# Orca Technical Note: Guests and Guest Controllers

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

June 25, 2008

**Abstract**

*This paper incorporates material from a variety of previously published works with various authors, including Laura Grit's thesis and David Irwin's thesis. It supersedes related material from those sources.*

## 1 Introduction

This note explains support for dynamically adaptive guest applications in Orca. Orca is an software framework and open-source platform to manage a programmatically controllable shared substrate, which may include servers, storage, networks, or other devices. In this note we focus on a substrate of virtualized servers—a virtual cloud computing utility. Storage services are handled as special virtual servers using the facilities described here. Extensions for networks are under development and are out of scope for this note.

### 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 [3]; the Automat [5] control portal and related components; a new implementation of the SHARP framework [2] for accountable lease contracts and brokering; Cluster-on-Demand (COD [1]), 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 even within the group about when to use the name Orca rather than the names of the component implementations. 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 architectural principles extracted from the overall project, and now use specific component 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. For simplicity, we also frequently use the name "Orca" as a shorthand when it is unnecessary to distinguish among specific components or their implementations.

## 1.2 Hosts and Guests

Utility computing embodies the “host/guest model”. The substrate hosts diverse applications or environments as *guests*. In general, a guest is a distributed software environment with multiple virtual machines configured to order, possibly at different provider sites. The guests may range from virtual desktops to dynamic instantiations of distributed applications and network services, such as job management systems, multi-tier Web services, or “personal compute clouds” (e.g., Hadoop/MapReduce).

Some guests will be long-running services that must maintain service quality across flash crowds, faults, and other changes. Some may require different amounts of resources at different stages of execution. The platform must support *Dynamic guests* that monitor changing conditions and adapt as their needs change. To this end, Orca exports programmatic, service-oriented interfaces for self-managing guests to negotiate for resources and configure them on-the-fly.

In the Orca framework, a collection of automated control servers (*actors*) interact to provision and configure resources for each guest according to the policies of the participants. The actors represent various stakeholders in the shared infrastructure. 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. We use the term *guest controller* to refer to this control server, or more properly to plug-in modules within the control server that implements the policy and guest-specific support. Guest controllers may implement sense-and-respond policies for self-management and self-repair, including adaptive resource provisioning.

The control server is also called a *service manager* in SHARP and Shirako papers and code; guest controllers run within the “service manager” role, which 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.

Importantly, the guest 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.

## 1.3 Automat: a Testbed for Dynamic Guests

The framework for pluggable controllers is a vehicle for experimenting with the mechanisms and policies for *autonomic hosting centers* that configure, monitor, optimize, diagnose, and repair themselves under closed-loop control. The rich policy space for adaptive hosting centers and guests is a primary focus of our ongoing research and development. This activity is central to the Automat project, which NSF funded in 2007 under the title *Foundations for a Programmable Hosting Center.*

Automat was conceived as an open testbed architecture for adaptive services and autonomic hosting centers based on the Orca platform and Shirako/COD implementation. The intent is to allow users to install their own guest services, experiment with pluggable controller modules, and explore the interactions of application services, workloads, faultloads, and control policies. Ultimately we expect the testbed features for Automat to generalize beyond a single hosting center to a networked testbed substrate such as GENI. The scope includes resource management policies at the infrastructure level, but this note focuses on dynamic guests.

The software we call “Automat” has several parts, which we are under active development:

- Glue to integrate relevant off-the-shelf components for automated use within the Orca platform. These include instrumentation systems (notably Hyperic), standard test applications (e.g., Rubis and supporting infrastructure such as clustered JBoss), and various load generators. This “glue” code consists mostly of handlers and drivers, as described in Section 2.

- A set of extensions to the original Shirako Web portal to allow users to upload and deploy guests and controllers, subject them to programmed test scenarios, and record and display selected measures as the experiment unfolds.
- Tools for building complex controllers. These are layered above the original Shirako controller upcall interfaces, and/or are based on the original Shirako classes for reservation calendars.
- Extensions to 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 all of the resources and delegates them to actors, which may be instantiated dynamically as the granularity of extensible policy control. A key example is the Automat idea of a *virtual data center*, which allows researchers to experiment with hosting center management policies on a subset of the resources in the facility.
- Packaged support for controllers to subscribe to instrumentation streams published by the infrastructure elements, or by various levels of the software stacks including the guest applications and operating systems. These data streams may act as continuous feedback signals to drive the control policies or as inputs for various forms of diagnosis and statistical analysis.

What we call the Automat project also encompasses research in control policies built using these tools, and evaluated on the Automat testbed. Some of this research deals with control policies for a hosting center or other resource providers (i.e., the authority servers for sites or domains, or other substrate component aggregates in GENI terminology) and brokering intermediaries that handle federation and coordinated provisioning across multiple provider sites (corresponding to the GENI concept called Clearinghouse). These aspects of the project are outside the scope of this note, which focuses on instantiation and control of dynamic guests.

Note that PlanetLab shares this focus on a testbed for dynamic guests. The Automat concept is distinct from PlanetLab in two major ways. First, it enables experimentation with autonomic management at the infrastructure level as well, as noted above. Second, it focuses on adaptive resource control and negotiation for dynamic guests, while PlanetLab was conceived for a best-effort model. Among other limitations, the best-effort model makes it difficult or impossible to produce repeatable experiments. This is not to be critical of PlanetLab, which runs over the live Internet and cannot control it (yet). It is just to summarize one key difference in our focus. With the advent of virtual networks these platforms may ultimately combine in GENI.

## 1.4 Research Issues

There are a number of research issues associated with controllers in the Automat project, and more generally in Orca. The "hard" problems engage the long-standing deep issues in robust and effective feedback control of complex software systems. These problems are particularly interesting with systems for which knowledge of the software behavior is incomplete or uncertain. Models must be inferred from instrumentation data, which may also be incomplete or uncertain. Moreover, behavior may change over time. Changes in behavior may result from the functions of the software itself, or they may be due to external changes in the workload or resources, which might not be apparent to the controller.

The problems become even more interesting when aspects of optimal resource allocation come into play, requiring actors to assign value to candidate allocations and select among alternatives. In particular, Orca is designed to serve as a platform for a resource economy. Economic actors act in strategic self-interest under limited budgets or other constraints, according to rules set by auction protocols or other mechanisms. All of these actor behaviors would be based within the controllers. The global system behavior emerges from the controller interactions among all of the actors in the system.

