AN OPERATING SYSTEM ARCHITECTURE FOR NETWORKED SERVER INFRASTRUCTURE

by

David E. Irwin

Department of Computer Science
Duke University

Date: ______________________________

Approved:

____
Dr. Jeffrey S. Chase, Supervisor

____
Dr. Landon P. Cox

____
Dr. Carla S. Ellis

____
Dr. Parthasarathy Ranganathan

____

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2008

ABSTRACT

AN OPERATING SYSTEM ARCHITECTURE FOR NETWORKED SERVER INFRASTRUCTURE

by

David E. Irwin

Department of Computer Science
Duke University

Date: ______________________

Approved:

____
Dr. Jeffrey S. Chase, Supervisor

____
Dr. Landon P. Cox

____
Dr. Carla S. Ellis

____
Dr. Parthasarathy Ranganathan

____

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2008

Copyright © 2008 by David E. Irwin
All rights reserved

Abstract

Collections of hardware components are the foundation of computation and consist of interconnections of different types of the same core elements: processors, disks, memory cards, I/O devices, and network links. Designing a system for managing collections of hardware is challenging because modern infrastructures (i) distribute resource control across multiple autonomous sites, (ii) operate diverse sets of hardware, and (iii) support a variety of programming models for developing and executing software services.

An operating system is a software layer that manages hardware by coordinating its interaction with software. This thesis defines and evaluates an architecture for a networked operating system that manages collections of hardware in infrastructures spread across networks, such as the Internet. The foundation of a networked operating system determines how software services share a common hardware platform. A fundamental property common to all forms of resource sharing is that software services, by definition, share hardware components and do not use them forever. A lease is a natural construct for restricting the use of a shared resource to a well-defined length of time.

Our architecture employs a general neutrality principle, which states that a networked operating system should be policy-neutral, since only users and site administrators, and not operating system developers, know how to manage their software and hardware. Experience building, deploying, and using a prototype has led us to view neutrality as a guiding design principle. Our hypothesis is that an operating system architecture for infrastructure resource management that focuses narrowly on leasing control of hardware provides a foundation for multi-lateral resource negotiation, arbitration, and fault tolerance. In evaluating our hypothesis we make the following contributions:

- Introduce a set of design principles for networked operating systems. The principles adapt and extend principles from node operating system design to a networked environment. We evaluate existing systems with respect to these principles, describe how they deviate from them, and explore how these deviations limit the capabilities of higher level software.

- Combine the idea of a reconfigurable data center with the SHARP framework for secure resource peering to demonstrate a prototype networked operating system capable of sharing aggregations of resources in infrastructures.

- Design, implement, and deploy the architecture using a single programming abstraction—the lease—and show how the lease abstraction embodies the design principles of a networked operating system.
- Show that leases are a foundational primitive for addressing arbitration in a networked operating system. Leasing currency defines a configurable tradeoff between proportional-share scheduling and a market economy, and also serves as a basis for implementing other forms of arbitration.
- Show how combining the use of leases for long-term resource management with state recovery mechanisms provides robustness to transient faults and failures in a loosely coupled distributed system that coordinates resource allocation.
- Evaluate the flexibility and performance of a prototype by managing aggregations of physical and virtual hardware present in modern data centers, and showing that the architecture could scale to manage thousands of machines.
- Present case studies of integrating multiple software services including the PlanetLab network testbed, the Plush distributed application manager, and the GridEngine batch scheduler, and leverage the architecture to prototype and evaluate Jaws, a new light-weight batch scheduler that instantiates one or more virtual machines per task.

Acknowledgements

First, I would like to thank my adviser Jeff Chase for his guidance throughout my graduate career. It is impossible to accurately convey how much I have learned from Jeff over the past six years. I would also like to thank my committee members Landon Cox, Carla Ellis, and Partha Ranganathan. I have enjoyed hearing Landon's thoughts, criticisms, and ideas at the various seminars and colloquia at Duke over the past two years. Carla has always been supportive and encouraging of my research efforts since the beginning of my graduate career. I interned for Partha at HP Labs during my third year of graduate school, where his enthusiasm and drive in developing and implementing new ideas serves as a model example for any graduate student.

I would like to thank all of my collaborators over the past six years including David Becker, Laura Grit, Anda Iamnitchi, Varun Marupadi, Lavanya Ramakrishnan, Matthew Sayler, Piyush Shivam, Sara Sprenkle, Ken Yocum, and Aydan Yumerefendi. I would especially like to thank Aydan and Laura for their collective contributions to Shirako, and David Becker for his help in answering my numerous questions. I would also like to acknowledge my officemates and/or roommates during my graduate school tenure, including Allister Bernard, Andrew Danner, and Jaidev Patwardhan, and my friends in San Diego, especially Chip Killian, Dejan Kostíc, Priya Mahadevan, Barath Raghavan, Patrick Reynolds, and Kashi Vishwanath. I would also like to acknowledge and thank Diane Riggs for coordinating the graduate school bureaucracy on my behalf during my many absences from Duke.

