Sharing Networked Resources with Brokered Leases

David Irwin, Jeffrey Chase, Laura Grit, Aydan Yumerefendi, and David Becker
Duke University
{irwin,chase,grit,aydan,becker}@cs.duke.edu

Kenneth G. Yocum
University of California, San Diego
kyocum@cs.ucsd.edu

Abstract

This paper presents the design and implementation of Shirako, a system for on-demand leasing of shared networked resources. Shirako is a prototype of a service-oriented architecture for resource providers and consumers to negotiate access to resources over time, arbitrated by brokers. It is based on a general lease abstraction: a lease represents a contract for some quantity of a typed resource over an interval of time. Resource types have attributes that define their performance behavior and degree of isolation.

Shirako decouples fundamental leasing mechanisms from resource allocation policies and the details of managing a specific resource or service. It offers an extensible interface for custom resource management policies and new resource types. We show how Shirako enables applications to lease groups of resources across multiple autonomous sites, adapt to the dynamics of resource competition and changing load, and guide configuration and deployment. Experiments with the prototype quantify the costs and scalability of the leasing mechanisms, and the impact of lease terms on fidelity and adaptation.

1 Introduction

Managing shared cyberinfrastructure resources is a fundamental challenge for service hosting and utility computing environments, as well as the next generation of network testbeds and grids. This paper investigates an approach to networked resource sharing based on the foundational abstraction of *resource leasing*.

We present the design and implementation of Shirako, a toolkit for a brokered utility service architecture.[1] Shirako is based on a common, extensible resource leasing abstraction that can meet the evolving needs of several strains of systems for networked resource sharing—whether the resources are held in common by a community of shareholders, offered as a commercial hosting service to paying customers, or contributed in a reciprocal fashion by self-interested peers. The Shirako architecture reflects several objectives:

- *Autonomous providers.* A provider is any administrative authority that controls resources; we refer to providers as *sites*. Sites may contribute resources to the system on a temporary basis, and retain ultimate control over their resources.
- *Adaptive guest applications.* The clients of the leasing services are hosted application environments and managers acting on their behalf. We refer to these as *guests*. Guests use programmatic lease service interfaces to acquire resources, monitor their status, and adapt to the dynamics of resource competition or changing demand (e.g., flash crowds).
- *Pluggable resource types.* The leased infrastructure includes edge resources such as servers and storage, and may also include resources within the network itself. Both the owning site and the guest supply type-specific configuration actions for each resource; these execute in sequence to setup or tear down resources for use by the guest, guided by configuration properties specified by both parties.
- *Brokering.* Sites delegate limited power to allocate their resource offerings—possibly on a temporary basis—by registering their offerings with one or more brokers. Brokers export a service interface for guests to acquire resources of multiple types and from multiple providers.
- *Extensible allocation policies.* The dynamic assignment of resources to guests emerges from the interaction of policies in the guests, sites, and brokers. Shirako defines interfaces for resource policy modules at each of the policy decision points.

Section 2 gives an overview of the Shirako leasing services, and an example site manager for on-demand cluster sites. Section 3 describes the key elements of the system design: generic property sets to describe resources and guide their configuration, scriptable configuration ac-

[1]This research is supported by the National Science Foundation through ANI-0330658 and CNS-0509408, and by IBM, HP Labs, and Network Appliance. Laura Grit is a National Physical Science Consortium Fellow.

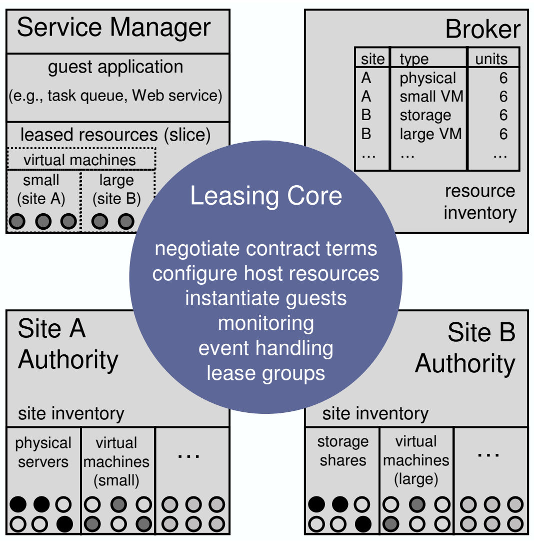


Figure 1: An example scenario with a guest application acquiring resources from two cluster sites through a broker. Each resource provider site has a server (site authority) that controls its resources, and registers inventories of offered resources with the broker. A service manager negotiates with the broker and authorities for leases on behalf of the guest. A common lease package manages the protocol interactions and lease state for all actors. The Shirako leasing core is resource-independent, application-independent, and policy-neutral.

tions, support for lease extends with resource flexing, and abstractions for grouping related leases. Section 4 summarizes the implementation, and Section 5 presents experimental results from the prototype. The experiments evaluate the overhead of the leasing mechanisms and the use of leases to adapt to changes in demand. Section 6 sets Shirako in context with related work.

2 Overview

Shirako's leasing architecture derives from the SHARP framework for secure resource peering and distributed resource allocation [13]. The participants in the leasing protocols are long-lived software entities (*actors*) that interact over a network to manage resources.

- Each guest has an associated *service manager* that monitors application demands and resource status, and negotiates to acquire leases for the mix of resources needed to host the guest. Each service manager requests and maintains leases on behalf of one or more guests, driven by its own knowledge of application behavior and demand.
- An *authority* controls resource allocation at each resource provider site or domain, and is responsible for enforcing isolation among multiple guests hosted on the resources under its control.
- *Brokers* (agents) maintain inventories of resources offered by sites, and match requests with their resource supply. A site may maintain its own broker to keep control of its resources, or delegate partial, temporary control to third-party brokers that aggregate resource inventories from multiple sites.

These actors may represent different trust domains and identities, and may enter into various trust relationships or contracts with other actors.

2.1 Cluster Sites

One goal of this paper is to show how dynamic, brokered leasing is a foundation for resource sharing in networked clusters. For this purpose we introduce a cluster site manager to serve as a running example. The system is an implementation of Cluster-On-Demand (COD [7]), rearchitected as an authority-side Shirako plugin.

The COD site authority exports a service to allocate and configure *virtual clusters* from a shared server cluster. Each virtual cluster comprises a dynamic set of nodes and associated resources assigned to some guest at the site. COD provides basic services for booting and imaging, naming and addressing, and binding storage volumes and user accounts on a per-guest basis. In our experiments the leased virtual clusters have an assurance of performance isolation: the nodes are either physical servers or Xen [2] virtual machines with assigned shares of node resources.

Figure 1 depicts an example of a guest service manager leasing a distributed cluster from two COD sites. The site authorities control their resources and configure the virtual clusters, in this case by instantiating nodes running a guest-selected image. The service manager deploys and monitors the guest environment on the nodes. The guest in this example may be a distributed service or application, or a networked environment that further subdivides the resources assigned to it, e.g., a cross-instititutional grid or content distribution network.

The COD project began in 2001 as an outgrowth of our work on dynamic resource provisioning in hosting centers [6]. Previous work [7] describes an earlier COD prototype, which had an ad hoc leasing model with built-in resource dependencies, a weak separation of policy and mechanism, and no ability to delegate or extend provisioning policy or to coordinate resource usage across federated sites. Our experience with COD led us to pursue a more general lease abstraction with distributed, accountable control in SHARP [13], which was initially prototyped for PlanetLab [4]. We believe that dynamic leasing is a useful basis to coordinate resource sharing for other systems that create distributed virtual execution environments from networked virtual machines [9, 17, 18, 19, 20, 25, 26, 28].