Leaving those issues aside for the moment, there are also some practical architectural challenges for the underlying software platform, a few of which also rise to the level of research problems.

**Facility control.** The concept of supporting a testbed facility on the ORCA platform raises some architectural questions that may drive changes in the underlying platform. Should we base facility delegation (virtual data centers) on a generalized leasing concept that allows arbitrary nesting of leases? How should the facility support unified monitoring and control of groups of actors that are part of the same experiment or system? For example, the management plane should support monitoring and fault injection across systems with many actors. This is crucial to the goals of Automat (and GENI), but there is no support for it in Shirako/COD per se, and many open questions about the appropriate level of abstraction to export information and control.

**Guest neutrality for controllers.** An important goal of the Automat project is to support separate development of modular guest controllers, and even of generic controllers that manage whole classes of guests. What aspects of controllers are bound to specific guests, and what aspects are generalizable in a way that is applicable to multiple guests? How to separate the guest-specific elements from the guest-neutral elements?

**Composing modular controller elements.** If some controller elements are generalizable, what is the right way to combine these elements into complete controllers for a specific guest scenario? How do the elements interact? What communication abstractions are appropriate (e.g., events, blackboard)? What features are appropriate for a language to specify complex control policies by combining off-the-shelf elements? Software layered above Automat could provide tools to specify controller policies declaratively, or assemble them by drawing on libraries of reusable controller components.

**Coordinating controller elements.** If a controller combines multiple elements, how to govern their interactions? How to reconcile their policy decisions when they conflict, or coordinate them when they are not fully independent? For example, consider a complex application with many interacting components, such as a familiar multi-tier web service. The tiers may be provisioned and configured independently by "separate" controllers, but the optimal choices for each tier often depend on the choices made for the others. The hard part is how we organize controllers so that they can provision all the leases for this complex application in a coordinated way.

**Contracts and negotiation.** Guest controllers acquire resources by requesting leasing contracts from brokers and component authorities. There is a large space of possible languages and features for advanced contracting and negotiation to explore. What is the "right" way to represent available resources, requests for resources, and contracts conferring the right to use specific resources? This issue is at the root of the lengthy debates over the GENI *RSpec* and the tension between standardizing it ("Lingua Franca") and leaving it open for extensibility ("Tower of Babel"). *RSpec* was conceived as a language to represent resources in the PlanetLab substrate, but in the general case requests and contracts may include other elements, e.g., a space of candidate resource alternatives and valuations, complex bids, varying degrees of probabilistic assurance, and constraints on placement or colocation. How should these be represented? Should they be folded into *RSpec* or specified separately?

**Adaptive instrumentation.** How does a guest controller interact with the instrumentation system? How does it know what metrics are available, or control metric collection and granularity, or otherwise subscribe to instrumentation streams? What aspects of the instrumentation plane should be replaceable or configurable? How does a controller initiate queries against the instrumentation data? Can it control where those queries execute?

## 1.5 Reflective applications

Another intriguing research direction explores *reflective* applications that introspect on the resources available to them and change their functions accordingly. We might structure such a reflective application in any one of a number of ways. One way is to build an advanced guest controller whose policies tell the guest application what to do, rather than simply managing its mapping onto leased resources.

For example, consider an automated workbench system to meet a high-level benchmarking objective, such as mapping a response surface to show the impact of a set of workload and configuration parameters on the peak throughput of a network file service [4]. A workbench controller initiates a set of experiments, each of which configures a test system and workload generator to obtain performance measures for some specified setting of the parameters. A planner selects the settings for each new experiment to maximize the yield of new information, considering the experiments it has already initiated and their results.

The automated workbench problem is well-suited to virtual infrastructure. The workbench controller could be built as a guest controller that leases resources and configures them to order for each experiment. It can use the leasing mechanisms to instantiate new test systems and workload generators on the fly as it obtains resources. At first glance this appears to be an ordinary example of a guest application: it runs its experiments whenever suitable resources are available. It could run multiple experiments in parallel if it has sufficient resources. Its rate of progress depends on the rate of resource flow to it, but it is not truly "adaptive" in the same sense as a network service, because it defines its own workload internally and its workload does not change.

What is new here is that this guest could incorporate an application-driven form of adaptation that is sometimes called "reflection": the policy might consider availability of resources in planning what experiments to conduct. For example, if the experiment planner considers the results of prior experiments, then it might act speculatively to make use of new resources that become available before a previous experiment completes. In general, the controller must understand the set of candidate experiments and their dependencies to decide how to make the best use of resources as they become available. The automated workbench is an interesting context to study integrated control policies that balance accuracy, time-to-result, and overall cost.

Reflection blurs the implicit separation of the guest application and the guest controller. We originally conceived guest controllers as add-ons to existing applications that are dynamic but not under feedback control; that is, the guests can handle changes to their resource sets, but do not drive those changes themselves. For example, a batch scheduler for a compute cluster can make use of new servers added to the cluster, or restart jobs if a server fails. We can run such a system as a guest and add a controller to resize its virtual cluster on the fly. In this case the controller does not affect the jobs that are actually executed; it only affects the schedule. But once that control capability exists it introduces the opportunity for reflective controllers that are integral to the controlled applications.

From the standpoint of the software architecture, reflection introduces a new need for the guest controller to interact directly with the guest application. The controller may need to invoke operations on the guest application directly, e.g., to submit a task or receive notifications that some task is complete. What is the right way to manage these interactions within the controller framework, or extend the framework to support such interactions?

## 2 Overview

An Orca system is a dynamic collection of interacting control servers (actors) that work together to coordinate asynchronous actions on the underlying resources. 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.

