inventory pools and delegations/advertisements are represented internally as slices and leases. For example, a broker acts as a client for its delegations from an authority: these delegations are in fact tickets that the broker may subdivide, and they are represented internally as tickets and may be renewed/extended like other tickets. The `ReservationClient` class that encodes a ticket-holder's FSM is common between the SM and broker roles.

Todo: More discussion of the reservation class structure.

5.4 Kernel Entry and Exit

Invocations of the kernel are (in general) nonblocking. They may change the state of an object, and they may queue work for the receiver to do (e.g., please assign resources to this lease request). In general, they return immediately after a state change. The requesting actor receives a callback when the requested operation completes.

All incoming requests pass through `KernelWrapper`. Incoming requests from other actors enter the wrapper from the messaging stubs. The stubs validate the signature on the message and retrieve any locally cached information about the principal that originated the message, which is wrapped in an `AuthToken`.

Any operations initiated by the actor's controller or "main" also pass through `KernelWrapper` indirectly, e.g., injection of new lease objects representing fresh requests initiated on the SM.

`KernelWrapper` acts as a reference monitor for incoming requests. It applies authorization policy by upcalling an authorization module called `AccessMonitor` for each incoming request. The subject principal is abstracted as an `AuthToken` attached to the request. The target object is identified and retrieved, and an associated `Guard` is obtained from the object. `AccessMonitor` is called with the `AuthToken`, `Guard`, and a code indicating the nature of the operation. The `AccessMonitor` and associated classes is the place to install authorization policy (subdirectory `auth` in the source pool).

Note. `AccessMonitor` needs more work to implement the authorization policy in 2.6.

Note. The wrapper needs a generalized hook for other calls initiated by the slice controller on leases or slivers in the **active** state. Such calls don't need to synchronized with the lease state machine: they just pass through to the handler. E.g., on the AM side: sliver reset, or the post-redeem `configure` operations for two-pass stitching.

5.5 Synchronization

The kernel maintains a core *kernel lock* for each actor. The lock is also called the *manager lock* or *core lock* in some comments. It controls all core data structures, including the lease FSMs.

The kernel lock is obviously a potential bottleneck for multicore actors and containers. Note, however, that there are no locks shared among actors hosted on the shared container, and multiple actors may run concurrently within the container:

- The substrate resources in an aggregate can be partitioned among an arbitrary number of authority actors. The tradeoff is that this deployment choice restricts assignment choices within each authority, and pushes more of the assignment burden off to the broker.
- Multiple broker actors may run within a clearinghouse service. The tradeoff is that this deployment choice precludes a broker policy from coscheduling resources owned by different brokers.
- Different end-users on a portal or an SM hosting service may instantiate different slice controllers on the service. The security model was designed for this mode. Currently: each actor operates on behalf of a single principal.

To prevent bottlenecks, use of the kernel lock is tightly controlled. The kernel lock should not be acquired outside of `Kernel`.

A thread holding the kernel lock should never block or attempt to acquire some other lock. Lease (reservation) objects also have locks, and there are a few instances in which a lease lock is acquired while the kernel lock is held. The lease lock is primarily used to synchronize certain FSM state transitions with any local controller threads that are registering to be notified of events on the lease. It is a low-conflict lock, and threads do not block while this lock is held. `IKernelReservation` implements several methods with `service` prefixes: these methods drive blocking configuration actions on the underlying `ResourceSet` and `ConcreteSet` and are called with no locks held.

Synchronization for plugin modules is a delicate balancing act. The original principle was that the kernel lock would never be held while running code in any extension (plugin). However, that choice limits extensibility for functions that are tightly integrated with the core. In particular, lease event handlers (see Section 5.7) are synchronized with lease state changes by the core and should never block for any reason.

Synchronization in the current code is vulnerable in two key respects:

- **Plugins.** The kernel lock is held across upcalls to the actor's policy controller. This choice simplified plugin synchronization considerably and squashed several annoying bugs. In essence, it means that actors are single-threaded, except for the slice controller plugin, database transactions, request validation, and operations on the underlying slivers or components (`IConcreteSet` and handler invocations). We must revisit this choice if we want to support scalable actors on multicore.
- **RPC calls.** The kernel lock is held across RPC requests to other actors. The kernel was designed for asynchronous messaging. We need to add some code to queue outgoing messages and return quickly, and propagate any error responses or exceptions from the peer actor back to the lease object on which the request was issued. This problem is a bug: we suspect that it can cause actor deadlock. Also, Rampart/Axis probably chews most of the CPU.

Todo: Discuss locking in the resource set classes, and plugins invoked without locks from the core. Other locks internal to COD: SubnetBlock, Machine, net/IPList. Calendar classes and policies in `orca.core.policy` also have some internal locking. There is a separate actor lock that is used for configuration.

5.6 Resource Sets

Lease objects hold an attached `ResourceSet` describing the logical resources for the lease. The set represents the resource type, the number of units requested, and the number of units present. The `ResourceSet` holds a reference to an object representing the underlying resources, which it invokes through the `IConcreteSet` interface. Classes that implement that interface include `Ticket` and `NodeGroup` (see Section 6).

The `ResourceSet` interface is useful to resource management policy code. Resource sets may be split and merged. One resource set may be extracted from another. Allocation policies can examine a `ResourceSet` to determine how many units of the resource are needed to fill a *deficit* between the requested and allocated units.

`ResourceSet` methods are grouped into several sets. Methods with the `prepare` prefix are "pre-op" methods. Methods with the `service` prefix are "post-op" methods. These methods filter down to drive configuration operations on the underlying `ConcreteSet`. These methods should not hold any higher-level locks, e.g., the kernel lock or reservation lock. Concrete sets are responsible for their own threading and synchronization.

Todo: There is some funky locking in `ResourceSet` and issues with order-dependence of concurrent operations.

5.7 Lease Event Handlers

Slice controllers may register for various notifications from the leasing engine before or after state changes on specific leases. The event upcalls give the controller an opportunity to manipulate the lease and/or redirect the state machine.

To notify the controller of these events, the Shirako toolkit supports an `EventHandler` upcall interface (not shown in Figure 5). `EventHandler` is supported on the SM core state machines, but not on the other actors. It is intended for use by the slice controller.

For example, before a ticket is redeemed at the AM, the slice controller may attach additional configuration properties to indicate how its resources should be configured (e.g., VM image location).

Another use of the `EventHandler` is for automated extensions of a `renewable` lease. The controller may register to be notified prior to each extension, so that it may have an opportunity to cancel the extension if it judges that the resource is no longer needed. A controller can also drive such choices by continuous monitoring from its clock handler, but some controller writers have found the event-driven approach to be convenient.

`EventHandler` is distinct from the guest handler interface. In particular, it is registered and invoked per-lease, while the guest handlers and resource handlers are invoked per-sliver. Also, `EventHandler` enables upcalls to the controller code on a wider range of events. Table 3 lists examples of how SMs use `EventHandler`.