2.2 Resource Leases

The resources leased to a guest may span multiple sites and may include a diversity of resource types in differing quantities. Each SHARP resource has a *type* with associ-

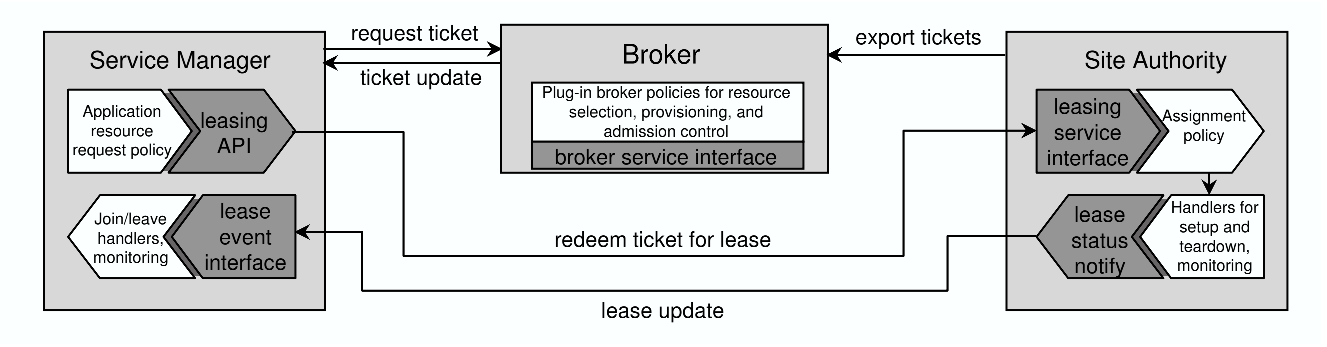

request ticket
ticket update
export tickets
Service Manager
Application resource request policy
leasing API
Join/leave handlers, monitoring
lease event interface
Broker
Plug-in broker policies for resource selection, provisioning, and admission control
broker service interface
Site Authority
leasing service interface
Assignment policy
lease status notify
Handlers for setup and teardown, monitoring
redeem ticket for lease
lease update

Figure 2: Summary of protocol interactions and extension points for the leasing system. An application-specific service manager uses the lease API to request resources from a broker. The broker issues a ticket for a resource type, quantity, and site location that matches the request. The service manager requests a lease from the owning site authority, which selects the resource units, configures them (*setup*), and returns a lease to the service manager. The arriving lease triggers a *join* event for each resource unit joining the guest; the join handler installs the new resources into the application. Plug-in modules include the broker provisioning policy, the authority assignment policy, and the *setup* and *join* event handlers.

ated attributes that characterize the function and power of instances or *units* of that type. Resource units with the same type at a site are presumed to be interchangeable.

Each lease binds a set of resource units from a site (a *resource set*) to a guest for some time interval (*term*). A lease is a contract between a site and a service manager: the site makes the resources available to the guest identity for the duration of the lease term, and the guest assumes responsibility for any use of the resources by its identity. In our current implementation each lease represents some number of units of resources of a single type.

Resource attributes define the performance and predictability that a lease holder can expect from the resources. Our intent is that the resource attributes quantify capability in an application-independent way. For example, a lease could represent a *reservation* for a block of machines with specified processor and memory attributes (clock speed etc.), or a storage partition represented by attributes such as capacity, spindle count, seek time, and transfer speed. Alternatively, the resource attributes could specify a weak assurance, such as a best-effort service contract or probabilistically overbooked shares.

2.3 Brokers

Guests with diverse needs may wish to acquire and manage multiple leases in a coordinated way. In particular, a guest may choose to aggregate resources from multiple sites for geographic dispersion or to select preferred suppliers in a competitive market.

Brokers play a key role because they can coordinate resource allocation across sites. SHARP brokers are responsible for *provisioning*: they determine *how much* of each resource type each guest will receive, and *when*, and *where*. The sites control how much of their inventory is offered for leasing, and by which brokers, and when. The site authorities also control the *assignment* of specific resource units at the site to satisfy requests approved by the brokers. This decoupling balances global coordination (in the brokers) with local autonomy (in the site authorities).

Figure 2 depicts a broker's role as an intermediary to arbitrate resource requests. The broker approves a request for resources by issuing a *ticket* that is redeemable for a lease at some authority, subject to certain checks at the authority. The ticket specifies the resource type and the number of units granted, and the interval over which the ticket is valid (the term). Sites issue tickets for their resources to the brokers; the broker arbitration policy may subdivide any valid ticket held by the broker. All SHARP exchanges are digitally signed, and the broker endorses the public keys of the service manager and site authority. Previous work presents the SHARP delegation and security model in more detail, and mechanisms for accountable resource contracts [13].

2.4 System Goals

Shirako is a toolkit for constructing service managers, brokers, and authorities, based on a common, extensible leasing core. A key design principle is to factor out any dependencies on resources, applications, or resource management policies from the core. This decoupling serves several goals:

- The resource model should be sufficiently general for other resources such as bandwidth-provisioned network paths, network storage objects, or sensors. It should be possible to allocate and configure diverse resources alone or in combination.
- Shirako should support development of guest applications that adapt to changing conditions. For example, a guest may respond to load surges or resource failures by leasing additional resources, or it may adjust to contention for shared resources by deferring work or reducing service quality. Resource sharing

expands both the need and the opportunity for adaptation.

- Shirako should make it easy to deploy a range of approaches and policies for resource allocation in the brokers and sites. For example, Shirako could serve as a foundation for a future resource economy involving bidding, auctions, futures reservations, and combinatorial aggregation of resource bundles. The software should also run in an emulation mode, to enable realistic experiments at scales beyond the available dedicated infrastructure.

Note that Shirako has no globally trusted core; rather, one contribution of the architecture is a clear factoring of powers and responsibilities across a dynamic collection of participating actors, and across pluggable policy modules and resource drivers within the actor implementations.

3 Design

Shirako comprises a generic leasing core with plug-in interfaces for extension modules for policies and resource types. The core manages state storage and recovery for the actors, and mediates their protocol interactions. Each actor may invoke primitives in the core to initiate lease-related actions at a time of its choosing. In addition, actor implementations supply plug-in extension modules that are invoked from the core in response to specific events. Most such events are associated with resources transferring in or out of a *slice*—a logical grouping for resources held by a given guest.

Figure 2 summarizes the separation of the core from the plugins. Each actor has a *mapper* policy module that is invoked periodically, driven by a clock. On the service manager, the mapper determines when and how to redeem existing tickets, extend existing leases, or acquire new leases to meet changing demand. On the broker and authority servers, the mappers match accumulated pending requests with resources under the server's control. The broker mapper deals with resource provisioning: it prioritizes ticket requests and selects resource types and quantities to fill them. The authority mapper assigns specific resource units from its inventory to fill lease requests that are backed by a valid ticket from an approved broker.

Service managers and authorities register *resource driver* modules defining resource-specific configuration actions. In particular, each resource driver has a pair of event handlers that drive configuration and membership transitions in the guest as resource units transfer in or out of a slice.