There are three basic actor roles in the architecture, representing the providers, consumers, and intermediaries respectively. Figure 1 depicts these actors and their interactions. An *authority* controls infrastructure resources in a particular site, administrative domain, or component aggregate comprising a set of servers, storage units, network elements, or other components under common ownership and control. A *service manager* procures lease contracts granting rights to use specific resources at one or more sites, and deploys a guest environment or software application on the resources it holds. The service manager acts as a controller

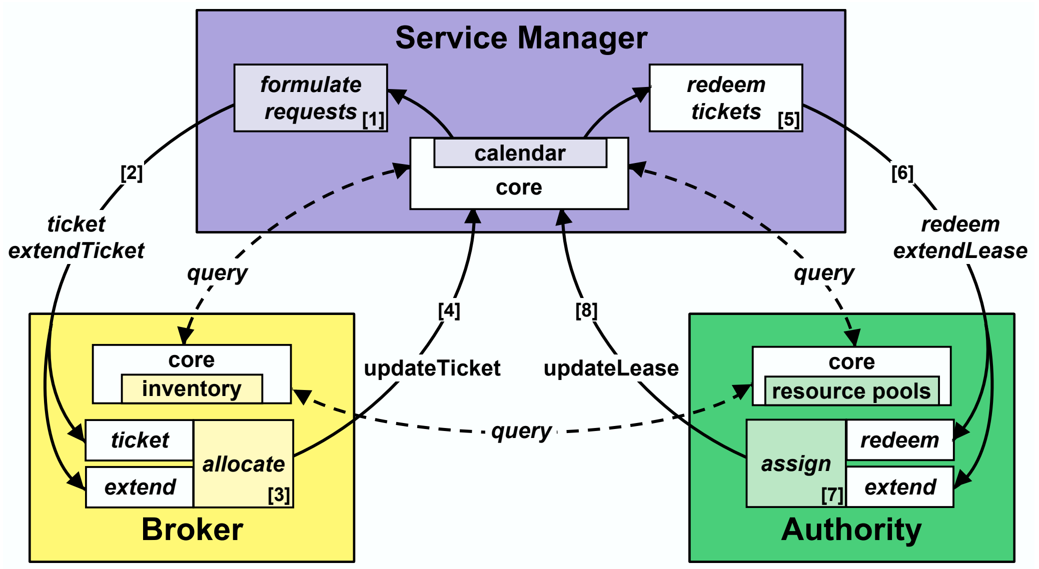

Figure 1: Leasing interactions and extension points for the leasing system. Colored boxes within each actor represent the controller policy module plugins to the leasing core. The arrows illustrate the leasing protocols. A service manager formulates its requests [1] and issues one or more `ticket` or `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.

for the guests that it deploys. Finally, a *broker* mediates resource discovery and arbitration by controlling the scheduling of resources at one or more sites over time.

We use the Shirako toolkit to build Orca actors in the Java environment. This section gives an overview of the interfaces and "moving parts" of a Shirako-based system, focusing on instantiation and control of dynamic guests. The goal is to summarize the structure of a guest controller that uses leasing protocols to acquire, configure, contextualize, and release server resources at virtual utility sites. While we focus on Shirako, we expect that alternative software for Orca systems would embody similar concepts and interfaces.

## 2.1 The Leasing Core and Plugins

Shirako may be thought of as a distributed lease manager: it implements a set of protocols and conventions for actors to negotiate and coordinate lease contracts. 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.

A guest controller is just a set of plugins that run within a service manager and use the Shirako APIs to configure and control some guest. 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 plugins use downcall APIs on the local lease store to track their resource holdings and mark lease objects to drive the flow of requests and configuration actions.

This design reflects the Exokernel principle of visible allocation, modification, and revocation for extensible resource management. All events that affect the disposition of the guest's resources are reflected to the guest through upcalls to its controller plugins.

## 2.2 Guest Plugin Structure

There are three kinds of Shirako core plugins that are relevant for guest controllers:

- **Controller policy modules.** A control policy is driven by a clocking signal; on each clock tick upcall it examines its resource holdings and any selected instrumentation measures, then formulates new requests to adjust its resource holdings as needed to meet its objectives under changing conditions. This is the hook for feedback control policies.
- **Lease event handlers**. The guest controller may register event handlers for upcalls to notify it of completed changes or imminent changes in lease status (e.g., lease has become active, lease is about to expire).
- **Guest handlers**. The core invokes plugin handlers as each logical resource unit joins or leaves the guest's resource set. A typical example of a logical resource unit is a guest node, e.g., a virtual machine inhabiting a sliver. The *join* handler initializes each leased node to configure it, assign it a role in the application, and transmit any needed context information. The *leave* handler terminates any guest application functions on a node prior to relinquishing it. In essence the guest handlers are prologue/epilogue actions invoked at the start and end of a lease.

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. For example, Shirako/COD also provides a standard Java shim plugin that allows scripting of guest handlers using *Ant*. 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.

Plugins may also use or incorporate other code objects, which may run on a different machine from the actor itself. The software currently supports two examples:

- **Drivers.** Controllers and handlers often invoke external programs or scripts that run configuration actions directly on a controlled component (e.g., a guest node). There is a common need for such scripts, and we have considered how to support them and integrate them properly with the operation of the core for robust behavior in failure/recovery scenarios. To this end we have developed a framework for dynamically installable *driver* packages that run under a component agent (node agent) on the controlled resources, and are invoked from a controller or handler. Use of the driver framework is optional but it offers certain advantages. Drivers are discussed in Section 3.7.
- **Views.** 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 guest controller plugins can implement a graphical interface to set attributes or parameters for the guest, e.g., to modulate a workload generator. Views plug-ins 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.

## 2.3 Property Lists

The lease protocol interactions in Figure 1 must carry whatever information is needed to guide resource management and configuration. For example, a guest may need to pass specific requirements for configuring a resource to a site authority. Similarly, an authority must pass information about each resource unit back to a service manager so that it can use the resource effectively. For example, consider the IP address of a newly leased guest node. In principle, it could be either the site authority or the guest controller 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, a guest controllers request resources, but these requests are subject to an arbitration policy at the broker and an assignment policy at the authority, which may require more detailed information about the guest's preferences.

Rather than trying to "screw down" these various design choices, we instead factored any resource-specific and actor-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".

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 site providers 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.

## 2.4 Guest Packages and Configuration

An Orca system can can host a wide range of cluster-based guest applications or workloads. What is needed is to package the guest as a set of executable images for various components in the programmable substrate, together with wrappers that can launch the guest and control it. These wrappers include configuration code (handlers and drivers) and controller policy modules built to the plugin interfaces.