Finally, I would like to thank my family, especially my parents, John and Sue, and my wife, Jeannie. My parents helped me in enumerable ways throughout my graduate career. Without Jeannie's support throughout the majority of my graduate career, I am certain I would neither have started nor finished this dissertation—now that I am done we can look forward to starting a "normal" life that does not require apartments in both San Diego and Durham and bi-monthly cross-country flights.

Contents

Abstract . . . . **iv**

Acknowledgements . . . . **vi**

List of Tables . . . . **xii**

List of Figures . . . . **xiv**

1 Introduction **1**

1.1 Problem 2

1.1.1 Infrastructure Complexity 3

1.1.2 Impact of Virtualization 4

1.2 Solution 5

1.2.1 Focusing Questions 5

1.2.2 The Neutrality Principle 7

1.3 Hypothesis and Contributions 9

1.4 Thesis Outline 10

1.5 Acknowledgements 11

2 The Extensible Operating System Approach **14**

2.1 Relationship to Node Operating Systems 14

2.1.1 Lessons from Microkernels 16

2.1.2 Exokernel 17

2.2 An Extensible Networked Operating System 20

2.2.1 Multiplex Collections of Networked Resources 21

2.2.2 Allocate Resources across Multiple Sites 22

2.2.3 Coarse-grained Resource Allocation 24

2.3 Summary 26

3 Classical Resource Management . . . **27**
3.1 Hardware Virtualization . . . 29
3.1.1 Machine Virtualization . . . 29
3.1.2 Storage Virtualization . . . 30
3.1.3 Network Virtualization . . . 31
3.2 Middleware . . . 31
3.2.1 Batch Job Scheduling . . . 32
3.2.2 Grid Computing . . . 33
3.3 Data Center Operating Systems . . . 34
3.3.1 PlanetLab . . . 37
3.3.2 Utility Computing . . . 37
3.4 Elements of Distributed Systems . . . 38
3.4.1 Resource Scheduling and Arbitration . . . 38
3.4.2 Leases and Fault Tolerance . . . 40
3.5 Summary . . . 41
4 A Networked Operating System Architecture . . . **42**
4.1 Background . . . 42
4.1.1 Resource Leases . . . 43
4.1.2 Sharp Brokers . . . 44
4.2 Shirako and Cluster-on-Demand . . . 45
4.3 Design Principles . . . 47
4.3.1 Guest/Resource Neutrality . . . 48
4.3.2 Visible Allocation, Modification, and Revocation . . . 51
4.3.3 An Abort Protocol . . . 55
4.4 Example: Allocating Virtual Clusters to Guests . . . 55
4.4.1 Integrating Machine Resources . . . 56

4.4.2 Support for Virtual Machines . . . 57
4.4.3 Defining Machine Properties . . . 57
4.5 Summary . . . 59
5 Exposing Names and Secure Bindings . . . 60
5.1 Example: Virtual Machines . . . 60
5.2 Overview . . . 61
5.3 Exposing Names . . . 63
5.3.1 Site-assigned Computons . . . 65
5.3.2 Broker-managed Hosts . . . 67
5.3.3 Broker-guided Colocation . . . 71
5.3.4 Broker-guided Colocation and Sliver Naming . . . 73
5.4 Secure Bindings . . . 80
5.4.1 Logical Unit Naming . . . 80
5.4.2 Delegating Hardware Management . . . 82
5.4.3 Virtual Machine Examples . . . 84
5.4.4 Delegation using Leases . . . 88
5.4.5 Guest Failure Handling . . . 89
5.5 Exposing Information . . . 89
5.6 Summary . . . 90
6 The Lease as an Arbitration Mechanism . . . 91
6.1 Overview . . . 92
6.2 Self-Recharging Virtual Currency . . . 94
6.2.1 Rationale . . . 95
6.2.2 The PDP-1 Market . . . 97
6.2.3 Generalizing the PDP-1 Market . . . 98
6.3 Using Credits in a Simple Market . . . 100

6.3.1 A Market-based Task Service . . . 101
6.3.2 Effect on Global Utility . . . 105
6.4 Summary . . . 110
7 Leases and Fault Tolerance . . . 112
7.1 Approach . . . 112
7.1.1 Overview . . . 113
7.1.2 Interacting State Machines . . . 114
7.2 Actor Disconnections . . . 116
7.2.1 Actor Stoppage and Network Partitions . . . 116
7.2.2 Local State Recovery . . . 117
7.3 Lease Synchronization . . . 118
7.3.1 Actor Recovery . . . 120
7.3.2 Guest/Resource Recovery . . . 120
7.4 Implementation . . . 125
7.4.1 Lease State Machine . . . 125
7.4.2 Persistence and Recovery . . . 126
7.4.3 Idempotent Handlers . . . 127
7.5 Summary . . . 129
8 Flexibility and Performance . . . 130
8.1 Goals . . . 130
8.2 Implementation . . . 131
8.3 Flexibility . . . 133
8.3.1 Physical Machines, Virtual Machines, and Slivers . . . 135
8.3.2 Amazon's Elastic Compute Cloud . . . 139
8.4 Performance . . . 141
8.4.1 Illustration . . . 141