- The authority invokes a *setup* action to configure (*prime*) each new resource unit assigned to a slice by the mapper. The authority issues the lease when all of its setup actions have completed.
- The service manager invokes a *join* action to notify the guest of each new resource unit. Join actions are driven by arriving lease updates.
- *Leave* and *teardown* actions close down resource units at the guest and site respectively. These actions are triggered by a lease expiration or resource failure.

3.1 Properties

Shirako actors must exchange context-specific information to guide the policies and configuration actions. For example, a guest expresses the resources requested for a ticket, and it may have specific requirements for configuring those resources at the site. It is difficult to maintain a clean decoupling, because this resource-specific or guest-specific information passes through the core.

Shirako represents all such context-specific information in *property lists* attached as attributes in requests, tickets, and leases. The property lists are sets of *(key, value)* string pairs that are opaque to the core; their meaning is a convention among the plugins. Property sets flow from one actor to another and through the plugins on each of the steps and protocol exchanges depicted in Figure 2.

- *Request properties* specify desired attributes and/or value for resources requested from a broker.
- *Resource properties* attached to tickets give the attributes of the assigned resource types.
- *Configuration properties* attached to redeem requests direct how the resources are to be configured.
- *Unit properties* attached to each lease define additional attributes for each resource unit assigned.

3.2 Broker Requests

The Shirako prototype includes a basic broker mapper with several important features driven by request properties. For example, a service manager may set request properties to define a range of acceptable outcomes.

- Marking a request as `elastic` informs the broker that the guest will accept fewer resource units if the broker is unable to fill its entire request.
- Marking a request as `deferrable` informs the broker that the guest will accept a later start time if its requested start time is unavailable; for example, a service manager may request resources for an experiment, then launch the experiment automatically when the resources are available.

Request properties may also express additional constraints on a request. For example, the guest may mark a set of ticket requests as members of a *request group*, indicating that the broker must fill the requests atomically, with the same terms. The service manager tags one of its lease requests as the group leader, specifying a unique `groupID` and a `leaseCount` property giving the number of requests in the group. Each request has a `groupID` property identifying its request group, if any.

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*	`cats01.cats.cod.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 1: Selected properties used by Cluster-on-Demand, and sample values.

When all leases for a group have arrived, the broker schedules them for a common start time when it can satisfy the entire group request. Because request groups are implemented within a broker—and because SHARP brokers have allocation power—a co-scheduled request group can encompass a variety of resource types across multiple sites. The default broker requires that request groups are always `deferrable` and never `elastic`, so a simple FCFS scheduling algorithm is sufficient.

The request properties may also guide resource selection and arbitration under constraint. For example, we use them to encode bids for economic resource management [16]. They also enable attribute-based resource selection of types to satisfy a given request. A number of projects have investigated the matching problem, most recently in SWORD [22].

3.3 Configuring Virtual Clusters

The COD plugins use the configuration and unit properties to drive virtual cluster configuration (at the site) and application deployment (in the guest). Table 1 lists some important properties used in COD. These property names and legal values are conventions among the package classes for COD service managers and authorities.

To represent the wide range of actions that may be needed, the COD resource driver event handlers are scripted using *Ant* [1], an open-source OS-independent XML scripting package. Ant scripts invoke a library of packaged tasks to execute commands remotely and to manage network elements and application components. Ant is in wide use, and new plug-in tasks continue to become available. A Shirako actor may load XML Ant scripts dynamically from user-specified files, and actors may exchange Ant scripts across the network and execute them directly. When an event handler triggers, Ant substitutes variables within the script with the values of named properties associated with the node, its containing lease, and its containing slice.

The *setup* and *teardown* event handlers execute within the site's trusted computing base (TCB). A COD site authority controls physical boot services, and it is empowered to run commands within the control domain on servers installed with a Xen hypervisor, to create new virtual machines or change the resources assigned to a virtual machine. The site operator must approve any authority-side resource driver scripts, although it could configure the actor to accept new scripts from a trusted repository or service manager.

Several configuration properties allow a COD service manager to guide authority-side configuration.

- *OS boot image selection*. The service manager passes a string to identify an OS configuration from among a menu of options approved by the site authority as compatible with the machine type.
- *IP addressing*. The site assigns public IP addresses to nodes if the `visible` property is set.
- *Secure node access*. The site and guest exchange keys to enable secure, programmatic access to the leased nodes using SSL/SSH. The service manager generates a keypair and passes the public key as a configuration property. The site's *setup* handler writes the public key and a locally generated host private key onto the node image, and returns the host public key as a unit property.

The *join* and *leave* handlers execute outside of the site authority's TCB; they operate within the isolation boundaries that the authority has established for the slice and its resources. The unit properties returned for each node include the names and keys to allow the *join* handler to connect to the node to initiate *post-install* actions. In our prototype, a service manager is empowered to connect with root access and install arbitrary application software. The *join* and *leave* event handlers also interact with other application components to reconfigure the application 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 across a distributed application.

Ant has a sizable library of packaged tasks to build, configure, deploy, and launch software packages on various operating systems and Web application servers. The COD prototype includes service manager scripts to launch applications directly on leased resources, launch and dynamically resize cluster job schedulers (SGE and PBS), instantiate and/or automount NFS file volumes, and load Web applications within a virtual cluster.

3.4 Extend and Flex

There is a continuum of alternatives for adaptive resource allocation with leases. The most flexible model would permit actors to renegotiate lease contracts at any time. At the other extreme, a restrictive model might disallow any changes to a contract once it is made. Shirako leases may be extended (renewed) by mutual agreement. Peers may negotiate limited changes to the lease at renewal time, including *flexing* the number of resource units. In our prototype, changes to a renewed lease take effect only at the end of its previously agreed term.

The protocol to extend a lease involves the same pattern of exchanges as to initiate a new lease (see Figure 2). The service manager must obtain a new ticket from the broker; the ticket is marked as extending an existing ticket named by a unique ID. Renewals maintain the continuity of resource assignments when both parties agree to extend the original contract. An extend makes explicit that the next holder of a resource is the same as the current holder, bypassing the usual *teardown/setup* sequence at term boundaries. Extends also free the holder from the risk of a forced migration to a new resource assignment—assuming the renew request is honored.

With support for resource flexing, a guest can obtain these benefits even under changing demand. Without flex extends, a guest with growing resource demands is forced to instantiate a new lease for the residual demand, leading to a fragmentation of resources across a larger number of leases. Shrinking a slice would force a service manager to vacate a lease and replace it with a smaller one, interrupting continuity.

Flex extends turned out to be a significant source of complexity. For example, resource assignment on the authority must be sequenced with care to process shrinking extends first, then growing extends, then new redeems. One drawback of our current system is that a Shirako service manager has no general way to name victim units to relinquish on a shrinking extend; COD overloads configuration properties to cover this need.

Common lease core	2755
Actor state machines	1337
Cluster-on-Demand	3450
Policy modules (mappers)	1941
Calendar support for mappers	1179
Utility classes	1298

Table 2: Lines of Java code for Shirako/COD.

3.5 Lease Groups

Our initial experience with SHARP and Shirako convinced us that associating leases in *lease groups* as an important requirement. Section 3.2 outlines the related concept of *request groups*, in which a broker co-schedules grouped requests. Also, since the guest specifies properties on a per-lease basis, it is useful to obtain separate leases to allow diversity of resources and their configuration. 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. For example, consider a batch task service with a master server, worker nodes, and a file server obtained with separate leases: the file server must initialize before the master can *setup*, and the master must activate before the workers can *join* the service.