Let us limit our attention to a programmable substrate that allocates virtual machines. Good mechanisms exist for static 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. Tools package the software as an "appliance" 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.

A practical Orca system must bridge the gap between the static packaging and dynamic control after the 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.

An important challenge is how best to manage dependencies among composable controller elements, and ensure consistent configurations. For example, the set of properties and their meanings is an important part of the specification for any site or broker, and guest 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.

The Orca 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 users.

### 2.5 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.

**Instrumentation.** 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.

## 3 Design

Here we expand on the concepts and give more detailed descriptions, illustrating with specific examples with virtual computing in the current Shirako-based prototype.

We first summarize the data model: properties and lease objects. Then some examples relating to how plugins operate on these date elements to monitor and control what is going on. Some of that is related to how lease contracts are negotiated and how they change. Then more about plugin structure and interfaces.

Then we present some more detailed examples from virtual cloud computing, and specific guest examples.

### 3.1 Properties

We must specify how plugins access properties, what properties are available, and which plugins may read or write specific properties. For this purpose Orca classifies properties into three groups:

**ResourceLease**

```
// get attributes
public Term getTerm()
public int getUnits()
public ResourceType getResourceType()

// get/set properties
public void setProperty(String propertyName, String property)
public String getProperty(String propertyName)

// state queries
public boolean isActive()
public boolean isTicketed()
public boolean isClosed()
public boolean isFailed()
public boolean isNascent()
```

Figure 2: An overview of the Shirako lease object interfaces. An actor may query the `ResourceLease` attributes and state. The interface also defines a way for an actor to set properties on the `ResourceLease`.

- *Resource properties* specify attributes that are common to all resources of a given type. Brokers attach resource properties to each ticket. The properties are readable to all downstream plugins on the guest and authority. [Can authority assignment policy or anyone else ever modify them?]
- *Configuration properties* direct how the authority instantiates or configures resources on setup, or modified on an extend. The guest 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 authority at redeem or extend time, where they are readable to the authority-side plugins.
- *Unit properties* define additional attributes for each resource unit assigned. Authority plugins attach unit properties to each resource unit. The core transmits them to the service manager with each lease and then through to the guest plugins.

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. Most unit properties and type properties define immutable properties of the resource that service managers and authorities may cache and reuse freely. Finally, the slice itself also includes a property set that all leases within the slice inherit.

## 3.2 Lease Objects

Shirako actors retain and cache lease state in memory. Each actor has a local lease object to represent each lease, regardless of its state. There are different lease classes with interfaces tailored to the different actor roles. We summarize the lease interface using a fictional class called `ResourceLease`. Since the toolkit lease classes are layered to offer different views of lease functionality, there is no class that corresponds directly to `ResourceLease` as described here, although there could be and perhaps should be. The point here is to introduce the abstraction: specific implementation details are not included for ease of exposition.

Figure 2 shows the primary interfaces of a `ResourceLease` object on the service manager.

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.** The state for a brokered lease spans three interacting state machines, one in each of the three actors involved in the lease: the service manager that requests the resources, the broker that provisions them, and the provider that owns and assigns them. Each lease object has local state variables that represent the actor's view of the lease state. The lease state machines govern all functions of the core.

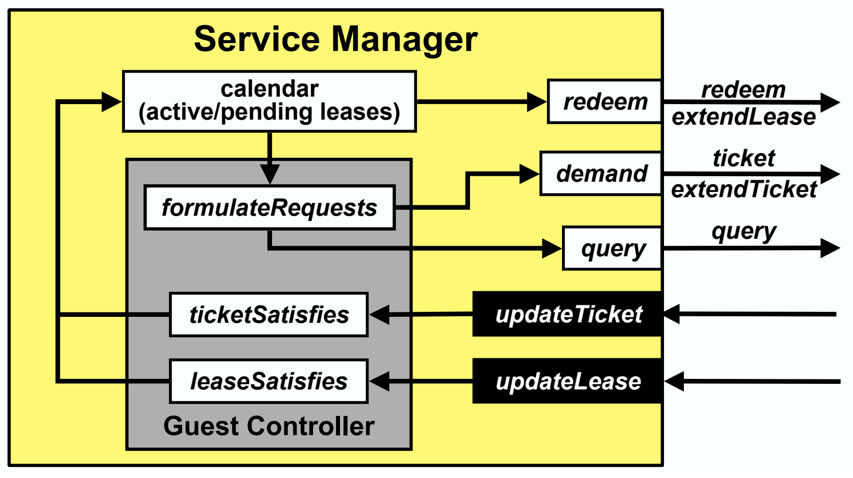

Figure 3: An overview of a service manager implemented with the Shirako toolkit. Shirako defines the leasing protocols (e.g., `updateTicket`) as well as methods such as `demand` to send a ticket request to a broker. The guest controller defines a clock tick upcall (`formulateRequests`) and (optionally) lease event handler interfaces, including `ticketSatisfies` and `leaseSatisfies` methods to qualify incoming tickets and leases.

State transitions in each actor are initiated by arriving requests or lease/ticket updates, and by events such as the passage of time or changes in resource status.

- **Term.** The interval of time over which the lease is valid. In practice there are multiple term variables used during negotiation. When a guest 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 `ResourceLease`'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. However, various resource properties associated with the type may be mutable. 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).
- **Unit count.** Each lease pertains to a block of units with identical resource attributes, e.g., a cluster of virtual machines with identical size.

### 3.3 Guest Controller Structure

Figure 3 shows an overview of a service manager and the guest 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.

Figure 4 gives an overview of the steps a guest controller goes through to formulate and demand a request. The following are the general steps of a guest controller policy:

1. **Track guest status and translate into resource demand.** The guest 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.

**Guest Controller**

```
/* query hosted application for status and
   translate status into resource demand */
application.query();

/* query calendar for current holdings (active leases) */
calendar.getHoldings(now);

/* (optional) query broker for properties */
Properties props = Broker.query();

/* create a ResourceLease */
ResourceLease request = new ResourceLease(units,resourceType,startTime,length);

/* set request properties */
request.setProperty(renewable,true);
request.setProperty(elasticSize,true);
request.setProperty(deferrable,true);
                    ⋮
request.getResources().setProperty(elasticSize.min,10);

/* demand request by sending ResourceLease to service manager */
serviceManager.demand(request);
```

Figure 4: An outline of the steps for a guest controller to generate a request by using the interfaces defined in the Shirako toolkit. To determine the required resources, the guest controller queries the guest application and compares its resource needs to its current resources in its active leases. The guest controller may also query a broker to determine how to formulate its request for arbitration by the broker's policy. In this example the guest controller creates a `ResourceLease` and attaches various properties before submitting the request to the selected broker.