8.4.2 Scalability . . . 144
8.5 Summary . . . 147
9 Guest Case Studies . . . 148
9.1 GridEngine . . . 148
9.1.1 Integration . . . 150
9.1.2 Demonstration . . . 153
9.1.3 Lessons . . . 156
9.2 Plush . . . 158
9.2.1 Integration . . . 159
9.2.2 Lessons . . . 161
9.3 Planetlab . . . 161
9.3.1 Integration . . . 162
9.3.2 Discussion . . . 163
9.3.3 Lessons . . . 166
9.4 Jaws . . . 166
9.4.1 Design and Prototype . . . 167
9.4.2 Lessons . . . 169
9.5 Summary . . . 169
10 Final Thoughts . . . 172
10.1 Contributions . . . 172
10.2 Future Directions . . . 175
Bibliography . . . 178
Biography . . . 192

List of Tables

2.1 The principles underlying Exokernel and Shirako are similar. Both systems focus on hardware multiplexing, visible allocation/revocation, secure bindings, an abort protocol, and exposing names. 20

4.1 The lease abstraction for the service manager implements the guest handler interface. Service manager developers may add or modify guest handlers to support new types of guests. 49
4.2 The lease abstraction for the site authority implements the resource handler interface. Site administrators add new resource handlers or update existing resource handlers to lease different types of resources. 49
4.3 The lease abstraction for the service manager implements the guest lease handler interface to give service managers an opportunity to modify leases when their state changes. 53
4.4 Selected properties used by Cluster-on-Demand, and sample values. 58

6.1 The bidding window is the number of auctions *Burst* bids when it has queued tasks. The bidding window from Figure 6.4 dictates the amount of currency bid for each auction. The table above shows the amount of currency *Burst* bids for each bidding window. The large the window the lower the amount of currency bid, since its total currency holdings are split across more bids. 108

7.1 A table outlining the the effect of a single actor failure on a guest in a small network of actors that includes a single service manager, broker, and site authority. 124

8.1 Lines of Java code for Shirako/COD. 132
8.2 A table outlining the different resource drivers currently supported by Shirako. Aspects of the specific technology are hidden from the lease abstraction beneath the resource handler interface. 135
8.3 A table outlining the resource driver interface for a virtual machine. The authority controls the binding of slivers to virtual machines, which are in italics. An authority may export control of the hardware management services in bold since their execution does not concern resource multiplexing or interfere with competing guests. 136
8.4 Parameter definitions for Section 8.4 144
8.5 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'. 146
8.6 Impact of overhead from LDAP access. 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'. 146

9.1 A description of functions exposed by the Plush master via XML-RPC. External entities, such as a service manager, call `Load`, `CreateResource`, `Run`, `AddResource`, and `RemoveResource` to drive application execution. The Plush application description registers two callbacks with the Plush service manager, `NotifyExit` and `NotifyFailure`, which the Plush master invokes when an applications completes or fails, respectively. 159

9.2 A summary of the different guests that we have integrated into Shirako as service managers. Integration enables a level of adaptation and resource control not available to each platform, itself. In addition to these guests, others have used the architecture to integrate the Rubis multi-tier web application [185] and Globus grids [138]. 169

List of Figures

2.1 Distributed computing environments, such as PlanetLab and Globus Grids, are built above hardware-level resource abstractions exported by resource owners. 15

2.2 An overview of Exokernel's structure from [61]. The kernel acts as a secure multiplexer of hardware, in the form of CPU quanta, disk blocks, and memory pages, for multiple guests. The kernel does not define programming abstractions such as file systems, processes, threads, or virtual memory that restrict guests' use of the hardware. Guests are free to use the hardware directly to define their own programming abstractions, or link against library operating systems that provide a set of common programming abstractions. 18

2.3 The figure compares the structure of Exokernel with a networked operating system. The primary design differences stem from a focus on multiplexing collections of resources from multiple sites at the coarser granularity of physical machines, virtual machines, and slivers. The focus motivates both a physical separation between the service manager, the authority, and the guests and resources they control, and a logical separation of policy between multiple authorities and brokers. 22

3.1 Physical and virtual hardware, node operating systems, and middleware are building blocks for a new underware layer that programmatically maps software onto hardware. Authorities control the binding of the resources they own to the service managers that request them. Service managers use network-accessible APIs exported by each authority to control the mapping of software (node operating systems, middleware, etc.) for their guests onto each site's hardware. Brokers represent the cohesive entity that coalesces resources from multiple sites and offers them to service managers. . . 28