The Shirako leasing core enforces a specified configuration sequencing for lease groups on the service manager. It represents dependencies as 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 ticketed 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 plugin method before transitioning out of the blocked state; the upcall gives the plugin an opportunity to manipulate properties on the lease before it fires, or to impose more complex trigger conditions.

4 Implementation

A Shirako deployment runs as a dynamic collection of interacting peers that work together to coordinate asynchronous actions on the underlying resources. Each actor is a multithreaded server written in Java and running within a Java Virtual Machine. Actors communicate using an asynchronous peer-to-peer messaging model through a replaceable stub layer. SOAP stubs allow actors running in different JVMs to interact using Web Services protocols (Apache Axis).

Our goal was to build a common toolkit for all actors that is understandable and maintainable by one person. Table 2 shows the number of lines of Java code (semicolon lines) in the major system components of our prototype. In addition, there is a smaller body of code, definitions, and stubs to instantiate groups of Shirako actors from XML descriptors, encode and decode actor exchanges using SOAP messaging, and sign and validate SHARP-compliant exchanges. Shirako also includes a few dozen Ant scripts, averaging about 40 lines each, and other supporting scripts. These scripts configure the various resources and applications that we have experimented with, including those described in Section 5. Finally, the system includes a basic Web interface for Shirako/COD actors; it is implemented in about 2400 lines of Velocity scripting code that invokes Java methods directly.

The prototype makes use of several other open-source components. It uses Java-based tools to interact with resources when possible, in part because Java exception handling is a basis for error detection, reporting, attribution, and logging of configuration actions. Ant tasks and the Ant interpreter are written in Java, so the COD resource drivers execute configuration scripts by invoking the Ant interpreter directly within the same JVM. The event handlers often connect to nodes using key-based logins through *jsch*, a Java secure channel interface (SSH2). Actors optionally use *jldap* to interface to external LDAP repositories for recovery. COD employs several open-source components for network management based on LDAP directory servers (RFC 2307 schema standard) as described below.

4.1 Lease State Machines

The Shirako core must accommodate long-running asynchronous operations on lease objects. For example, the brokers may delay or batch requests arbitrarily, and the *setup* and *join* event handlers may take seconds, minutes, or hours to configure resources or integrate them into a guest environment. A key design choice was to structure the core as a non-blocking event-based state machine from the outset, rather than representing the state of pending operations on the stacks of threads, e.g., blocked in RPC calls. The lease state represents any pending action until a completion event triggers a state transition. Each of the three actor roles has a separate state machine.

Figure 3 illustrates typical state transitions for a resource lease through time. The state for a brokered lease spans three interacting state machines, one in each of the three principal actors involved in the lease: the service manager that requests the resources, the broker that provisions them, and the authority that owns and assigns them. Thus the complete state space for a lease is the cross-product of the state spaces for the actor state machines. The state combinations total about 360, of which about 30 are legal and reachable.

The lease state machines govern all functions of the core leasing package. 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. Actions associated with each transition may invoke a plugin, commit modified lease state and properties to an external repository, and/or generate a message to another actor. The service manager state machine is the most complex because the brokering architecture requires it to maintain ticket status and lease status independently. For example, the *ActiveTicketed* state means that the lease is active and has obtained a ticket to renew, but it has not yet redeemed the ticket to complete the lease extension. The broker and authority state machines are independent; in fact, the authority and broker interact only when resource rights are initially delegated to the broker.

The concurrency architecture promotes a clean separation of the leasing core from resource-specific code. The resource handlers—*setup/teardown*, *join/leave*, and status *probe* calls—do not hold locks on the state machines or update lease states directly. This constraint leaves them free to manage their own concurrency, e.g., by using blocking threads internally. For example, the COD node drivers start a thread to execute a designated target in an Ant script. In general, state machine threads block only when writing lease state to a repository after transitions, so servers need only a small number of threads to provide sufficient concurrency.

4.2 Time and Emulation

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

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

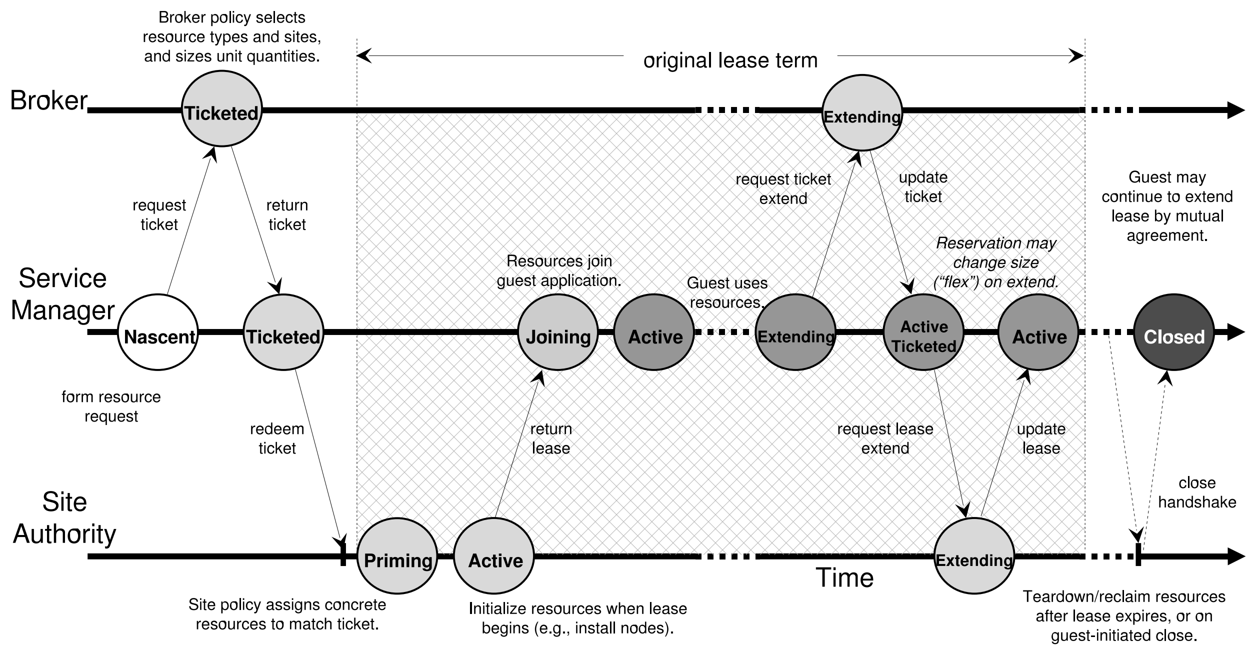

Broker policy selects resource types and sites, and sizes unit quantities.
original lease term
Broker
Ticketed
Extending
request ticket
return ticket
request ticket extend
update ticket
Guest may continue to extend lease by mutual agreement.
Reservation may change size ("flex") on extend.
Resources join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

Figure 3: Interacting lease state machines across three actors. A lease progresses through an ordered sequence of states until it is active; the rate of progress may be limited by delays imposed in the policy modules or by latencies to configure resources. Failures lead to retries or to error states reported back to the service manager. Once the lease is active, the service manager may initiate transitions through a cycle of states to extend the lease. Termination involves a handshake similar to TCP connection shutdown.