2. **Track resource status.** The guest 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 guest controller formulates its requests as `ResourceLease` objects and sets its desired term, number of resource units, and resource types for each one (Section **??**). 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 guest 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.

## 3.4 Resource Negotiation Issues

**Expiration and unilateral reclaim.** Leases provide a mutually agreed upon termination time that mitigates the need for an explicit notification from the authority to the service manager at termination time as implemented in node operating systems using a weaker give away/take away abstraction. As long as actors loosely synchronize their clocks, a service manager 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 service manager does not extend it by presenting a fresh ticket from the broker in an *extendLease*. The decision of when to reclaim 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. Unilateral resource teardown by an authority is reminiscent of Exokernel's abort protocol.

| *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 1: Guest handler interface for service manager plugins.

**Renew/extend.** The service manager may request to extend/renew leases before they expire. The service manager core will automatically renew a lease object marked as "renewable" (automated meter feeding). 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. 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.

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

**Close/Vacate.** Service managers may require a way to vacate their leases and release their resources for use by other guests. It is future work to add the interface for service managers to vacate a lease. Currently, service managers may indicate to the authority that they are finished with a lease by calling `close` on the authority, but the authority does not propagate the relinquished resource information to the broker for allocation to other service managers.

## 3.5 Guest Handlers

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

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 1 and summarized below.

- Upon receiving a lease notification from the authority the service manager invokes a *join* action to notify the guest of each new resource unit in the lease.
- The service manager core may invoke a *modify* action for various property if the subtype properties or unit properties change on a lease extension. [How is this checked/triggered?]
- Before the end of a lease, the service manager invokes a *leave* action for each resource unit to give the guest an opportunity to gracefully vacate the resources. Once all leave actions for a lease are complete the service manager notifies the corresponding authority.

The authority has similar upcall handler interfaces for *setup* and *teardown* on each logical resource unit. The *join* is useful for post-install actions that are specific to a guest. The *join* and *leave* guest event handlers may interact with drivers to reconfigure the guest for membership changes. For example, the handlers could link to standard entry points of a Group Membership Service that maintains a consistent view of membership

| *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 2: 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.

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.

Shirako/COD provides a generic implementation of the handler interfaces that allows handlers to be scripted using Ant [**?**]. 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. Section 4.3 discusses handler scripting is more detail.

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. [Note: a handler may involve multiple physical resources, possibly covered by separate leases....how does it get the properties for other units? As in the SGE example of "who is my master?".] The handler may also mark properties, e.g., to represent the return status or other information returned from a configuration action. [Need to be clear about the rules for this.]

## 3.6 Guest Lease Event Handlers

Guest controllers may want to perform operations before or after lease state changes. For example, before a ticket is sent to the provider to be redeemed, the guest controller may want to attach additional configuration properties to indicate how its resources should be configured (e.g., VM image location). Another example is that a guest controller may want to clean the machines in a lease that is ending before the lease closes. To notify the controller of these events, the Shirako toolkit supports an `EventHandler` (not shown in Figure 3). This object is an upcall target that is called on state changes; guest controllers may define one or more handler entry points for the upcalls they wish to receive. *Note:* guest lease event handlers are synchronized with lease state changes by the core and should never block for any reason. It is expected that most guest controllers will not define their own lease event handlers. [Check the locking model with Irwin.]

Another use of the `EventHandler` is for lease extensions. If a lease is marked as *renewable* the service manager shell automatically issues an extend request to the broker for the duration of the previous lease. However, if a guest controller wishes to have complete control over lease extensions, it may register with the `EventHandler` to receive a call before the service manager issues an `extendTicket` call to the broker. When the guest controller receives the event call, it may choose to modify request attributes, such as the lease term or even mark the request as nonrenewable so that its term is not extended.

Each lease uses a *guest lease handler*, which the service manager invokes once when the state of a lease changes. The guest lease handler exports an interface to functions defined by a service manager for each lease. The core upcalls a guest lease handler function before various lease state changes. These are invoked per-lease, and not per-resource unit. The guest lease handler exposes lease state transitions to service managers, which may take the opportunity to perform lease-wide actions, such as examining the load of a guest to determine whether or not to extend or modify a renewing lease.

Guest lease handlers enable service managers to respond to resource flexing or modification in aggregate, rather than responding at a per-resource granularity using the guest handler. Table 2 lists examples of how

| **Request Properties:** passed in a lease request or response | |
|---|---|
| `elasticSize` | Reservation will accept 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` | Reservation may start at a later time than in the request |
| `elasticTime` | Reservation may be for less time 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 group. [Check... missing from Laura's thesis.] |

Table 3: Example properties that may be set on a resource request for standard broker policies.

service managers use different functions of the guest lease handler. We have not found the need to implement a guest slice handler that allows service managers to expose functions for slice-wide state transitions (*i.e.*, upcalled once if *any* lease in a slice changes state).

[Are ticketSatisfies and leaseSatisfies part of the lease event handler interface? They should be.]

## 3.7 Drivers

Guest handler tasks for nodes may invoke actions that run on the leased resource component itself. Handlers invoke guest driver interfaces to attach and detach allocated resources to and from the guest. A guest driver is a wrapper or package of scripts that defines a set of control actions on a guest instance. A guest driver exports standard actions to instantiate and configure the guest, probe its status, and also to attach and detach managed resources (servers, storage) to and from the service, and to notify it of VM resizing or migration.

Many resource units can be viewed as *nodes*. 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. An error may trigger a failure policy on the service manager that resets a resource or re-assigns the resource to an equivalent machine.

## 3.8 Property Examples

Now we discuss some uses of the properties. These are common conventions that should be supported by a range of resource controllers.