4.1 Summary of protocol interactions and extension points for the leasing system. A guest-specific service manager uses the lease API to request resources from a Sharp 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 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 guest. Plug-in modules for *setup* and *join* event handlers are applicable to different resource types and guests, respectively. 45

4.2 An example scenario that illustrates Sharp and COD using a lease abstraction. The scenario depicts a guest acquiring machines from two sites through a broker. Each site maintains an authority server that controls its physical machines, and registers inventories of offered machines with the broker. A service manager interacts with a broker to lease machines from two different authorities on behalf of its guest. The lease abstraction is resource-independent, guest-independent, and policy-neutral, and masks all protocol interactions and lease state maintenance from the service manager controlling its guest, the authority controlling its resources, and the broker mediating between the two. 47

5.1 Sharp tickets specify a resource type and resource unit count. COD associates each resource unit with a single host. 64

5.2 Each color represents a different lease. A computon is a predefined number of grains that comprise a sliver. To support logical unit colocation using computons, an authority associates a resource type with a computon and a resource unit count with a number of logical units. An authority assignment policy colocates multiple computons and logical units on each host. 66

5.3 Authorities may permit brokers to determine the quantity of each grain that forms a sliver as long as each ticket includes information specifying the host. Sharp accommodates host specification if brokers specify a host by associating it with a resource type. 67

5.4 Brokers use vector addition to represent the two-dimensional packing of slivers onto hosts. The figure depicts a host with slivers that comprise 16 grains in two different dimensions. A broker carves tickets a, b, and c for slivers from the host. 68

5.5 Augmenting each ticket with a list of host names permits brokers to allocate multiple slivers within each ticket. Each color represents a different lease. 72

5.6 Naming individual slivers permits brokers to separate the different grains of a sliver into different leases, enabling a range of mechanisms that require sliver identification, including generalized victim selection, lease forking and merging, and host selection. 75

5.7 Naming logical units separately from slivers allows authorities to bind/unbind multiple slivers to a logical unit. These slivers may include remote slivers bound to different hosts, such as a remote storage server exporting a virtual machine root disk image. The scheme is general enough to also apply to logical resources, such as scarce public IP address space. 81

5.8 The figure depicts an authority that reserves control of the hardware management functions of each virtual machine. The service manager selects a root image from a template of options and the authority creates the virtual machine using the *setup* handler. The authority transfers control of the virtual machine to a service manager by configuring `sshd`, or an equivalent secure login server, with a public key, which the service manager transfers using a configuration property. 83

5.9 The figure depicts four service managers executing example hardware management functions for their leased virtual machines. Secure bindings permit service managers or authorities to access functions that control the hardware management services of a logical unit, such as a virtual machine. Delegating control of hardware management functions to service managers alleviates authorities from defining multiple software configurations and allows authorities to focus solely on hardware multiplexing and configuring access control for secure bindings. 84

6.1 Flow of credits and resources in a networked infrastructure. Leasing rights for resources flow from sites down through a broker network to consuming service managers. Brokers allocate resource rights using any policy or auction protocol: payments in credits flow up through the broker network. 95

6.2 The *credit recharge rule* returns credits to the buyers after a configurable recharge time r from when the currency is committed to a bid in an auction, or if a bidder loses an auction. 97

6.3 An example value function. The task earns a maximum value if it executes immediately and completes within its minimum run time. The value decays linearly with queuing delay. The value may decay to a negative number, indicating a penalty. The penalty may or may not be bounded. 103

6.4 Two market-based task services, called *Burst* and *Steady*, bid for resources to satisfy their load. As the recharge time increases, the global utility increases until the point at which the task services' static bidding strategies are too aggressive: they run out of currency before they finish executing their task load. The result is the global utility falls below that of a static partitioning of resources. The bidding window for *Burst* (each line) is the expected length of time to complete a *Burst* of tasks, according to its static bidding strategy. As the bidding strategy becomes more conservative (*i.e.*, lower numbers) relative to the recharge time the allocation achieves a higher global utility despite longer recharge times. 107

7.1 Interacting lease state machines across three actors. A lease progresses through an ordered sequence of states until it is active; delays imposed by policy modules or the latencies to configure resources may limit the state machine's rate of progress. 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. 115

8.1 Handler for configuring a virtual machine. Note that the handler is a simplification that does not show the details of each Ant target. 137

8.2 Handler for configuring a physical machine. Note that the handler is a simplification that does not show the details of each Ant target. 138

8.3 The progress of *setup* and *join* events and CardioWave execution on leased machines. 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 GridEngine batch scheduler instantiated and subjected to a synthetic load. The fastest boot times are for virtual machines with flash-cloned iSCSI roots (far left). 141

8.4 Fidelity is the percentage of the lease term usable by the guest, excluding setup costs. Xen virtual machines are faster to *setup* than physical machines, yielding better fidelity. 143

8.5 The implementation overhead for an example scenario for a single emulated site with 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. 145