Function	*Description*	*Example*
onExtendTicket	Invoked before issuing a request to extend a ticket.	A guest may inspect its resource usage and decide if it needs to increase the leased units or select victims and shrink units.
onActiveLease	Invoked when all resources in a lease are active.	The plugin may issue a directive to launch an activity in the guest.
onCloseLease	Invoked before the guest leave handlers execute.	Download and store data from one or more leased machines.

Table 3: The leasing core upcalls the guest lease handler interface to notify controller plugins of lease state changes. The plugin may modify the lease attributes before returning.

`EventHandler` is tightly integrated with the core. These upcalls hold the kernel lock, which makes them dangerous.

5.8 ShirakoPlugin

The resource-specific and substrate-specific aspects of the leasing system are factored behind a major internal extension interface called `IShirakoPlugin`. Each actor can have at most one instance of this plugin. A base class is available in `plugins/ShirakoPlugin`. When an actor is launched an instance of this plugin class, or a subclass, is created for the actor from the boot `ConfigurationProcessor`. The plugin class is selected from a configuration file or other inputs read by the actor boot code.

`ShirakoPlugin` acts as a registry for various internal objects, and the leasing core invokes this registry to obtain references to these objects. The purpose is to enable the plugin to interpose on or extend the functions of these internal objects. The plugin can extend internal classes that represent slices used by or known to the actor, and resource set representations for each lease (`IConcreteSet`), including ticket representations. It can also extend the keystore and internal logging functions. Each of these classes and interfaces is an independent extension point. For example, to extend the keystore functions used in an actor, one could install new keystore classes and add them to the `ShirakoPlugin` object.

We should think of a specific plugin module as consisting of one or more `ShirakoPlugin` subclasses, together with any other classes or other extensions of internal objects instantiated or selected by the plugin. As discussed for the COD example in Section 6, the plugin includes resource representation classes (implementations of `IConcreteSet`). Since it includes those, it also includes communication proxies, ticket and certificate validation code (e.g., `RemoteCallCertificatePolicy`), and translation functions for resource representations as they move from one actor to another. It also defines configuration handlers for the resources supported by the plugin. It can also interpose on or control the representations of resources in a back-end database.

Figure 3 summarizes the interface for the plugin object itself. Beyond support for interposing on other parts of actor functionality, the basic operations are to create and release slices. Most activity in the plugin module (e.g., lease setup and teardown) is driven from classes associated with the resource representation, i.e., implementations of `IConcreteSet`. See Section 5.6.

The plugin is central to the recovery architecture. On recovery, it gets `revisit` upcalls for each slice and reservation in the actor's database, and a call to restart any configuration actions after internal state recovery is complete. It is presumed that the `ShirakoPlugin` itself can serialize/recover to/from a property list. See Section 5.12.

`ShirakoPlugin` must be able to release any resources bound to a slice (`releaseSlice`, even if stuff is stuck or in the process of instantiating. The plugin is responsible for its own synchronization.

To understand the ShirakoPlugin, it is useful to study a specific example: see Section 6.

5.8.1 Issues With ShirakoPlugin Interface

Todo: Limit knowledge of ShirakoPlugin within the leasing core. It should not be exposed outside of the kernel or its call wrapper. Why does PeerRegistry know it?

ShirakoPlugin may combine too many different pieces, some of which are mostly independent of resource or substrate (e.g., elements of security code). We may want to reconsider factoring some of this out.

On the SM and broker, the `ShirakoPlugin` should be as substrate-independent as it can be, since a slice controller (service manager) may combine resources from multiple substrates.

In general, we need these resource representations to be switched by resource type. The `ShirakoPlugin` is powerful enough to do this, via the `ConcreteSetFactory`, but we use a fixed set of representations in COD.

5.9 Controller Structure, Views, and Management API

Controllers must be dynamically installable and they must be manageable through the Automat Web portal and other configuration machinery that is (stricly speaking) outside of Shirako.

A controller has:

- A controller object. Most of the lines are in the controller object.
- A manager object. Stuff you can do to it to configure the controller from outside (e.g., from the web portal).
- A management proxy interface (just an invocation interface).
- Management proxy implementations, e.g., local, for local invocation from a locally running portal.
- Portal plugin (view).
- A *factory* to create and register instances of the controller and link their parts together.

Things that can be installed as extensions through the Web portal and managed through the management layer are instances of `ManagerObject`. The package is `orca.manage.extensions.api` in `./manage/extensions/api/src/main/java/orca/manage/extensions/api`.

Views interact with the controller using the management proxy. A management proxy resides inside the JVM hosting the controller and its actor.

Todo. Proxies enable the portal to communicate with controllers running on remote actors (e.g., SOAP, XMLRPC).

5.10 Calendars

Todo: Say more about the calendar classes in the plugin policy modules, including the base classes and samples in `orca.core.policy`, where much of the code resides.

The `AuthorityPolicy` is mostly policy-free. All it does is call `IResourceControl.assign` in a specific order to satisfy flexing reservations if the resource is constrained: shrinking leases first, then growing leases. It relies on the broker to handle time scheduling and set satisfiable start and end times and resource amounts for each ticket. Thus it does not have to handle advance reservations explicitly, and it does not have to know what the state of the resource will be in the future. This means that the assignment policy itself (`IResourceControl`) is really the important policy extension point in an authority/AM.

5.11 Secure Communication

Actors communicate using an asynchronous peer-to-peer messaging model through a replaceable stub layer. Multiple actors may inhabit the same container and interact through local procedure calls. For cross-container calls, authentication and authorization are based on digitally signed messages (WS-Security) and the Java Cryptography Architecture (e.g., keystore files). The implementation stores the key registry in a Java keystore file (jks) for the actor. The keystore is encrypted with a password; a common password is hardwired into the source code to enable programmatic use.

An actor's keypair and key registry comprise its *security configuration*. Most aspects of security configuration are fully automated, but there are also manual tools available for a container owner to generate, view, and edit security configurations.

The actor's GUID and keypair are typically generated automatically when the actor is created. The code that instantiates an actor and binds it to a GUID and keypair is part of the *boot* code. The boot code runs to populate a freshly instantiated container with actors specified in a container configuration file. The same code also runs when an actor is created dynamically, e.g., through the Web portal.

For more control, a container owner may use the tools to generate the GUID and security configuration by hand in advance. In this case the configuration file or portal request may specify a GUID. [Note/Q: is it possible to hardwire an actor's public key, and install its private key in its security configuration.]

The security configuration files reside in the *runtime directory* of the actor's container. The file names are derived from the actor's GUID, so the files persist and can be retrieved by GUID.

5.11.1 Rampart/Axis

The current implementation uses Rampart/Axis2 for authenticated communication over SOAP. An actor's security configuration also includes an Axis2 configuration file that determines which signing keys to use for SOAP messages sent by that actor.