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. This feature is useful to exercise the system and control the timing and order of events. In particular, it enables emulation experiments in virtual time, as for several of the experiments in Section 5. The emulations run with null resource drivers that impose various delays but do not actually interact with external resources. All actors retain and cache lease state in memory, in part to enable lightweight emulation-mode experiments without an external repository.

4.3 Cluster Management

COD was initially designed to control physical machines with database-driven network booting (PXE/DHCP). The physical booting machinery is familiar from Emulab [28], Rocks [23], and recent commercial systems. In addition to controlling the IP address bindings assigned by PXE/DHCP, the node driver controls boot images and options by generating configuration files served via TFTP to standard bootloaders (e.g., *grub*).

A COD site authority drives cluster reconfiguration in part by writing to an external directory server. The COD schema is a superset of the RFC 2307 standard schema for a Network Information Service based on LDAP directories. Standard open-source services exist to administer networks from a LDAP repository compliant with RFC 2307. The DNS server for the site is an LDAP-enabled version of BIND9, and for physical booting we use an LDAP-enabled DHCP server from the Internet Systems Consortium (ISC). In addition, guest nodes have read access to an LDAP directory describing the containing virtual cluster. Guest nodes configured to run Linux use an LDAP-enabled version of AutoFS to mount NFS file systems, and a PAM/NSS module that retrieves user logins from LDAP.

COD should be comfortable for cluster site operators to adopt, especially if they already use RFC 2307/LDAP for administration. The directory server is authoritative: if the COD site authority fails, the disposition of the cluster is unaffected until it recovers. Operators may override the COD server with tools that access the LDAP configuration directory.

4.4 COD and Xen

In addition to the node drivers, COD includes classes to manage node sets and IP and DNS name spaces at the slice level. The authority names each instantiated node with an ID that is unique within the slice. It derives node hostnames from the ID and a specified prefix, and allocates private IP addresses as offsets in a subnet block reserved for the virtual cluster when the first node is assigned to it. Although public address space is limited, our prototype does not yet treat it as a managed resource. In our deployment the service managers run on a control subnet with routes to and from the private IP subnets.

In a further test of the Shirako architecture, we extended COD to manage virtual machines using the Xen hypervisor [2]. The extensions consist primarily of a modified node driver plugin and extensions to the authority-side mapper policy module to assign virtual machine images to physical machines. The new virtual node driver controls booting by opening a secure connection to the privileged control domain on the Xen node, and issuing commands to instantiate and control Xen virtual machines. Only a few hundred lines of code know the difference between physical and virtual machines. The combination of support for both physical and virtual machines offers useful flexibility: it is possible to assign blocks of physical machines dynamically to boot Xen, then add them to a resource pool to host new virtual machines.

COD install actions for node *setup* include some or all of the following: writing LDAP records; generating a bootloader configuration for a physical node, or instantiating a virtual machine; staging and preparing the OS image, running in the Xen control domain or on an OS-dependent trampoline such as Knoppix on the physical node; and initiating the boot. The authority writes some configuration-specific data onto the image, including the admin public keys and host private key, and an LDAP path reference for the containing virtual cluster.

5 Experimental Results

We evaluate the Shirako/COD prototype under emulation and in a real deployment. All experiments run on a testbed of IBM x335 rackmount servers, each with a single 2.8Ghz Intel Xeon processor and 1GB of memory. Some servers run Xen's virtual machine monitor version 3.0 to create virtual machines. All experiments run using Sun's Java Virtual Machine (JVM) version 1.4.2. COD uses OpenLDAP version 2.2.23-8, ISC's DHCP version 3.0.1rc11, and TFTP version 0.40-4.1 to drive network boots. Service manager, broker, and site authority Web Services use Apache Axis 1.2RC2.

Most experiments run all actors on one physical server within a single JVM. The actors interact through local proxy stubs that substitute local method calls for network communication, and copy all arguments and responses. When LDAP is used, all actors are served by a single LDAP server on the same LAN segment. Note that these choices are conservative in that the management overhead concentrates on a single server. Section 5.3 gives results using SOAP/XML messaging among the actors.

5.1 Application Performance

We first examine the latency and overhead to lease a virtual cluster for a sample guest application, the CardioWave parallel MPI heart simulator [24]. A service manager requests two leases: one for a coordinator node to launch the MPI job and another for a variable-sized block of worker nodes to run the job. It groups and sequences the lease joins as described in Section 3.5 so that all workers activate before the coordinator. The *join* handler launches CardioWave programmatically when the virtual cluster is fully active.

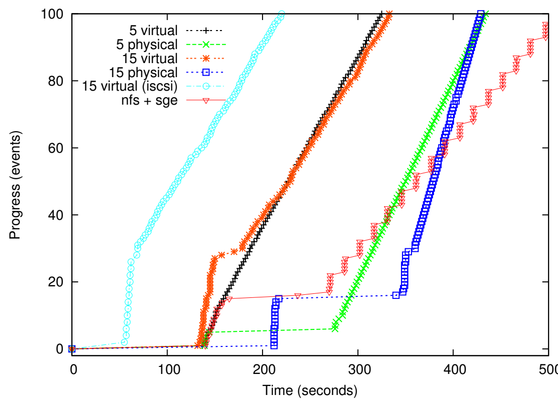


Figure 4: The progress of *setup* and *join* events and CardioWave execution on leased virtual clusters. The slope of each line gives the rate of progress. Xen clusters (left) activate faster and more reliably, but run slower than leased physical nodes (right). The step line shows an SGE batch scheduling service instantiated and subjected to a synthetic load. The fastest boot times are for VMs with flash-cloned iSCSI roots (far left).

Figure 4 charts the progress of lease activation and the CardioWave run for virtual clusters of 5 and 15 nodes, using both physical and Xen virtual machines, all with 512MB of available memory. The guest earns progress points for each completed node join and each block of completed iterations in CardioWave. Each line shows: (1) an initial flat portion as the authority prepares a file system image for each node and initiates boots; (2) a step up as nodes boot and join, (3) a second flatter portion indicating some straggling nodes, and (4) a linear segment that tracks the rate at which the application completes useful work on the virtual cluster once it is running.

The authority prepares each node image by loading a 210MB compressed image (Debian Linux 2.4.25) from a shared file server and writing the 534MB uncompressed image on a local disk partition. Some node setup delays result from contention to load the images from a shared NFS server, demonstrating the value of smarter image distribution (e.g., [15]). The left-most line in Figure 4 also shows the results of an experiment with iSCSI root drives flash-cloned by the *setup* script from a Network Appliance FAS3020 filer. Cloning iSCSI roots reduces VM configuration time to approximately 35 seconds. Network booting of physical nodes is slower than Xen and shows higher variability across servers, indicating instability in the platform, bootloader, or boot services.

Cardiowave is an I/O-intensive MPI application. It shows better scaling on physical nodes, but its perfor-

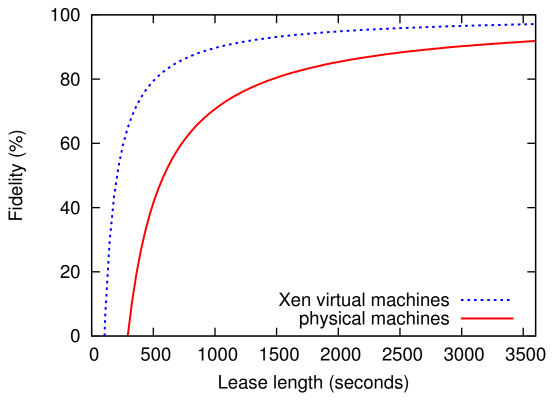


Figure 5: Fidelity is the percentage of the lease term usable by the guest application, excluding setup costs. Xen VMs are faster to *setup* than physical machines, yielding better fidelity.