9.1 Number of GridEngine tasks in each batch queue over time during a trace-driven execution. Note from the y-axis that the batch scheduler is experiencing intense constraint from the task load (2400 tasks at peak load) relative to the 71 machines that are available. 155

9.2 Number of machines in each of three virtual clusters over time during a trace-driven execution. Machines transfer to higher priority research groups as their task load increases. Strict priority arbitration results in machine reallocations to the highest priority research group whenever they have queued tasks. 156

9.3 Combined size of the Architecture running and pending task queues, and virtual cluster size over an eight-day period. 157

9.4 An example Plush application description that includes a request for resources from a Shirako broker/authority. 160

9.5 MyPLC and Jaws obtaining Xen virtual machines from the Shirako prototype. The prototype manages sharing of hardware resources between MyPLC and Jaws with a simple policy: MyPLC gets any resources not used by Jaws. 170

Chapter 1

Introduction

"*The sum is more than the whole of its parts.*"

Aristotle

Collections of hardware components are the foundation of computation and consist of interconnections of different types of the same core elements: processors, memory, disks, I/O devices, and network links. Since modern hardware components link to standard network interconnects it is possible to control and coordinate their management programmatically. An *operating system* is a software layer that manages hardware by coordinating its interaction with software: this thesis defines and evaluates an operating system architecture for managing collections of hardware components spread across networks, such as the Internet.

To avoid confusion, we use the term *networked operating system* to reference this category of operating system. We use the term *node operating system* to denote operating systems that centralize the management of hardware components for a single machine. A *physical machine* is a collection of *hardware components*, which may include one or more processors, disks, memory cards, and network cards, connected over a bus. Examples of physical machines include routers, switches, sensors, and mobile phones. We also use the term node operating system to encompass *distributed operating systems*, which decentralize conventional techniques to manage multiple machines by presenting the view of a single machine.

A networked operating system controls and coordinates the mapping of distributed applications to networked collections of hardware in infrastructures. As with node operating systems, a networked operating system determines the method by which software executes on collections of hardware including what software executes, when it executes, where it executes, and for how long. Managing collections of hardware encompasses a range of actions, which include one or more of the following: installing software, uninstalling software, updating software [49], starting/stopping software services, altering software configurations, diagnosing software performance problems [7, 141], monitoring software execution [35], scheduling software execution [170], or assigning software to execute on

specific hardware [98, 119, 162].

While an operating system for networked resource management must address a wide variety of problems, *its foundation ultimately rests on how applications share a common, networked hardware platform*. This thesis focuses on defining an architecture that serves as the foundation for a networked operating system that is able to address all of the software management subproblems mentioned above. Section 1.1 makes a case for networked operating systems by examining current problems in modern infrastructure management. Section 1.2 presents a simple solution to solve these problems, asks a series of focusing questions that shape our vision of a networked operating system, and defines a single design principle for building a networked operating system. Section 1.3 presents the hypothesis and contributions of this thesis and Section 1.4 presents an outline for the remainder of the thesis.

1.1 Problem

A data center is a collection of hardware components under the control of a single *sphere of authority* located in the same room or building. Administrators have the responsibility of installing and maintaining software in data centers. These administrators continuously monitor and adjust multiple, active software environments on a set of physical machines using one or more management tools [1, 126]. Administrators not only keep critical management software operational, users also request them to configure and reconfigure hardware for new uses. Managing multiple software environments on a diverse and growing computing infrastructure can exceed the abilities of even the most capable administrators.

We refer to any software environment, from a node operating system and its processes executing on a single machine to multiple node operating systems and their processes executing across multiple machines, as a *guest* of the hardware to emphasize the distinction between an infrastructure's hardware platform and the software that runs on it. We also use the term guest to disambiguate general software from node operating systems, processes, and applications.

A process is a programming abstraction defined by many node operating systems. There is no accepted definition of application in the literature; however, in this thesis, we use the term application to describe software running on one or more hardware components that performs a

specific function. *Distributed applications* execute across multiple physical machines. For example, a distributed hash table or DHT is a distributed application spanning multiple machines, each executing a node operating system, that provides one specific function: key-based data insertion and retrieval [155]. The term guest may include one or more node operating systems and their processes, running on one or more hardware components, and executing one or more distributed applications.

1.1.1 Infrastructure Complexity

An article from the New York Times captures the state of data center infrastructures in 2007 [115]. The article speculates on the size of a data center recently constructed by Google on the bank of the Columbia river in Oregon: "...the two buildings here—and a third that Google has a permit to build—will probably house tens of thousands of inexpensive processors and disks..." The same article posits that Google's current operation consists of "...at least 450,000 servers spread over at least 25 locations around the world." Microsoft's computing initiative is of similar scale; the infrastructure is "...based on 200,000 servers, and the company expects that number to grow to 800,000 by 2011 under its most aggressive forecast." The scale of these numbers is expected to increase over time—managing millions of machines at multiple locations is on the horizon [173].