**Request properties.** Table 3 provides example request properties. These examples provide simple ways for the guest controller to indicate a range of possible resource levels and times for the request. Combining requests together into a A consumer can also 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 3). 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 do not have them.

| **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 4: Selected properties used by Cluster-on-Demand, and sample values.

**Configuring virtual machines.** Table 4 lists some of the important node properties for COD. These property names and legal values are conventions among the guest and resource handlers for COD-compatible service managers and authorities. Several configuration properties allow a service manager to guide authority-side configuration of machines.

- *Image selection.* The service manager passes a string to identify an operating system configuration 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 site and guest exchange keys via properties to enable secure, programmatic access to the leased nodes, as described below.

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 using some 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 controller runs with an identity and at least one asymmetric key pair. These must be installed somewhere as defined by the service manager framework. Without loss of generality we assume there is only one. Guest nodes also have an asymmetric key pair.

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

For example, the KeyMaster protocol is implemented using plugins that interact through the property lists. The guest controller module stores a hash of its public key as a configuration property. A resource driver on the authority side passes the hash into the image of a guest node 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 keymaster protocol, passing its public key to the keymaster in the guest node, 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 guest 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.

## 3.9 Packaging and Configuration

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.

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.

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.

Various tools, classes, and methods define actor configuration and how actors associate with each other. This may be accomplished through an automated policy, or manually by a system administrator. For example, a system administrator may call a method, such as `addBroker` or `addGuest`, to associate actors and give them the authorization to use that actor as a broker or guest. The Shirako system supports a registry for actors to learn about each other.

The system also 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 guest 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.

## 3.10 Instrumentation

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 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.]

## 3.11 Lease Groups

Shirako provides a grouping primitive to sequence guest handler invocations. This useful for complex guests with configuration dependencies among their components. Since the service manager specifies properties

| 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 5: Example properties a guest controller may set on a query to learn more about a broker and a description of what the broker may return as a result of the query.

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 authority (*setup*) or within the service manager (*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 there is a redeem predecessor and the service manager has not yet received a lease for it, 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. Also, if a join predecessor exists, the service manager holds the lease in a blocked state and does not fire its *join* until the join predecessor is active. In both cases, the core upcalls a lease event handler before transitioning out of the blocked state; the upcall gives the controller an opportunity to manipulate properties on the lease before it fires, or to impose more complex trigger conditions.

It is also possible for a guest handler to access the properties of its predecessor leases. [Document: missing from Irwin's thesis.]

### 3.12 Queries

Queries are generic interfaces between actors. Actors use queries as a mechanism to learn about another actor. As an example, before making a request to a broker, a service manager 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 wish to know more about what request properties the broker's controller policy accepts and understands; for example, some brokers may understand an `elasticSize` property, whereas others may not. Table 5 provides examples of queries a service manager may make to a broker to gain information about its policy.

# 4 Implementation Details

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. Multiple actors may inhabit the same JVM and interact through local procedure calls.

Actors are externally clocked to eliminate any dependency on absolute time. Time-related state transitions are driven by a *virtual clock* that advances in response to external *tick* calls. All actors retain and cache lease state in memory, in part to enable lightweight emulation-mode experiments without an external repository.

In the past we have claimed that the Shirako leasing core is compact and is understandable by one person. That may still be true, but the plugin structure and management interfaces have lots of moving parts. Also, there are various support modules layered on top of the basic plugin interfaces. External policy modules and calendar support represent nearly half of the code size.

This section has some specific notes about the code. It is still pretty rough.

## 4.1 Plugin Architecture

This section has some notes on classes and interface implemented in the packages `orca.shirako.*`. These Java source files are in `./shirako/src/main/java/orca/shirako`. The relevant source file names are derived from the class names and directory hierarchy given here.

Synchronization for plugin modules is a delicate balancing act. The Shirako core maintains a core "manager lock" rooted in `kernel/Kernel` that locks all core data structures. The original principle was that the manager lock would never be held while running code in any extension (plugin). However, that choices limits extensibility for functions that are tightly integrated with the core.

This led us to a sort of two-level plugin architecture. The toolkit is designed to support *user plugins* implemented by toolkit users. But it also has hooks for *core extensions* that are replaceable extensions to the toolkit itself. These plugin interfaces are less public: users should not need to understand them, and they do not need to be documented as well. They are typically handled in layered policy classes that are part of the Shirako toolkit.

Guest controller clock handlers (`formulateBids`, which could just be called `tick`) and guest handlers are user plugins. User plugins may block or delay arbitrarily. If a user plugin blocks or delays, it may "get behind": it might fail to configure its own leases before they expire, or it might implement some adaptive policy that does not adapt quickly enough to keep up. But it will not harm the system.

Any extension method upcalled with the manager lock is a core extension. It must never block on any other resource, or it can stall or deadlock the core. Thus these core extensions should not be implemented by people who don't know what they are doing. Which means we should not tell as many people about them.

Shirako developers have drifted a bit and have been too casual about the distinction between core plugins and user plugins. This will lead to chaos if it continues. In particular, some of the management interfaces for external views and facility control are a bit relaxed about whether the manager lock is held or not. Guest lease event handlers are core extensions, but have been implicitly represented as user plugins. Also, `ticketSatisfies` and `leaseSatisfies` are invoked with the lock on: they should probably be considered part of the guest lease event handler interface. But some of the text represents them as user plugins. This is something we need to clean up.

COD is a core extension implementing the interface. A base class for `IShirakoPlugin` is available in `plugins/ShirakoPlugin`. COD deals with resource sets that consist of groups of nodes. Nodes are concrete resource objects with particular state and behaviors: for example, every node has at least one IP address. The COD plugin implements both client (service manager) and server (authority) sides of COD. It is not called with the manager lock held, but it is not well-documented and is not intended to be replaced by toolkit users. It is open question whether network extensions will require changes to the COD plugin, or a new replacement for the COD plugin, or another new implementation of `ShirakoPlugin` that somehow exists alongside COD within the actors.

The guest handlers are actually invoked from COD, rather than directly from the Shirako core. In fact, to confuse matters further, these are plugins to COD rather than to Shirako per se. Or something. The `ShirakoPlugin` architecture is quite convoluted and needs to be documented.

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 `IServiceManagerPolicy`.
- `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.