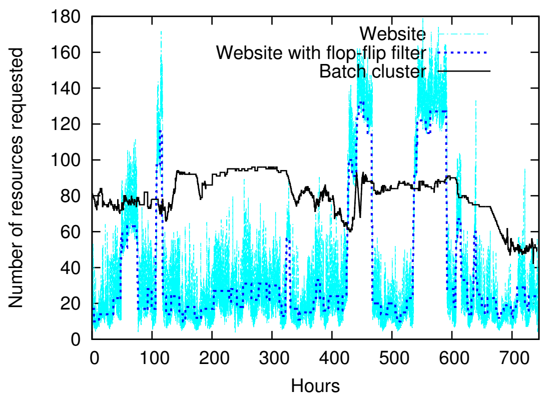


Figure 6: Scaled resource demands for one-month traces from an e-commerce website and a production batch cluster. The e-commerce load signal is smoothed with a *flop-flip* filter for stable dynamic provisioning.

mance degrades beyond ten nodes. With five nodes the Xen cluster is 14% slower than the physical cluster, and with 15 nodes it is 37% slower. For a long CardioWave run, the added Xen VM overhead outweighs the higher setup cost to lease physical nodes.

A more typical usage of COD in this setting would be to instantiate batch task services on virtual compute clusters [7], and let them schedule Cardiowave and other jobs without rebooting the nodes. Figure 4 includes a line showing the time to instantiate a leased virtual cluster comprising five Xen nodes and an NFS file server, launch a standard Sun GridEngine (SGE) job scheduling service on it, and subject it to a synthetic task load. This example uses lease groups to sequence configuration as described in Section 3.5. The service manager also stages a small data set (about 200 MB) to the NFS server, increasing the activation time. The steps in the line correspond to simultaneous completion of synthetic tasks on the workers.

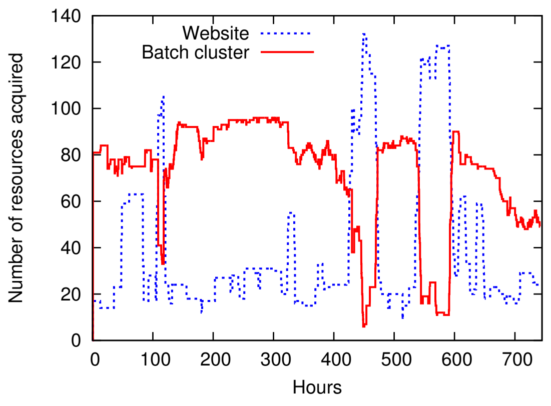
Figure 5 uses the *setup/join/leave/teardown* costs from the previous experiment to estimate their effect on the system's *fidelity* to its lease contracts. Fidelity is the percentage of the lease term that the guest application is able to use its resources. Amortizing these costs over longer lease terms improves fidelity. Since physical machines take longer to *setup* than Xen virtual machines, they have a lower fidelity and require longer leases to amortize their costs.

5.2 Adaptivity to Changing Load

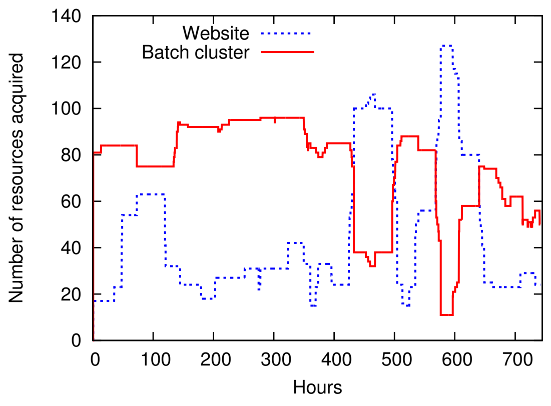
This section demonstrates the role of brokers to arbitrate resources under changing workload, and coordinate resource allocation from multiple sites. This experiment runs under emulation (as described in Section 4.2) with null resource drivers, virtual time, and lease state stored only in memory (no LDAP). In all other respects the emulations are identical to a real deployment. We use two emulated 70-node cluster sites with a shared broker. The broker implements a simple policy that balances the load evenly among the sites.

We implemented an adaptive service manager that requests resource leases at five-minute intervals to match a changing load signal. We derived sample input loads from traces of two production systems: a job trace from a production compute cluster at Duke, and a trace of CPU load from a major e-commerce website. We scaled the load signals to a common basis. Figure 6 shows scaled cluster resource demand—interpreted as the number of nodes to request—over a one-month segment for both traces (five-minute intervals). We smoothed the e-commerce demand curve with a "flop-flip" filter from [6]. This filter holds a stable estimate of demand E_t=E_{t-1} until that estimate falls outside some tolerance of a moving average ($E_t = \beta E_{t-1} + (1-\beta)O_t$) of recent observations, then it switches the estimate to the current value of the moving average. The smoothed demand curve shown in Figure 6 uses a 150-minute sliding window moving average, a step threshold of one standard deviation, and a heavily damped average β=7/8.

Figure 7 demonstrates the effect of varying lease terms on the broker's ability to match the e-commerce load curve. For a lease term of one day, the leased resources closely match the load; however, longer terms diminish the broker's ability to match demand. To quantify the effectiveness and efficiency of allocation over the one-month period, we compute the *root mean squared error* (RMSE) between the load signal and the requested resources. Numbers closer to zero are better: an RMSE of zero indicates that allocation exactly matches demand. For a lease term of 1 day, the RMSE is 22.17 and for a lease term of 7 days, the RMSE is 50.85. Figure 7 reflects a limitation of the pure brokered leasing model as prototyped: a lease holder can return unused resources to the authority, but it cannot return the ticket to the broker to allocate for other purposes.



(a) Lease term of 12 emulated hours.



(b) Lease term of 3 emulated days.

Figure 8: Brokering of 140 machines from two sites between a low-priority computational batch cluster and a high-priority e-commerce website that are competing for machines. Where there is contention for machines, the high priority website receives its demand causing the batch cluster to receive less. Short lease terms (a) are able to closely track resource demands, while long lease terms (b) are unable to match short spikes in demand.

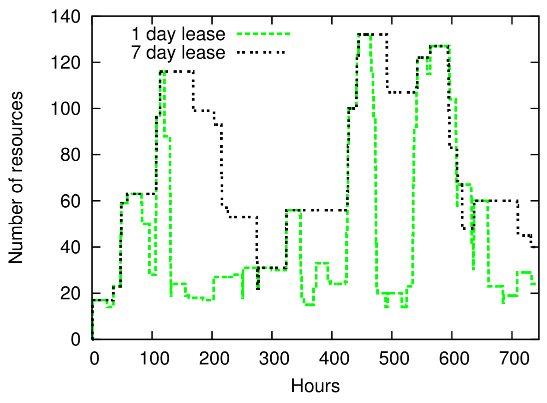


Figure 7: The effect of longer lease terms on a broker's ability to match guest application resource demands. The website's service manager issues requests for machines, but as the lease term increases, the broker is less effective at matching the demand.