Rampart assumes use of a PKI and is awkward to use with a decentralized trust management architecture, as in Orca. The Rampart code is based on certificate manipulation, rather than public key manipulation. Public keys are certified by certificates, and there is no means to register a public key for access without an accompanying certificate. This makes it difficult for actors to endorse keys, and for other actors to assign trust on the basis of such endorsements. Rampart presumes that certificates are endorsed by designated certifying authorities within a PKI hierarchy. The current Rampart-based Orca implementation uses self-signed certificates, and thus is insecure.

5.11.2 Actor Identity and the Portal

Todo. [Write about the binding betwen a user logged in on the portal and an AuthToken.]

How does an actor in the portal run on behalf of an external principal, when it does not have that principal's private key? Within the Web portal, when an authorized user creates an actor, mint a keypair for the actor, and create an X.509 certificate signed by the container and endorsing the public key of the new actor as bound to the principal of the requesting user. This enables the container to act as an identity provider. As a first cut, all users will issue their requests through Web portals that are bound to containers that are trusted as registered identity providers by the Clearinghouse (broker).

5.12 Recovery

System crashes, misbehaving extensions, and infrastructure failures may result in contract violations, and our goal is to design a robust resource leasing system that can effectively deal with such problems. In particular, the leasing system must be able to handle transient failures of actors or other elements in the leasing system with minimal interruption of service for lease holders. In this context, these techniques complement the standard mechanism to reclaim resources unilaterally on lease expiration, which is the "last line of defense" to protect availability of resources if a lease holder fails or communication is interrupted for an extended period [9].

Shirako handles recovery by restoring the individual state machines first, then repeating the last FSM transition to restart any configuration actions that were in progress. In particular, leases may transition and recover their state independently of other leases. This property simplifies the state machines and allows for higher concurrency and independent recovery of leases managed by the same server. This approach extends naturally to distributed recovery of the cooperating state machines, if the FSM implementations follow some rules. The hard part is recovering the plugin state. We represent user-supplied extension modules—policy controllers and configuration modules for particular resource types and guest applications—as separate FSMs interacting through well-defined interfaces and protocols.

Upon restarting after a crash, each actor obtains a list of leases from its local data store and rebuilds the lease states and local data. Since lease objects are small, the leasing core commits the entire state of the lease on each state transition. Our current prototype uses a MySQL database for each actor. The committed state includes resource attributes and other data describing the state of the resources. In essence, the commit is a *checkpoint* of the lease state. Once the state is rebuilt, the recovery sequence executes a “recovery transition” for each lease FSM to restart any activities associated with its current state.

These restarted tasks will restart any interrupted workflow by reissuing any messages sent from the state before the failure. Lease FSMs maintain a send and receive sequence number for each neighbor they interact with (there are at most two of them). Send sequence numbers are incremented on message transmission, and receive sequence numbers are incremented on message receipt. Importantly, the state machines are designed so that there is at most one message to a given lease state machine per local state transition. This property, together with committing sequence numbers to the data store, ensures that during recovery retransmitted messages are assigned the same sequence number that was assigned to the original message. It also ensures that messages are delivered in order even across restarts of the transport session.

Before accepting an incoming message, the receiving FSM examines the message sequence number to detect duplicate messages. Messages with the same sequence number are identified as duplicates. A duplicate message may be blocked, if the state machine is still processing the original message. In any case, the result is returned, whether or not the operation is reexecuted.

Policy controllers may attach arbitrary local data to the lease objects as property lists, so that they are saved and recovered on lease state transitions. As part of the initial recovery sequence, each policy controller is informed about every recovered lease. This gives the policy module an opportunity to rebuild its local data from attributes attached to each lease object. For example, broker policies must determine what resources from the inventory are available for allocation, what resources are currently allocated, what requests are pending, and when allocated resources will expire.

Since the controller’s local data pertaining to each lease is committed only on the next transition of the parent FSM for that lease, decisions made by the policy are atomically durable with respect to each lease. That is, if the policy decision was recorded in a state transition of the parent lease FSM, then it was recovered in its entirety. In this case, the parent restarts any task actions resulting from the policy choice: for example, a broker may reissue a ticket to a waiting service manager, or an authority may restart configuration of a specific resource unit that has been allocated and assigned to a request. On the other hand, if the policy decision was incomplete or had not been recorded in a parent state transition when the actor failed, then the prior lease data is recovered, indicating to the policy controller that a decision is still pending (e.g., an open request has not yet been filled).

The task restart approach places several requirements on the operation of the state machines and associated tasks, such as configuration handlers (guest handler and resource handler plugins, and any driver actions they invoke). In particular, tasks must be:

- **Restartable.** To ensure that the restarted tasks behave as expected, each log contains all information necessary to recover any data used by those tasks.
- **Idempotent.** Since a task may have completed some or all of its actions before the failure, actions taken by tasks must be *idempotent*: the result of the action is the same even if the action is invoked multiple times.

6 Node/NodeGroup Resource Model

The current code distribution includes a standard `ShirakoPlugin` module, which is called `cod`. The package evolved from our work in 2002 on Cluster-on-Demand, an early form of cloud computing. COD is designed for a substrate of virtualized servers and file servers—a virtual cloud computing utility.

Most of the package is sufficiently general to handle any substrate that can be viewed as groups of "nodes" of various types. If you squint, that is just about anything. We have determined that the model applies more generically to other kinds of substrate, and we are overloading COD for that purpose. However, in principle, any given actor could be configured to use a different implementation of `ShirakoPlugin`, e.g., to implement an Authority for some substrate provider with a very different concept of resources.

Todo: Since we are overloading the COD package to integrate other substrates and management systems, including Eucalyptus, we have determined that COD should be generalized/genericized and the fishy names should be banished or hidden from view.

6.1 Node/Group Model

COD deals with resource sets (`ResourceSet`) that consist of groups of *nodes*. Nodes (`orca.core.cod.Node`) represent concrete resource objects that are independently configurable and independently programmable. Nodes may represent slivers ("virtual" nodes) or components ("physical" nodes). The same classes are used internally for both.

Nodes are typed and have attributes defined by their type. For example, every node has at least one IP address, and is reachable through IP for configuration actions.

Each node may be an element of a group (`orca.core.cod.NodeGroup`). `NodeGroup` implements the `IConcreteSet` interface, so groups of nodes may back substrate inventory pools or active leases.

The key assumptions of the Node/NodeGroup model in `cod` are:

- Nodes are the basic units of configuration. Each resource handler or guest handler invocation operates on a single node, e.g., by contacting a Node Agent running on the node through its IP address.
- Each sliver is hosted on a single component, i.e., each "virtual" node is hosted on a single "physical" node.
- Nodes may extracted from groups or merged into groups. In general, nodes within a group are independent and/or interchangeable. For example, configuration actions (handlers) may run concurrently on all of the nodes in a group.
- Nodes are state machines whose configuration status can be polled and captured as one of a discrete set of states from `NodeStates`: `priming, active, closing, closed, failed`. These states are used to drive various actions: restart or retry configuration, replace nodes or remove them from service, transition a lease to active, etc.