## 4.2 Controller Structure and Management

Controllers and handlers are registered and selected according to the types and attributes of the specific resource units assigned to a lease.

Controllers must be installable into Shirako, but they also need to be manageable through the Automat Web portal and other configuration machinery that is (stricly speaking) outside of Shirako.

Every 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.

The various features for dynamically registering and uploading them require conformance with certain classes and interfaces that are defined outside of Shirako.

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/mai`

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/s`

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 but it doesn't do anything. [Document how the upcalls described in this document get into the controller.]

## 4.3 Handler Invocation

[Say where the hook is to invoke Ant handlers.] 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”.

# 5 Example Guests

We illustrate by outlining two guest controllers for job managers that are included as reference Shirako/COD controllers in the standard release. These are services that export tools or network interfaces to submit jobs. They control a set of nodes as a unified cluster resource for running the jobs. Adding a guest controller makes it possible to instantiate these systems on demand on an ORCA substrate. This makes it possible to run multiple isolated instances on shared resources, provision and/or configure the instances independently to serve different user communities, adapt the resources assigned to each instance according to various policies as needs change, and share the server substrate with other dynamically provisioned services.

## 5.1 GridEngine

Sun GridEngine [?], also called SGE, is a widely used batch job middleware system for networked clusters under common administrative authority. SGE assumes that all nodes run the same software stack and have the same logical file tree, so any job can run correctly on any node. GridEngine runs with a single *master* server and a dynamic pool of *worker* nodes. Submitted jobs queue at the master, which schedules them according to its internal scheduling policy and launches them on the worker nodes.

In a typical GridEngine deployment, a single master runs a scheduler (*sge_schedd*) that dispatches submitted tasks across an *active set* of workers. Users submit tasks by executing the *qsub* command, which transmits the information necessary to execute a task to the master. The workers run with a common set of user identities and a shared network file volume mounted through NFS. The NFS volume includes the GridEngine distribution and master status files (the SGE_ROOT directory) and all program and data files for the user tasks. In our prototype, either the master machine acts as the NFS server or the master and each worker mount a file volume exported by an external NFS server.

The controller obtains separate leases for masters and workers. The lease for the master may be long (*e.g.*, weeks) to ensure stability and continuous operation while each lease for one or more workers may be short to provide the flexibility to adapt GridEngine’s machine allotment to the number of queued tasks.

There are two guest handlers: one to configure the master machine and one to configure each worker machine. Each handler is a script that executes a sequence of GridEngine administrative commands and starts and stops the proper GridEngine daemons.

It uses lease groups to sequence the instantiation of the guest handlers for the master and each worker and bind worker machines to the master. To group each worker lease with the master lease, the service manager sets the master lease as the predecessor of each worker lease. The predecessor relationship not only ensures that the master is active before each worker, it allows the guest handler for each worker to reference the properties of the master lease. In particular, the guest handler for a worker must know the master's IP address to register itself with the master.

The *join* handler for the master installs the GridEngine software to the master machine then executes the *sge_master* daemon. It starts an NFS server and exports a file volume to each worker machine.

The *leave* handler for the master lease stops the *sge_master* daemon and removes the installed GridEngine software.

As an alternative, the service manager could also lease an additional machine to act as the NFS server; in this case, the lease for the NFS server would be a predecessor of the master lease and *join* for the master lease would mount the appropriate file volume by referencing a property of the NFS server lease.

To add a worker, *join* executes the GridEngine *qconf* command on the master with a standard template to activate a machine by its domain name and establish a task queue for the worker. In order to execute *qconf* on the master, the guest handler of the worker must know the master's IP address; the predecessor relationship ensures that the information is available as a property the guest handler of the worker may reference. In our prototype, guest handlers reference unit properties of the predecessor lease by prepending `predecessor.` to the name of a unit property. If there are multiple units within the predecessor lease the property includes a comma-separated list of the values of each unit in the predecessor lease.

For example, to reference the master's IP address the guest handler for a worker uses the property `predecessor.host.privIPa` After enabling the queue on the master, *join* for the worker's guest handler executes the GridEngine daemon processes—*sge_commd* and *sge_execd*—on the worker using a remote execution engine, such as *ssh*. If the master acts as an NFS server *join* also mounts the appropriate file volume on the worker. To remove a machine, *leave* executes GridEngine commands—*qconf* and *qmod*— on the master to disable the worker's task queue, reschedule any tasks on the queue, and destroy the queue. Finally, *leave* stops *sge_commd* and *sge_execd* on the worker to deactivate the machine, and, if necessary, unmount any file volumes.

### 5.1.1 Adaptation Policy

A simple adaptation policy monitors the size of the master's task queue over time. It uses queue size as a feedback to drive dynamic adaptation. It inspects the task queue on lease extensions to determine whether to shrink, grow, or maintain the lease's resources. If a service manager shrinks a lease it must select victim machines; if it grows a lease it must determine how many additional resources to request.

We configure GridEngine to schedule at most one task on each active worker and assume that that tasks are sequential, compute-bound, and run for longer than the reconfiguration times. Slivers are fixed-size and do not change: we only adjust the number of workers in the pool.

The policy is to request a new worker for every $X$ pending tasks and relinquishes any idle worker at the end of its lease. The policy attempts to ensure a minimum idle reserve of $k$ worker machines by extending any worker leases, even if the workers are idle, when the active set of workers falls below $k$. We set $X = 1$ to enable the GridEngine service manager to respond quickly to newly submitted task and $k = 1$ to ensure that at least one worker is active.

The controller checks the size of the task queue by executing the *qstat* command using a secure `ssh` channel to obtain a list of queues and the tasks scheduled to them. This should be done with a driver. In truth, all guest services should be programmatically manageable: CLIs are so yesterday.

The controller uses `onExtendTicket` to inspect the task queue for each worker covered by an expiring worker lease. If a worker is idle, it decrements the worker lease unit count by one and appends the workers name to the value of a configuration property (*i.e.*, `lease.victims`). The authority uses the names to select victim machines to teardown on a shrinking lease. Our prototype uses the IP address to name victim machines. [Presumably the monitoring is done by a separate thread since `onExtendTicket` is not permitted to block. Also, how does any lease extension know which workers are covered by the lease, so that it can check their queues?]