To illustrate adaptive provisioning between competing workloads, we introduce a second service manager competing for resources according to the batch load signal. The broker uses FCFS priority scheduling to arbitrate resource requests; the interactive e-commerce service receives a higher priority. Figure 8 shows the assigned slice sizes for lease terms of (a) 12 emulated hours and (b) 3 emulated days. As expected, the batch cluster receives fewer nodes during load surges in the e-commerce service. However, with longer lease terms, load matching becomes less accurate, and some short demand spikes are not served. In some instances, resources assigned to one guest are idle while the other guest saturates but cannot obtain more. This is seen in the RMSE calculated from Figure 8: the website has a RMSE of (a) 12.57 and (b) 30.70 and the batch cluster has a RMSE of (a) 23.20 and (b) 22.17. There is a trade-off in choosing the length of lease terms: longer terms are more stable and better able to amortize resource setup/teardown costs improving fidelity (from Section 5.1), but are not as agile to changing demand as shorter leases.

N	cluster size
l	number of active leases
n	number of machines per lease
t	term of a lease in virtual clock ticks
α	overhead factor (ms per virtual clock ticks)
t'	term of a lease (ms)
r'	average number of machine reallocations per ms

Table 3: Parameter definitions for Section 5.3

5.3 Scaling of Infrastructure Services

These emulation experiments demonstrate how the lease management and configuration services scale at saturation. Table 3 lists the parameters used in our experiment: for a given cluster size N at a single site, one service manager injects lease requests to a broker for N nodes (without lease extensions) evenly split across l leases (for $N/l = n$ nodes per lease) every lease term t (giving a request injection rate of l/T). Every lease term t the site must reallocate or "flip" all N nodes. We measure the total overhead including lease state maintenance, network communication costs, actor database operations, and event polling costs. Given parameter values we can derive the worst-case minimum lease term, in real time, that the system can support at saturation.

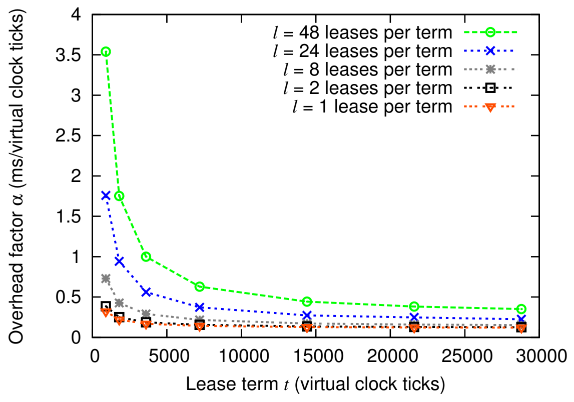


Figure 9: The implementation overhead for an example Shirako scenario for a single emulated cluster of 240 machines. As lease term increases, the overhead factor α decreases as the actors spend more of their time polling lease status rather than more expensive setup/teardown operations. Overhead increases with the number of leases (l) requested per term.

As explained in Section 4.2, each actor's operations are driven by a virtual clock at an arbitrary rate. The prototype polls the status of pending lease operations (i.e., completion of *join/leave* and *setup/teardown* events) on each tick. Thus, the rate at which we advance the virtual clock has a direct impact on performance: a high *tick rate* improves responsiveness to events such as failures and completion of configuration actions, but generates higher overhead due to increased polling of lease and resource status. In this experiment we advance the virtual clock of each actor as fast as the server can process the clock ticks, and determine the amount of real time it takes to complete a pre-defined number of ticks. We measure an *overhead factor* α: the average lease management overhead in milliseconds per clock tick. Lower numbers are better.

Local communication. In this experiment, all actors run on a single x335 server and communicate with local method calls and an in-memory database (no LDAP). Figure 9 graphs α as a function of lease term t in virtual clock ticks; each line presents a different value of l keeping N constant at 240. The graph shows that as t increases, the average overhead per virtual clock tick decreases; this occurs because actors perform the most expensive operation, the reassignment of N nodes, only once per lease term leaving less expensive polling operations for the remainder of the term. Thus, as the number of polling operations increases, they begin to dominate α. Figure 9 also shows that as we increase the number of leases injected per term, α also increases. This demonstrates the increased overhead to manage the leases.

At a clock rate of one tick per second, the overhead represents less than 1% of the latency to prime a node (i.e., to write a new OS image on local disk and boot it). As an example from Figure 9, given this tick rate, for a lease term of 1 hour (3,600 virtual clock ticks), the total overhead of our implementation is t'=$t\alpha$=2.016 seconds with l=24 leases per term. The lease term t' represents the minimum term we can support considering only implementation overhead. For COD, these overheads are at least an order of magnitude less than the *setup/teardown* cost of nodes with local storage. From this we conclude that the *setup/teardown* cost, not overhead, is the limiting factor for determining the minimum lease term. However, overhead may have an effect on more fine-grained resource allocation, such as CPU scheduling, where reassignments occur at millisecond time scales.

N (cluster size)	α	stdev α	t'
120	0.1183	0.001611	425.89
240	0.1743	0.000954	627.58
360	0.2285	0.001639	822.78
480	0.2905	0.001258	1,045.1

Table 4: The effect of increasing the cluster size on α as the number of active leases is held constant at one lease for all N nodes in the cluster. As cluster size increases, the per-tick overhead α increases, driving up the minimal lease term t'.

RPC Type	Database	α	stdev α	t'	r'
Local	Memory	.1743	.0001	627	.3824
Local	LDAP	5.556	.1302	20,003	.0120
SOAP	Memory	27.902	1.008	100,446	.0024
SOAP	LDAP	34.041	.2568	122,547	.0019

Table 5: Impact of overhead from SOAP messaging and LDAP access. SOAP and LDAP costs increase overhead α (*ms*/virtual clock tick), driving down the maximum node flips per millisecond r' and driving up the minimum practical lease term t'.

Table 4 shows the effect of varying the cluster size N on the overhead factor α. For each row of the table, the service manager requests one lease (l=1) for N nodes (N=n) with a lease term of 3,600 virtual clock ticks (corresponding to a 1 hour lease with a tick rate of 1 second). We report the average and one standard deviation of α across ten runs. As expected, α and t' increase with cluster size, but as before, t' remains an order of magnitude less than the setup/teardown costs of a node.

SOAP and LDAP. We repeat the same experiment with the service manager running on a separate x335 server, communicating with the broker and authority using SOAP/XML. The authority and broker write their state to a shared LDAP directory server. Table 5 shows the impact of the higher overhead on t' and r', for N=240. Using α, we calculate the maximum number of node flips per millisecond r'=$N/(T\alpha)$ at saturation. The SOAP and LDAP overheads dominate all other lease management costs: with $N = 240$ nodes, an x335 can process 380 node flips per second, but SOAP and LDAP communication overheads reduce peak flip throughput to 1.9 nodes per second. Even so, neither value presents a limiting factor for today's cluster sizes (thousands of nodes). Using SOAP and LDAP at saturation requires a minimum lease term t' of 122 seconds, which approaches the setup/teardown latencies (Section 5.1).

From these scaling experiments, we conclude that lease overhead is quite modest, and that costs are dominated by per-tick resource polling, node reassignment, and network communication. In this case, the dominant costs are LDAP access and SOAP operations and the cost for Ant to parse the XML configuration actions and log them.

6 Related Work

Variants of leases are widely used when a client holds a resource on a server. The common purpose of a lease abstraction is to specify a mutually agreed time at which the client's right to hold the resource expires. If the client fails or disconnects, the server can reclaim the resource when the lease expires. The client renews the lease periodically to retain its hold on the resource.