If the node/group model is sufficient to represent your substrate, then you do not need to provide a substrate-specific implementation of `IConcreteSet`. Lots of substrate semantic can be captured in the handlers and/or drivers (see Section 4.1), and off-the-shelf `ResourceControl` assignment implementations may also be sufficient.

To summarize, the the core views resource management in terms of operations on sets of leases, while `cod` maps actions on the leases into actions on sets of underlying nodes. Loosely, a lease becomes `active` when all of its nodes become `active`. A lease is embedded by invoking `ResourceControl` to assign components for each of its slivers. A lease is instantiated by invoking configuration action handlers for each of its slivers. Figure 4 depicts end-to-end instantiation of a single node, i.e., redeeming a ticket for a single sliver.

6.2 Internal Objects

The `ShirakoPlugin` module for COD contains references to various related classes to extend various internal functions, some of which might also be useful in other implementations, should they be needed.

- `ICodManagerObjectDatabase`. wraps the underlying database object to define additional functionality for a substrate consisting of sets of "machines", named storage servers and devices with properties. All actors within a container share the database. These substrate components are owned by named Authority actors within the container.
- `CodContainerManagerObject`. exports a management API for operations on the substrate, e.g., add and manipulate components (machines and storage servers). authorize authority key to operate on components, install node agent keys, check for pending management operations, query ops to locate machines by site and center, etc. These operate directly on the substrate database.
- `AuthorityCodSlice`. COD extends (subclasses) the slice classes on the Authority side. Among other things, it can assign IP addresses to nodes in the slice from a subnet block owned by the slice.
- `Config`. An attached `Config` object implements the scriptable handler abstraction used to control substrate components (e.g., the *setup* and *teardown* handlers) and slivers (e.g., the *join* and *leave* handlers). The existing `Config` class has support for executing Ant handlers. *Todo:* Extend it for Groovy, Jython, JRuby, or whatever is cool. *Claim:* `AntConfig` has handlers registered by resource type, so we could select the scripting language per-handler there.
- `ICodClientReservation`. COD extends (subclasses) the SM-side lease object for use by a slice controller at its discretion. It simplifies various aspects of property management for configuration and so on. There is also something called `CodPredecessorReservation`. (*Todo:* simplify and reduce fishy odor.)
- `ResourceControl`. Several policy classes are suitable for managing assignment policies for groups of nodes.
- `ImageManager.` Interfaces for managing a registry of images that can be loaded into newly instantiated slivers. See Section 3.3.1.

Function	*Description*	*Example*
join	Incorporate a new resource for a guest.	Join for a guest batch scheduler configures a new virtual machine to be a valid compute machine.
modify	Modify an existing guest resource.	Modify for a guest batch scheduler notifies the scheduler that a virtual machine's sliver size has changed.
leave	Remove a resource from a guest.	Leave for a guest batch scheduler issues directives to the batch scheduler's master to remove a compute machine from the available pool.

Table 4: Guest handler interface for service manager plugins.

Knowledge of the `ShirakoPlugin` object winds deep within the various other parts of a plugin module.

`CodBasePlugin` has most of the common stuff. It takes lease operations and propagates them through to each of the nodes in the underlying group. The following are common to the Authority/AM and SM sides:

- Merging of slice properties into lease properties...but some code in subclasses.
- Hooks to transfer a node in and out of a group. On the AM side, these methods assign names to the new sliver, such as DNS names, IP addresses, and MAC addresses. These may be allocated from name spaces that are slice-wide (e.g., for private IP addresses) or site-wide (e.g., for public IP addresses). These invoke handler upcalls on both the SM and AM sides by indirecting through the `Config` object.
- Also some recovery-related code, e.g., handles revisits.

The Authority/AM side of COD (`Site`) allocates IP subnets and addresses as slivers (virtual nodes) are instantiated. There is a block of private IP space and it allocates subnets of a given size. And there also some publics that are managed as a set. A node can request a public IP address as a configuration property (`visible`). *Todo:* public IP addresses should be an allocatable resource with a ticket.

`SiteBase` has a `PoolManager` for resource pools in the AM substrate inventory. *Todo:* say more.

6.3 Guest Handlers

Guest handlers are action scripts that run within an actor to configure or operate on a logical resource unit (sliver).

Each guest handler includes three entry points that drive configuration and membership transitions in the guest as resource units transfer in or out of a lease. A description of each entry point for the guest handler is shown in Table 4 and summarized below.

- Upon receiving a lease notification from the authority the SM invokes a *join* action for each new resource unit in the lease, to notify the controller that these units have been added to the slice.

- The SM core may invoke a *modify* action for subtype or unit properties that change on a lease extension. To date, the modify event has been closely linked to sliver migration, which has only been prototyped for the authority assignment policy: the `IdControlChange` assignment policy is currently the only place in the code that makes use of this handler.
- Before a lease expires, the SM invokes a *leave* action for each resource unit to give the guest an opportunity to gracefully vacate the resources.

The *join* task is useful for post-install actions that are specific to a guest. The *join* and *leave* guest event handlers may reconfigure the guest for membership changes. For example, the handlers could link to standard entry points of a Group Membership Service in the guest that maintains a consistent view of membership across a distributed application. The guest handlers execute outside of the authority's TCB—they operate within the isolation boundaries that the authority has established for a service manager and its resources.

6.4 Handler Invocation and Properties

Shirako/COD provides a generic implementation of the handler interfaces that allows handlers to be scripted using Ant [1]. Ant is an extensible scripting language whose interpreter is implemented in Java and can be invoked directly from the Java core. Handler scripts in Ant can use builtin functions to interact with the guest through a variety of protocols such as SNMP, LDAP, and Web container management.

Handler upcalls include a property set for the logical unit and its containing lease. The properties are easily accessed from scripted handlers. In general, scripted handlers will reference only the properties and do not call back into the core or interact with other handlers.

A guest may involve multiple physical resources, possibly covered by separate leases. Guest controllers are able to define a *p*redecessor relationship between leases. The handler for a lease may access the properties of its predecessor lease. To see how properties are passed from a predecessor lease to a handler you can look at cod.orca.shirako.kernel.CodPredecessorReservation.

From David Irwin: In general, to reference properties in a predecessor lease prepend "`predecessor.`" to the property in the handler. For example, if I want to to reference my predecessor's IP address then I would use "`predecessor.unit.net.ip`". We also have a way of specifying the prepended value. So, in the case of SGE we typically set the prepended value to be "`master.`" rather than "`predecessor.`". So in this case it would be "`master.unit.net.ip`". There is no way to get properties for other units in the same lease (although the only properties that differ are the unit properites). If there are multiple units in a predecessor lease then the value of the property is delineated by commas. For example, if my lease has two units with two different public IPs then "`predecessor.unit.net.ip=192.168.0.1,192.168.0.2`" would be an appropriate key/-value pair.