If the controller detects one or more queued tasks [how/when] it attempts to obtain a new lease to grow. An alternative is to decrease responsiveness by waiting until the service manager extends an existing lease; at extension time the service manager may grow the lease to satisfy the new task. The overhead from fragmenting resources across multiple leases is minimal, and, in this case, justified to achieve a quick response to spikes in the number of tasks.

### 5.1.2 Anatomy of the SGE Controller

The SGE controller runs within an actor that is configured to use `ServiceManagerCalendarPolicy`. It instantiates a guest lease event handler and registers it with the policy.

The SGE factory creates a slice and a controller object, links them together, greates a manager object for the controller, creates a portal plugin descriptor, and registers the slice and and manager object with the portal plugin descriptor.

No base class/interface for management proxies. Local implementation just passes through to controller-specific manager object and the manager object mostly just passes through the controller object, maybe some registers and exception handling.

The controller object implements IController and it has an internal protected class that extends EventHandler. The EventHandler subclass only handles onBeforeExtendTicket. It calls the poller object to identify any idle workers. It sets set them as victims on the ICodClientReservation and adjusts unit count.

The controller itself has a bid(cycle) function and sets up reservations and calls sm.demand(r). Some ugliness grabbing a resource set and casting the concrete set to a NodeGroup...why needed not clear. Lots of code to set resource subtype properties on new reservations. These are hard coded sliver sizing. Nasty.

The poller is in a "batch" object. It asks questions like how many queued/running/completed jobs. It does ssh, and it has hardcoded paths in it. It has a runnable SgeProbe that does the ssh's. [Check the concurrency model here.]

The controller code shows that there are various ways to get at the different properties which is a bit tricky...needs document or cleanup.

- - ResourceProperties.setMin/Max (operates directly on Properties)
- - PropList.setProperty (operated directly on Properties)
- - PropertiesManager.SetElasticSize(rset) etc.
- orca.policy.core.util.PropertiesManager
- - master.setLocalProperty etc. etc....master.setImage

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?

## 5.2 Jaws

Jaws is a new type of batch scheduler built to integrate with a Shirako service manager. Rather than provide middleware to run jobs on a homogeneous worker pool, it builds workers to order for each job. This approach enables customized software stacks, sliver provisioning, and (in the future) checkpoint/migration on a per-job basis. The lease brokering policy acts as the job scheduler by scheduling a separate set of leases specifically for each job.

Jaws exposes a `submit` interface similar to GridEngine [?]. The service method for this interface runs in the same JVM as its controller.

Jaws submits one or more ticket requests for each submitted task. The service manager interface includes the `queueRequest` function, which Jaws uses to notify the service manager of a resource request using a local procedure call [explain]. The task descriptor specifies the desired sliver size [or sizes] directly. The Jaws controller marks each lease request as `deferrable` to allow the broker to schedule it.

[It is not clear how the lease terms are set. The submitted task descriptor estimates an upper bound on completion time. David Irwin's thesis says Jaws derives the requested lease term from the estimate, but also says it requests and continually renews short leases to minimize waste if the task completes early.]

The Jaws prototype uses Plush to execute tasks. For each submitted job, Jaws instantiates a Plush master on a unique port to handle the task's execution. It tags the request with the port of the Plush master, its IP address, and the location of the Plush descriptor for the task. Jaws registers a callback with the Plush master that notifies Jaws when a task completes or fails. In this case, failure implies the return of a non-zero exit code from any process in the task workflow. Jaws does not address task-specific failure semantics.

The *join* handler adds a machine to the Plush master instance managing the task's execution using the Plush master's XML-RPC interface. When all resources are ready a lease event handler fires (`onActiveLease`) and triggers the Plush master to run the task. [This is probably buggy synchronization.] On receiving a callback that signals task completion or failure from the Plush master, Jaws disables renewal of the job's leases by setting a shared variable; the controller queries this state from its `onExtendTicket` handler to determine whether or not to renew a lease.

## 6 Some Future Work and Objectives

Define basic default policies and property conventions for cloud computing with best-effort slivers and basic fixed-size (non-adjustable) slivers, with both upper and lower share bounds, and guest-provided directives for colocation. The only form of guest flexing supported will be to change the unit count (number of VMs).

Add an authority policy that assigns and migrates to balance workload consistent with the colocation directives. The authority interface should be general enough to enable us of the VMware load balancing. A simpler version would simply assign to a fixed set of preallocated slivers, like the original COD (the Xen-Control resource controller). That one could work with EC2 or VMware at the back end, with the right drivers.

Add basic cluster-wide proportional share in the broker, using Winks. It should work across multiple sites. No broker-driven migration.

Also introduce human-in-the-loop controllers that request permission from the operator on both the broker and authority. These could be used instead of the reference policies, or in addition to.

The prototype should support installable guests. And we should have functioning reference implementations of the guests we have already done: SGE, Jaws, Hadoop, multitier. Jaws should launch rPath and VCL images. Can it run VCL as a service?

This reference prototype with these features should be the starting point for the work on adaptive slivering, which is restarting. Start by exploring the site/broker tension for these reference policies.

The Orca leasing abstractions allow contract renegotiations to occur at any time during the term of the contract. However, to simplify the protocol and to minimize actor communication, the Shirako toolkit constrains the ability of a service manager to modify its lease. A service manager may increase the duration of its lease, and these lease extensions (or *renews*) must be requested before the end of a lease. Lease holders may negotiate limited changes to the lease at renewal time, including *flexing* the size of their allotments (e.g., the number of resource units and the size of VM slivers). Since Shirako combines lease updates and extensions, the `extendTicket` and `extendLease` interfaces are used for lease flexes and extends.

Since Shirako does not allow lease renegotiations outside of lease renewals, service managers decide if they will react to growing demand by requesting additional leases or if they will request short leases so they may renew their lease for more resources if their guest demand grows over time. In the future it may be useful to incorporate higher-level contracts that allow service managers to negotiate long-term resource guarantees across multiple lease renewals [**?**]. This would allow for many short renewals and give the service manager additional flexibility to adjust resources to dynamic application behavior while still providing basic resource assurances.

## References

[1] 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.

[2] 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.

[3] 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.

[4] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *Proceedings of the USENIX Technical Conference*, June 2008.

[5] 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.