Computing infrastructures are not only growing in physical size. Google's infrastructure, which is "...spread over at least 25 locations...," exemplifies an increase in geographical diversity that extends beyond any single data center. We use the term *infrastructure* to refer to a collection of hardware components under the control of multiple spheres of authority, such as 25 independently administered data centers, and the term *site* to refer to a collection of hardware components under the control of one sphere of authority, such as a single data center. The terms infrastructure and site are broad enough to refer to any collection of hardware. For instance, the network of independent autonomous systems that manage the core routers, switches, and network links that embody the Internet are a set of independent sites that comprise an infrastructure. Sensor networks composed of interconnected groups of sensors also follow the infrastructure-site paradigm. We use the description of Google's infrastructure to exemplify characteristics—large size and geographic dispersion—that are common across all infrastructures.

1.1.2 Impact of Virtualization

As infrastructures grow to accommodate more users and sites, the number of different types of guests and hardware components that they must support increases. Infrastructures are beginning to address the problem using virtualization. Virtualization platforms, such as Xen [22] and VMware [168], enable each physical machine to host one or more virtual machines.

Virtual machines expand an infrastructure's management options by decoupling the machine from the physical hardware. This decoupling makes hardware management more flexible and dynamic by giving site administrators, or trusted software acting on their behalf, the power to create and destroy virtual machines, bind isolated *slivers* of hardware components to virtual machines, and offer a wide array of *hardware management services* (see Section 2.2). However, virtualization's power introduces a new set of management challenges that exacerbate existing problems managing an infrastructure. For example, the virtual infrastructure can be an order of magnitude larger than the physical infrastructure since virtualization increases the number of manageable hardware instances. Furthermore, these instances may now be active or inactive (*i.e.,* have their live, in-memory state stored on disk) or have persistent state spread across multiple disks.

Many organizations maintain large collections of physical and virtual machines: universities, financial institutions, small businesses, hospitals, governments, and even individuals increasingly operate and depend on sizable, complex computing infrastructures. A recent Forrester Research report states that 50% of global 2000 firms will have server virtualization deployed by mid-2007 [77]. The range of different hardware management services a site administrator can leverage for each virtual machine is substantial, and, while each one of these services is useful for managing an infrastructure, invoking them manually at the request of users does not scale.

Virtualization technology is also spreading beyond data centers—recent work extends hardware virtualization to mobile devices [48], remote sensors [110], and network routers [24]. While virtualization increases management complexity, it also provides an element of the solution: rich, programmatic interfaces to control slivering and hardware management services. Broad adoption of virtualization, and its complexity, motivates a networked operating system that simplifies infrastructure resource management by leveraging these new capabilities.

1.2 Solution

One solution to decreasing infrastructure management's complexity is to automate resource management by giving guests enough control to obtain resources whenever and wherever they require them. Rather than requiring intervention by a site administrator to install, maintain, and protect guests, we envision a system where each guest, itself, continuously monitors its load, determines its resource requirements, requests and configures new resources, and incorporates those resources into its collective software environment. The infrastructure provides the means for guests and automated guest controllers to find, request, and control sets of resources programmatically.

Delegating resource management to guests permits flexible infrastructures that are able to scale to meet the diverse needs and requirements of many different guests. The solution not only unburdens site administrators, it also empowers guests. Guests benefit from the power to dynamically regulate their performance by requesting resources to react to changes in load, resource scarcity, or failures. Infrastructures benefit from increased efficiency through infrastructure-wide statistical multiplexing, if proper incentives exist to encourage guests to release resources when they do not require them.

1.2.1 Focusing Questions

While simply stated, the proposed solution raises a number of important questions that shape our vision of a networked operating system. The questions, and their answers, evolved from our experiences building and using multiple prototypes—they represent the basis for the set of networked operating system design principles we define and evaluate in this thesis.

How do we support a full range of guests? A networked operating system should support any guest capable of executing on its hardware, and should not limit the capabilities of the hardware by restricting support to a specific class of guests. For example, a guest of a data center may be a load-leveling batch scheduler [75, 113, 183], a network testbed [23, 142], a compute grid [69], a multi-tier web application [36], a single computational task, or a node operating system intended for general use.

How do we balance the resource needs of each site with the needs of each guest? Guests should be able to know how much resource they have and for how long (*e.g.*, a *lease*).

Designing a system around any weaker model does not directly support guests that require strict resource assurances to meet contractual obligations, which may promise clients a specific level of service (*i.e.*, a Service-level Agreement or SLA). In a utility computing model (see Section 3.3.2) guests exchange money to rent resources from an infrastructure [74]: it is unlikely that a system without resource assurances can be viable in a model that forces customers to rent a resource despite being unsure of what they are getting, when they are getting it, and how long they can use it. Similarly, an infrastructure needs resource assurances to know how much resource it is allocating to each guest and for how long to make informed decisions about how to multiplex resources between guests over time. For example, an infrastructure that rents resources to external guests may periodically wish to reserve resources for internal use.