The handler may also mark properties, e.g., to represent the return status or other information returned from a configuration action. The relevant code here is in `orca.shirako.plugins.Config`. Any property that begins with "`shirako.`" and set inside of the handler will be stored on the lease object and referenceable from the higher level Java code. For example, we usually set "`shirako.exit.code`" inside of the handler to indicate if an operation executed successfully. The authority assignment policy eventually checks the exit code, and sets a node to failed if the code non-zero.

6.5 Drivers

Handlers may interact with drivers running on the nodes. Many nodes are suitably generic to run Java-based node drivers. (See Section 4.1.2). Examples include physical servers, virtual machines, or any resource that can be controlled by scripts or programs running at some IP address, such as a domain-0 control interface for a virtual machine hypervisor. Node drivers run under a standard *node agent* that runs on a node, e.g., in a guest VM, a control VM (Xen dom0), or a control node for some other substrate component. A *node driver* is a packaged set of actions that run under the node agent to perform resource-specific configuration on a node.

The driver approach enables continuous dynamic control during operation, and it generalizes to guests that are installable after booting as packages. It also enables dynamic installs and upgrades of guest drivers, and asynchronous invocation of arbitrary driver programs or scripts. The node agent accepts signed requests from an authorized actor on the network, e.g., using SOAP/WS-Security or XMLRPC, including requests to install, upgrade, and invoke driver packages. The guest join handler may load the guest driver and guest code itself onto the allocated nodes from an external source, such as an external package repository or Web server, or a shared file service.

The result of each driver method invocation is also a property list, which flows back to the handler that invoked it. The property list conveys information about the outcome of a driver invocation (*e.g.*, exit code and error message). The handler determines how to represent any result properties in the lease state. A driver error reflects back to the handler and is exposed to the Java code in the actor through the node states and unit properties. An error may trigger a failure policy on the receiving AM and SM, e.g., that resets a resource or re-assigns the sliver to an equivalent component. *Todo:* The error codes coming back from our drives are totally cryptic, e.g., error 255 if a ZFS file server fails. Can we rationalize these?

6.6 COD Configuration Properties

Configuring virtual machines. Table 5 lists some of the important node properties for COD. These property names and legal values are conventions among the guest and resource handlers in the SM and AM. Several configuration properties allow a service manager to guide authority-side configuration of nodes.

- *Image selection.* The SM passes a string or GUID to identify an image or appliance from among a menu of options approved by the authority as compatible with the machine type. Each node is instantiated with a logical copy of the named image or appliance.
- *IP addressing.* The site may assign public IP addresses to machines if the `visible` property is set.
- *Secure node access.* The AM and SM exchange keys via properties to enable secure, programmatic access to the leased slivers from the guest handler, as described below.

Resource type properties: passed from broker to service manager		
`machine.memory`	*Amount of memory for nodes of this type*	`2GB`
`machine.cpu`	*CPU identifying string for nodes of this type*	`Intel Pentium4`
`machine.clockspeed`	*CPU clock speed for nodes of this type*	`3.2 GHz`
`machine.cpus`	*Number of CPUs for nodes of this type*	`2`
Configuration properties: passed from service manager to authority		
`image.id`	*Unique identifier for an OS kernel image selected by the guest and approved by the site authority for booting*	`Debian Linux`
`subnet.name`	*Subnet name for this virtual cluster*	`cats`
`host.prefix`	*Hostname prefix to use for nodes from this lease*	`cats`
`host.visible`	*Assign a public IP address to nodes from this lease?*	`true`
`admin.key`	*Public key authorized by the guest for root/admin access for nodes from this lease*	*[binary encoded]*
Unit properties: passed from authority to service manager		
`host.name`	*Hostname assigned to this node*	`irwin1.cod.cs.duke.edu`
`host.privIPaddr`	*Private IP address for this node*	`172.16.64.8`
`host.pubIPaddr`	*Public IP address for this node (if any)*	`152.3.140.22`
`host.key`	*Host public key to authenticate this host for SSL/SSH*	*[binary encoded]*
`subnet.privNetmask`	*Private subnet mask for this virtual cluster*	`255.255.255.0`

Table 5: Selected properties used by `cod`, and sample values.

The unit properties returned for each node include the names and keys to allow the *join* event handler to connect to each machine to initiate post-install actions. A service manager connects with root access through a node agent or some other server pre-installed on the image and started after boot (*e.g.*, `sshd` and `ssh`) to install and execute arbitrary guest software.

Victim selection. Choosing which unit to relinquish on a revoking or shrinking of a lease is important. Some units may be working harder or storing more state than others. Guest controllers overload configuration properties to specify candidate victim virtual machines by IP address on a shrinking lease extension. The current SGE controller does this.

Secure Binding. The slice controller runs with an identity and at least one asymmetric key pair. These must be installed somewhere as defined by the SM framework. Without loss of generality we assume there is only one. Slivers also have an asymmetric key pair. Note: this is just like standard passwordless *ssh* setup, but we want to do it automatically.

The keys are used to bind controllers to their guest instances securely, so that only a properly authorized SM or guest handler can issue control operations on a guest. The idea is to pass the public key with suitable endorsement to the AM and through to the image as a configuration property, and return the public key of each sliver endorsed by the AM as a unit property.

For example, the KeyMaster protocol is implemented using handler plugins that interact through the property lists. The slice controller stores a hash of its public key as a configuration property on each lease. A resource driver on the authority side passes the hash into the sliver image on `setup`, and obtains a hash of a host public key for the newly instantiated guest node, which it returns as a unit property. [Need to say a bit more about how the properties are handled.] The guest's `join` handler executes the guest side of the protocol, passing its public key to the keymaster in the running guest image, receiving the node's public key, and comparing it to the validated hash endorsed by the authority as a unit property. If both keys check out, it triggers a pluggable action on the keymaster side that installs a public key for the guest controller so that it may invoke its own drivers or other root-empowered actions to control the node. The join handler must effect whatever exchange is needed to enable access for its subsequent control operations or that of its drivers. There are various versions of the protocol, and they are a convention among the handlers and the image. But your controller must invoke the right handler.

6.7 Node State Machines

Handlers and drivers execute in separate node state machines within the actor and within the node agent respectively. These tasks execute asynchronously, and task completions generate events to transition the node state machine. The actor FSM for a lease polls the node FSM for each of its resource units, and initiates lease state transitions as operations complete or the subsidiary state changes. When an actor recovers, the node state machines recover the state and local data at the time of the last completed transition. Any handler tasks associated with that state are restarted, and these tasks reissue any driver operations that are not known to have completed before the failure.