Lifetime management. Leases are useful for distributed garbage collection. The technique of robust distributed reference counting with expiration times appeared in Network Objects [5], and subsequent systems—including Java RMI [29], Jini [27], and Microsoft .NET—have adopted it with the "lease" vocabulary. Most recently, Web Services WSRF [10] has defined a lease protocol as a basis for lifetime management of hosted services.

Mutual exclusion. Leases are also useful as a basis for distributed mutual exclusion, most notably in cache consistency protocols [14, 21]. To modify a block or file, a client first obtains a lease for it in an exclusive mode. The lease confers the right to access the data without risk of a conflict with another client as long as the lease is valid. The key benefit of the lease mechanism itself is availability: the server can reclaim the resource from a failed or disconnected client after the lease expires. If the server fails, it can avoid issuing conflicting leases by waiting for one lease interval before granting new leases after recovery.

Resource management. As in SHARP [13], the use of leases in Shirako combines elements of both lifetime management and mutual exclusion. While providers may choose to overbook their physical resources locally, each offered logical resource unit is held by at most one lease at any given time. If the lease holder fails or disconnects, the resource can be allocated to another guest. This use of leases has three distinguishing characteristics:.

- Shirako leases apply to the resources that host the guest, and not to the guest itself; the resource provider does not concern itself with lifetime management of guest services or objects.
- The lease quantifies the resources allocated to the guest; thus leases are a mechanism for service quality assurance and adaptation.
- Each lease represents an explicit promise to the lease holder for the duration of the lease. The notion of a lease as an enforceable contract is important in systems where the interests of the participants may diverge, as in peer-to-peer systems and economies.

Leases in Shirako are also similar to soft-state advance reservations [8, 30], which have long been a topic of study for real-time network applications. A similar model is proposed for distributed storage in L-bone [3]. Several works have proposed resource reservations with bounded duration for the purpose of controlling service quality in a grid. GARA includes support for advance reservations, brokered co-reservations, and adaptation [11, 12].

Virtual execution environments. New virtual machine technology expands the opportunities for resource sharing that is flexible, reliable, and secure. Several projects have explored how to link virtual machines in virtual networks [9] and/or use networked virtual machines to host network applications, including SoftUDC [18], In Vigo [20], Collective [25], SODA [17], and Virtual Playgrounds [19]. Shared network testbeds (e.g., Emulab/Netbed [28] and PlanetLab [4]) are another use for dynamic sharing of networked resources. Many of these systems can benefit from foundation services for distributed lease management.

PlanetLab was the first system to demonstrate dynamic instantiation of virtual machines in a wide-area testbed deployment with a sizable user base. PlanetLab's current implementation and Shirako differ in their architectural choices. PlanetLab consolidates control in one central authority (PlanetLab Central or PLC), which is trusted by all sites. Contributing sites are expected to relinquish permanent control over their resources to the PLC. PlanetLab emphasizes best-effort open access over admission control; there is no basis to negotiate resources for predictable service quality or isolation. PlanetLab uses leases to manage the lifetime of its guests, rather than for resource control or adaptation.

The PlanetLab architecture permits third-party brokerage services with the endorsement of PLC. PlanetLab brokers manage resources at the granularity of individual nodes; currently, the PlanetLab Node Manager cannot control resources across a site or cluster. PLC may delegate control over a limited share of each node's resources to a local broker server running on the node. PLC controls the instantiation of guest virtual machines, but each local broker is empowered to invoke the local Node Manager interface to bind its resources to guests instantiated on its node. In principle, PLC could delegate sufficient resources to brokers to permit them to support resource control and dynamic adaptation coordinated by a central broker server, as described in this paper.

One goal of our work is to advance the foundations for networked resource sharing systems that can grow and evolve to support a range of resources, management policies, service models, and relationships among resource providers and consumers. Shirako defines one model for how the PlanetLab experience can extend to a wider range of resource types, federated resource providers, clusters, and more powerful approaches to resource virtualization and isolation.

7 Conclusion

This paper focuses on the design and implementation of general, extensible abstractions for brokered leasing as a basis for a federated, networked utility. The combination of Shirako leasing services and the Cluster-on-Demand cluster manager enables dynamic, programmatic, reconfigurable leasing of cluster resources for distributed applications and services. Shirako decouples dependencies on resources, applications, and resource management policies from the leasing core to accommodate diversity of resource types and resource allocation policies. While a variety of resources and lease contracts are possible, resource managers with performance isolation enable guest applications to obtain predictable performance and to adapt their resource holdings to changing conditions.

References

[1] Ant, September 2005. `http://ant.apache.org/`.

[2] P. Barham, B. Dragovic, K. Faser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[3] A. Bassi, M. Beck, T. Moore, and J. S. Plank. The logistical backbone: Scalable infrastructure for global data grids. In *Proceedings of the 7th Asian Computing Science Conference on Advances in Computing Science*, December 2002.

[4] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *First Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, December 1993.

[6] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.

[7] 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-12)*, June 2003.

[8] M. Degermark, T. Kohler, S. Pink, and O. Schelen. Advance reservations for predictive service in the Internet. *Multimedia Systems*, 5(3):177–186, 1997.

[9] R. J. Figueiredo, P. A. Dinda, and F. Fortes. A case for grid computing on virtual machines. In *International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[10] I. Foster, K. Czajkowski, D. F. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and managing state in distributed systems: The role of OGSI and WSRF. *Proceedings of the IEEE*, 93(3):604–612, March 2005.

[11] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, June 1999.

[12] I. Foster and A. Roy. A quality of service architecture that combines resource reservation and application adaptation. In *Proceedings of the International Workshop on Quality of Service*, June 2000.

[13] 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.

[14] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[15] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

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

[17] X. Jiang and D. Xu. Soda: A service-on-demand architecture for application service hosting utility platforms. In *12th IEEE International Symposium on High Performance Distributed Computing*, June 2003.

[18] M. Kallahalla, M. Uysal, R. Swaminathan, D. Lowell, M. Wray, T. Christian, N. Edwards, C. Dalton, and F. Gittler. SoftUDC: A software-based data center for utility computing. In *Computer*, volume 37, pages 38–46. IEEE, November 2004.

[19] K. Keahey, K. Doering, and I. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *5th International Workshop in Grid Computing*, November 2004.

[20] I. Krsul, A. Ganguly, J. Zhang, J. Fortes, and R. Figueiredo. VMPlants: Providing and managing virtual machine execution environments for grid computing. In *Supercomputing*, October 2004.

[21] R. Macklem. Not quite NFS, soft cache consistency for NFS. In *USENIX Association Conference Proceedings*, pages 261–278, January 1994.

[22] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs in Wide-Area Resource Discovery. In *Proceedings of Fourteenth Annual Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[23] P. M. Papadopoulous, M. J. Katz, and G. Bruno. NPACI Rocks: Tools and techniques for easily deploying manageable Linux clusters. In *IEEE Cluster 2001*, October 2001.

[24] J. Pormann, J. Board, D. Rose, and C. Henriquez. Large-scale modeling of cardiac electrophysiology. In *Proceedings of Computers in Cardiology*, September 2002.

[25] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[26] N. Taesombut and A. Chien. Distributed Virtual Computers (DVC): Simplifying the development of high performance grid applications. In *Workshop on Grids and Advanced Networks*, April 2004.

[27] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[28] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[29] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies (COOTS)*, June 1997.

[30] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, September 1993.