How does an infrastructure support guests that require resources from multiple sites? A networked operating system should coordinate the allocation of resources owned and controlled by different spheres of authority to support guests that require resources from multiple sites. Guests, such as Content Distribution Networks [71, 172] or overlay routing systems [12], require resources from geographically disparate sites for correct operation. Furthermore, as infrastructures expand in physical size they also spread out geographically, with local site administrators controlling each site's resources. Both Google's computing infrastructure, which is "spread over at least 25 locations" around the world, and federated computing platforms, formed from resources donated by many autonomous sites, represent examples of infrastructures that span sites operating under different spheres of authority. Studies suggest that statistical multiplexing between sites is important since individual data centers experience long periods of low server utilization [118]. As a result, an infrastructure must support the coordinated acquisition of resources from multiple sites to guests: *brokering services*, proposed by SHARP [73], address multi-site resource sharing by aggregating resource rights from many sites and coordinating their distribution to guests.

How do guests interact with an infrastructure to acquire and control resources? Guests should interact with infrastructures and resources programmatically using network-accessible APIs. Each guest has a controlling server, which we call *service manager*, that monitors, requests, and controls guest resources programmatically. Each site operates a server, which we call an *authority*, that exports a network-accessible API for guests to obtain and control resources. The authority

fulfills guest resource requests and isolates resources from co-hosted guests. Brokering services export a network-accessible API to programmatically receive and respond to service manager resource requests on behalf of multiple sites. We collectively refer to service managers, authorities, and brokering services as *actors* in a networked operating system.

How does an infrastructure arbitrate resources between guests when there are not enough resources for everyone? Infrastructures require a method for deciding the priority of guest resource requests. Markets are an attractive basis for arbitration in a networked infrastructure composed of, potentially self-interested, sites and guests that define natural incentives for guests to coordinate resource usage by associating usage with a price. Price provides a feedback signal that enables guests to respond to price changes and self-regulate their resource usage. However, market-based resource allocation is still an active research area with many unsolved problems that continue to prevent its transition from research prototypes to usable systems [152]. A networked operating system must include a basis for defining the incentives and controls that support evolving market-based technologies as well as other, more conventional, forms of resource arbitration, such as proportional-share or priority-based arbitration.

How do we design a system that is robust to faults and failures? A networked operating system should be resilient to intermittent server or network failures. We advocate distributing control of infrastructure resources across multiple actors: the service manager controlling each guest's software, the authority controlling each site's hardware, and brokering services acting on behalf of, or coordinating between, each guest and site. These actors, or the network linking them, may fail at any time, independently of the resources they control. Service managers, authorities, and brokering services must retain sufficient state to recover from actor failures, and define protocols for synchronizing their state with the state of their peers to tolerate periodic disconnections due to actor or network failures. Failures of actors, resources, guests, or the network must not compromise forward progress in the collective system.

1.2.2 The Neutrality Principle

In this thesis we explore a software architecture that embodies the answers to the questions above. The architecture defines a type of operating system that mediates the interaction between guests and

networked collections of hardware. Historically, operating systems have served two functions: they multiplex resources to simultaneously execute multiple applications and they provide programming abstractions to ease application development. We explore how a networked operating system should address these two functions by examining the following questions.

- What is the right set of programming abstractions for a networked operating system?
- How does an infrastructure use these programming abstractions to multiplex resources?

The answers lead us to employ a general *neutrality principle* in our design. The neutrality principle states that an operating system for networked resource management should be policy-neutral, since only users and site administrators, and not operating system developers, know how to best manage resources. In this thesis, we focus on how programming abstractions define policy. The principle is a restatement of the end-to-end principle for networked operating system design [145].

Exokernel, discussed in Chapter 2, applied a similar neutrality principle to node operating system design to eliminate programming abstractions, after arguing that these programming abstractions represent subtle, but ill-advised, embeddings of resource management policy [61]. A contribution of this thesis is the recognition that problems arising in the design of node operating systems are similar to problems arising in the design of networked operating systems. As a result, studying and adapting structures and techniques for designing node operating systems are able to resolve existing problems in networked resource management. Extensive experience building and using a networked operating system has led us to view neutrality as a guiding design principle. The goal of this thesis is to define a minimal set of elements common to all networked operating systems that are independent of any specific set of resource management policies or programming abstractions.

We find that a fundamental property common to all forms of networked resource sharing which does not, directly or indirectly, embed resource management policy is that guests, by definition, *share resources and do not use them forever*. We use the term *resource* to refer to a reusable aspect of the hardware, and do not intend for the phrase "resource sharing" to include "file sharing," and other similar forms of sharing, that describe higher-level constructs for using a resource. To address the time aspect inherent to resource sharing, we examine an operating system architecture for networked infrastructure based on a lease abstraction, which restricts guest resource use to well-defined, finite lengths of time.