Each node state machine, together with the associated lease state machine, contains sufficient information to recreate the arguments to a resource handler so that the handler can be rexecuted during recovery. The portion of information contained in the lease is transmitted in the message from the lease to the node state machine. This information is then combined with the local information inside the node state machine. The resulting request is self-contained and complete and obviates the need for drivers to maintain their own state; the request contains all required information. However, since driver calls may overlap, drivers, similarly to state machines, must handle overlapping actions by either waiting for the completion of or canceling the action in progress.

Handlers and drivers must conform to the idempotence requirement. The basic principle of our approach is that the type-specific resource handlers should follow the *toggle principle*. From the perspective of the actor lease FSM, each resource is in either of two basic states: on or off (possibly failed). The handlers hide intermediate states. Although handlers execute asynchronously from the core, the actor FSM invokes the handlers for each resource unit in a serial order. The handler and the drivers it uses must ensure that the final state reflects the last action issued by the core, independent of any intermediate states, incomplete operations, or transient failures. If handlers are deterministic and serial, then it is sufficient for actions to be (logically) serialized and idempotent at the driver level. That property may require persistent state in the driver (e.g., as in package managers, which typically adhere to the toggle principle); at minimum, it requires a persistent and consistent name space for any objects created by the driver (e.g., cloned storage luns or resource partitions). Drivers must also suppress redundant/duplicate operations triggered by the core to avoid disrupting the guest.

7 Instrumentation

Todo: This is old text: revise and extend.

Instrumentation should be easy to use, but it should also be extensible. A stated goal of Automat is to provide a standard interface for controllers to subscribe to named instrumentation streams in a uniform testbed-wide name space. At the same time, the testbed does not constrain the interactions between a controller and a guest (including its OS kernels), so users can experiment with other instrumentation frameworks. Initially we used Ganglia, but we are moving toward Hyperic.

As always, there is no perfect solution to balance the competing goals of extensibility on the one hand and interoperability and ease-of-use on the other. The guest controller must make certain assumptions about the instrumentation that is available. This has two troublesome implications. First, it creates another area of interdependency between the image and the controller: node-side support must either be built onto the image, or loaded by the guest controller at join time, e.g., from an external file system or package manager. Second, some instrumentation data may be needed from other substrate layers that are not under the guest's direct control, such as the network or hypervisors.

Ultimately we want to support a plugin-based instrumentation framework for guest elements to *publish* instrumentation streams and to *subscribe* to published streams. For example, system or application software in a multitier web service running on leased resources can continuously publish information about the usage of different service components. A guest controller for the service subscribe to this data and use feedback control to implement self-optimizing or self-healing behavior as discussed earlier.

Our initial approach reads instrumentation data published through Ganglia [?], a distributed monitoring system used in many clusters and grids. The default site controller instantiates a Ganglia monitor on each leased node unless the guest controller overrides it. The monitors collect predefined system-level metrics and publish them via multicast on a channel for the owning guest controller.

At least one node held by each guest controller acts as a listener to gather the data and store it in a round-robin database (*rrd*). Guests may publish new user-defined metrics using the Ganglia Metric Tool (*gmetric*), which makes it possible to incorporate new sensors into the Ganglia framework.

While Ganglia is a useful starting point, our intent is not to limit users of the testbed to Ganglia. In the multitier guest, the monitor gathering in the controller runs as a separate thread. It just walks out whenever it wants and hits the Hyperic server, with a plugin written in groovy. [Document how hyperic gets started and hooked in. The metrics and metric processors are specific to the guest applications to varying degrees.]

8 Slice Controllers

Todo: This is old text: revise and extend.

For each guest, the choice of what resources to request and how to use them is made by a programmed adaptation policy implemented by a control server that instantiates and configures the guest and monitors its behavior, making adjustments according to some policy.

The control server is called a *service manager* in SHARP and Shirako papers and code. It generalized the *virtual cluster manager* in the original COD work. Our Supercomputing 2006 paper used the name GROC (Grid Resource Oversight Coordinator) for a controller that manages a Globus grid running as a guest. Like all actors, the service manager is programmable by adding one or more plugin modules. We have used the term *guest controller* to refer to the plugin controllers for the SM.

For GENI, we have deprecated the term guest controller. We try to use the term *slice controller* to refer to this control server, or more properly to plugin modules within the control server that implements the policy and guest-specific support. The terminology has shifted for several reasons.

- GENI participants think of the guest as an experiment. In cloud computing context, the terms service and guest would be more general and meaningful than experiment, but these terms are confusing to GENI participants.
- The distinction between the actor and its plugin controller is also confusing to those who are not concerned with the internal details of ORCA. It is useful to have a name for the actor that reflects the control functions implemented in its plug-ins.
- Experiment controller is decoupled in GENI. Once the guest control functions are factored out, the service manager actor (and its plug-ins) control only the slice, and not the guest.

Importantly, the controller is not part of the utility itself; rather it is an automated control server that runs on behalf of the guest's authorized owner, which is a client of the utility. This view promotes a clean separation of guest-specific logic from a common underlying virtual computing platform, which is "guest-neutral". New guests and control policies may be deployed as plug-in extensions, without changing the ORCA software itself.

To recap, a Shirako slice controller is a plug-in module for a service manager actor, designed to configure and control some guest. It interacts with resource servers if the resource needs of the slice change, and it notifies the guest of any change to the status of resources in the slice. There is exactly one controller per slice, but a service manager actor may operate on multiple slices, and a single controller may manage multiple slices within the actor.

The control available to guests is determined by the basic common resource management mechanisms and the specific interfaces for each resource provider and broker. For example, guests might like to control colocation and migration. But since service managers do not control the infrastructure directly, the brokers and substrate authorities must run policy modules to support those features, and define properties to guide it. The Orca software release has basic handlers and policy modules that produce and consume various defined properties.

8.1 Installing Controllers

The system defines controller name scopes and limited means for controllers in the same scope to interact. A user can create a scope and authorize other users to install and manipulate controllers in the same scope. Each scope includes a registry for controllers in the scope. Controllers in the same scope can share limited state; in particular, they may access information about the resources currently held by the others. For example, a workload controller can obtain lists of IP addresses for the virtual servers running the system under test, enabling it to direct workload at those servers. Actors are the granularity of controller isolation. A service manager may register multiple controllers operating on different slices and leases. Controllers in the same actor share a common name registry (scope) and access control list; they run within the same JVM and they may share state.

Users may upgrade controller modules and their views dynamically. Our design uses a custom classloader in a Java-based Web application server to introduce user-supplied controller code into the portal service. The portal has menu items to restart an actor so that changes to its controller implementations take effect; the actor state is restored from a database from a standard schema consisting of system-defined entities (valid leases and pending requests) annotated with arbitrary properties for use by the controller implementation. Note that a custom view plug-in could be used to manipulate a declaratively specified controller policy on-the-fly without restarting it, assuming the controller's Java code is not affected.

8.2 GENI Solicitation 2 Goals

Flexible control of experiments across diverse and federated substrates is at the heart of what GENI is trying to achieve. The Orca project, and our partners in the GUSH and ORBIT projects, have promoted the idea of autonomous, programmable slice controllers as first-class entities within GENI. However, we do not yet fully understand the control flow, monitoring linkages, and authorization models needed for this to work.

We propose to enable more powerful slice controllers by enhancing the Orca core (slice controller state machine and aggregate manager/authority interfaces) to configure linkages among constituent parts of a slice ("stitching"), and to trigger adaptations to a slice or actions within a slice in response to monitoring data and events. This goal requires integrating instrumentation APIs, enhanced configuration sequencing, etc.

Goal: more advanced "stitching" of slice elements across aggregates, instrumentation-driven feedback to the slice controller, and delegation of identity credentials based on Shibboleth.

8.3 Slice Controller Structure

Figure 5 shows an overview of a service manager and the slice controller's API to the service manager for calls in both directions (upcalls and downcalls). The black boxes are the service manager's protocol interfaces to other actors. The white boxes outside the guest controller are classes and interfaces for the controller's convenience. The `query` method sends a query request to another actor and the `demand` method prepares a new lease object to generate a ticket request to a broker. The figure shows the controller's use of layered calendar classes within the Shirako toolkit to track its tickets and leases by time.

The controller goes through several steps to issue a request. The following are the general steps of a guest controller policy:

1. **Track guest status and translate into resource demand.** The controller learns the state of the guest to make decisions about how to service its resource needs. This is application-specific and the guest controller may implement it in a variety of ways.

2. **Track resource status.** The controller inspects the current state of its resource holdings and their expiration times, which are indexed by calendar classes in the toolkit. Since leases are dependent on time, the calendars provide a way to store and query present and future resource holdings. They also provide convenient methods to query lease objects by their states.

3. **Formulate resource requests.** The controller formulates its requests as `ResourceLease` objects and sets its desired term, number of resource units, and resource types for each one (Section 4.3). To define additional attributes on the request (e.g., request flexibility), the controller attaches *request properties* to the `ResourceLease`. These express additional constraints, degrees of freedom, or objectives (such as resource value) to a broker policy to guide resource selection and allocation.

4. **Issue the request.** Once the controller formulates its requests, it submits it using the `demand` method. The service manager shell verifies that each request is well-formed and queues it for transmission to the selected broker.

8.4 Controller Policies

Here are some of the classes that are relevant to the controller upcall interfaces other than guest handlers:

- `api/IReservationEventHandler` has the full interface for guest event handlers.
- `api/IPolicy`, `IClientPolicy` and `IServiceManagerPolicy` make up an interface class hierarchy that defines the interface for guest controller policy modules upcalled from the core.
- `core/ServiceManagerPolicy` has the service manager policy base class upcalled from the core, including registration and vectoring for the lease event handlers. It implements `IServiceManagerPolic`
- `kernel/ReservationClient` is the core class that implements the service manager lease state machine. It is the "neck" of a Shirako service manager. This is where the core upcalls into the guest lease event handlers.

In `./policy/core/src/main/java/orca/policy/`:

- `./core/ServiceManagerCalendarPolicy` extends `ServiceManagerPolicy` with methods to use the calendar classes to track reservations. This is the base policy class that is used in the SGE and JAWS guest controllers.

The simplest framework built above raw Shirako for guest controllers makes a number of assumptions: ask right away (no futures contracts), one default broker, auto-extend, advance close. And for that we have simple plugins. But if we want to be smarter with futures etc?

Guest controllers are associated with instances of a standard controller subclass of `ManagerObject` called `ControllerManagerObject`. The package is `orca.manage.extensions.standard.controllers` in `./manage/extensions/standard/src/main/java/orca/manage/extensions/standard/controllers`.

The guest controller itself is registered with the `ControllerManagerObject` and implements the interface `IController` so that it is registerable as a guest controller on a per-slice basis.

Shirako `api/IController` is an interface that is registerable on a per-slice basis in the service manager.

8.5 Handler Invocation

Ant defines the notion of a task: each task is responsible for executing a particular operation. Ant comes with a set of basic tasks, and many others are available through third-party projects. Ant tasks and the Ant interpreter are written in Java, so the authority and service manager execute the resource and guest handlers by invoking the corresponding Ant targets directly within the same JVM.

This is convenient because *Ant* tasks are available for many of the configuration actions that guest handlers need to use, *Ant* scripts can be used easily from a command line, and guest handlers operate on specific resource units and do not need to call back into the core. Invoke and node driver actions on the node agents. Ant makes it possible to assemble a handler from a collection of building blocks without requiring recompilation.

Guest handler scripts may be named from the controller as relative to the controller directory. They are XML ant scripts: the guest handlers for join/leave on the worker and master...named within the controller, and packaged in a subtree with the Java code...in a separate subtree of the controller package, called “resources”. The names used in the controller are relative to “resources”.

8.6 Active Issues and Areas of Focus

These are areas in which there is running code but still lots of active work, so the current system is changing and therefore undocumentable. Most of them are directly related to the research issues outlined in the previous section.

Portal facilities for a guest package library. One practical focus of the Automat project is to provide configuration tools that are sufficiently flexible and easy to use to allow users who aren’t intimately familiar with the underlying platform to build controllers or assemble them from off-the-shelf elements. In particular, the Web portal provides facilities to to install named guest packages and controller elements and instantiate them from a library, which may include hosted applications, workloads, and faultloads.

Reusable controller elements. For Automat, we are also working to enhance the support infrastructure for building effective controllers for complex applications. The Java interfaces are a foundation for more advanced controller tools. Our initial focus is on the Java interfaces to enable implementation of a comprehensive space of controllers. Automat will provide more advanced classes for calendar scheduling, resource allocation, processing of instrumentation streams, and common heuristics for optimization and placement.

Views interact with the controller using a management proxy supplied as part of the controller implementation. The proxy always runs within the portal service, but the controller code itself and its actor may run under a separate JVM communicating using SOAP or XMLRPC. This separation can allow the system better control over the resources consumed by the user-supplied controllers, and enhance isolation and dependability for testbed users.

In principle, controller views could also present a fault injection interface, e.g., to introduce exceptions, deadlocks, or other errors into an application through a custom driver. Automat users may monitor the real-time progress of an experiment using a generic applet to display per-node or per-guest metric streams named by the the controller.

9 Resource Authority Actors

[Omitted]

10 Broker Actors

[Omitted]

References

[1] Ant. `http://ant.apache.org/`.

[2] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, and C. C. S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth Symposium on Operating System Principles (SOSP)*, December 1995.

[3] 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 (an IBM-sponsored workshop)*, May 2008.

[4] 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.

[5] 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.

[6] I. Constandache, A. Yumerefendi, and J. Chase. Secure Control of Portable Images in a Virtual Computing Utility. In *First Workshop on Virtual Machine Security (VMSec)*, October 2008.

[7] A. Demberel, J. Chase, and S. Babu. Reflective Control for an Elastic Cloud Application: An Automated Experiment Workbench. In *Proc. of the First Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2009.

[8] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, December 1995.

[9] 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.

[10] 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.

[11] 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.

[12] H. Lim, S. Babu, J. Chase, and S. Parekh. Automated Control in Cloud Computing: Challenges and Opportunities. In *Proc. of the First Workshop on Automated Control for Datacenters and Clouds (ACDC)*, June 2009.

[13] 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.

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

```
public interface IShirakoPlugin
{
   public ISlice createSlice(SliceID sliceID, String name, ResourceData properties,
       Object other);

   /**
     * Returns an <code>IConcreteSetFactory</code> for the given
     * communication protocol
     *
    */
    public IConcreteSetFactory getFactory(String protocol);

    /**
     * Returns the ticket factory in use by this actor.
     * @return
     */
    public IResourceTicketFactory getTicketFactory();

    /**
     * Sets the ticket factory to use.
     * @param factory
     */
    public void setTicketFactory(IResourceTicketFactory factory);

    /**
     * Releases any resources held by the slice. Note: the database
     * record will not be removed.
     *
     * @param slice the slice
     *
     * @throws Exception if releasing resources fails
     */
    public void releaseSlice(ISlice slice) throws Exception;

    /**
     * Restarts any pending configuration actions for the specified
     * reservation
     *
     * @param reservation reservation
     *
     * @throws Exception if restarting actions fails
     */
    public void restartConfigurationActions(IReservation reservation) throws
        Exception;

    /**
     * Rebuilds plugin state associated with a restored reservation.
     * Called once for each restored reservation.
     *
     * @param reservation restored reservation
     *
     * @throws Exception if rebuilding state fails
     */
    public void revisit(IReservation reservation) throws Exception;

}
```

Figure 3: `ShirakoPlugin` interface summary: the major internal plugin interface for substrate-specific elements of a Shirako actor. Most integrators will not program directly to this interface: the `cod` implementation will be sufficient in most cases.

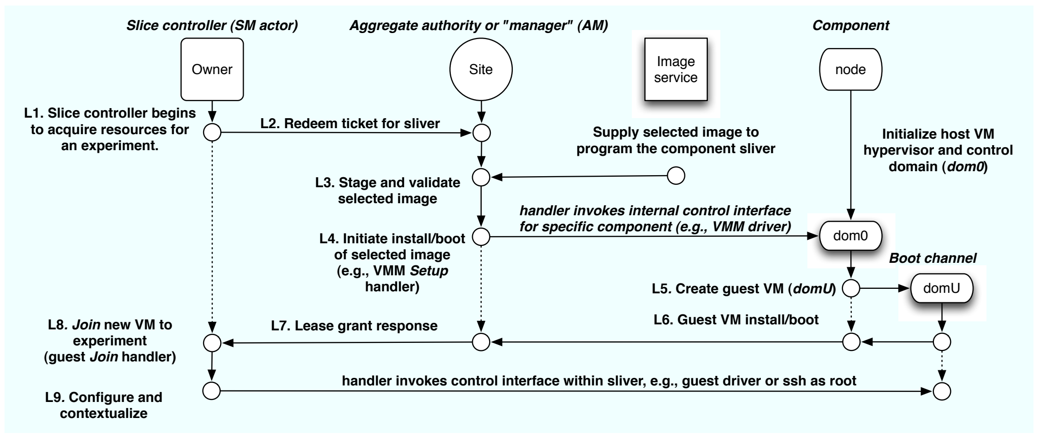
Slice controller (SM actor)
Aggregate authority or "manager" (AM)
Component
Owner
Site
Image service
node
L1. Slice controller begins to acquire resources for an experiment.
L2. Redeem ticket for sliver
Supply selected image to program the component sliver
Initialize host VM hypervisor and control domain (dom0)
L3. Stage and validate selected image
handler invokes internal control interface for specific component (e.g., VMM driver)
L4. Initiate install/boot of selected image (e.g., VMM Setup handler)
dom0
Boot channel
L5. Create guest VM (domU)
domU
L6. Guest VM install/boot
L7. Lease grant response
L8. Join new VM to experiment (guest Join handler)
L9. Configure and contextualize
handler invokes control interface within sliver, e.g., guest driver or ssh as root

Figure 4: Steps to instantiate a sliver on a substrate component. In this example, the sliver is a virtual node (a VM) and the component is a host node (a hypervisor-enabled server). The leasing engine upcalls a plugin resource handler on the AM side `setup` the sliver on the component (step L4), and a plugin guest handler on the SM side to attach to `join` the new sliver to the guest (step L8). These handlers may invoke node drivers or other interfaces on the component (step L5) and sliver (step L9) respectively. The AM runs an assignment policy (`IResourceControl`) to select a component to host the sliver, prior to step L3.

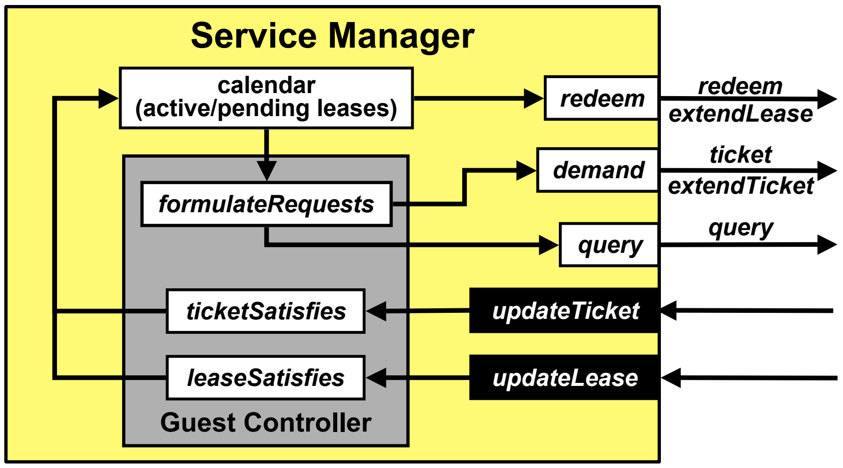

Figure 5: An overview of a service manager implemented with the Shirako toolkit. Shirako defines the leasing protocols (e.g., `updateTicket`) as well as methods such as `demand` to send a ticket request to a broker. The slice controller defines a clock tick upcall (`formulateRequests`) and (optionally) lease event handler interfaces, including `ticketSatisfies` and `leaseSatisfies` methods to qualify incoming tickets and leases.