A networked operating system that does not define resource management policy is capable of supporting any type of guest or resource. Chapter 2 argues that strict adherence to the neutrality principle requires that the lease abstraction be flexible enough to manage resources down to the level of the hardware components that comprise physical and virtual machines.

1.3 Hypothesis and Contributions

This thesis evaluates the following hypothesis: "An operating system architecture for infrastructure resource management that focuses narrowly on leasing control of hardware to guests is a foundation for multi-lateral resource negotiation, arbitration, and fault tolerance." In evaluating our hypothesis, we make the following specific contributions.

1. Introduce a set of design principles for networked operating systems based on our experiences building and using multiple prototypes. The design principles stem from the same motivation as Exokernel, which states that a node operating system should eliminate programming abstractions and concentrate solely on resource multiplexing. We show how to adapt and extend Exokernel design principles to a networked operating system that multiplexes collections of resources owned by sites spread across networks, such as the Internet.

2. Combine the idea of a reconfigurable data center from Cluster-on-Demand [39] and others [5, 15, 17, 97, 99, 116, 144, 177] with the Sharp framework [73] for secure resource peering to demonstrate a networked operating system architecture capable of sharing resources from multiple data center sites between multiple guests. We extend both models by developing the architectural elements required to adapt and control any type of guest using any type of resource.

3. Design, implement, and deploy Shirako, a prototype networked operating system, using a single programming abstraction: the lease. The lease abstraction embodies Exokernel design principles, including visible resource allocation/revocation, an abort protocol, exposing names, exposing information, and secure bindings, and applies them to a networked operating system that decentralizes control of resource allocation across multiple spheres of authority. We use

our experiences deploying the architecture to motivate extensions to SHARP and Cluster-on-Demand that expose the names of distinct physical machines, virtual machines, and slivers.

4. Show that leases are a foundational primitive for addressing arbitration in a networked operating system that supports a range of different arbitration policies, including market-based policies, which are of particular importance for a system composed of self-interested actors. Leasing currency defines a configurable tradeoff between proportional-share scheduling and a market economy.

5. Leases are a common tool for building highly available, fault-tolerant distributed systems, and have long been a basis for addressing network and server failures in distributed systems [107]. We show how combining the use of leases for long-term resource management with state recovery mechanisms provides robustness to transient faults and failures in a loosely coupled distributed system that coordinates resource allocation.

6. Evaluate the flexibility and performance of Shirako by sharing physical and virtual hardware present in modern data centers between multiple types of guests, and showing that the architecture could scale to manage thousands of machines.

7. Present case studies of integrating multiple guests that manage networked resources to evaluate our approach. Guests include the PlanetLab network testbed [23], the Plush distributed application manager [9], and the GridEngine batch scheduler [75]. We also leverage the architecture to prototype and evaluate Jaws, a policy-free batch scheduler that instantiates one or more virtual machines per task.

1.4 Thesis Outline

Chapter 2 details the philosophy that underlies the neutrality principle by drawing parallels to previous work on Exokernel [61]. Chapter 3 examines related work in classical resource management and its relationship to a networked operating system. In particular, we discuss: resource management at each layer of the software stack, networked management of multiple software stacks, and classical elements of distributed systems design. Chapter 4 presents a networked operating system architecture that employs the neutrality principle to define a general lease abstraction by combining

and extending the key design principles of Exokernel, Sharp, and Cluster-on-Demand. Chapter 5 uses our experiences deploying the architecture to motivate extensions to Sharp and Cluster-on-Demand that expose the names of distinct physical machines, virtual machines, and slivers. Chapter 6 shows how leases are a basis for resource arbitration policies and Chapter 7 discusses the architecture's use of leases for fault tolerance. Chapter 8 discusses the implementation of the architecture within Shirako, and quantifies its flexibility and performance. Chapter 9 presents case studies of integrating three existing guests and developing one new guest with Shirako's architecture. Chapter 10 concludes.

1.5 Acknowledgements

The starting point for the architecture and Shirako derives from ideas originally present in Cluster-on-Demand [39], led by then graduate student Justin Moore, and Sharp [73], led by then graduate student Yun Fu. Shirako is a group effort that serves a broader range of research goals than discussed in this thesis. The development of Shirako is part of a larger project, called Orca, that encompasses the study of other elements that comprise a fully functional networked operating system.

As described in later chapters, Shirako includes an interface to plug in multiple types of resources and guests as well as extensible interfaces for implementing and experimenting with different policies. Since Shirako's development is a shared effort, multiple people have contributed over the years to the different guests, resources, and policies that fill out the architecture, and we acknowledge their contribution. Laura Grit led the development of broker provisioning/arbitration policy implementations as well as a general substrate for developing new lease-based adaptation, assignment, and arbitration policies. Aydan Yumerefendi led the development of the first assignment policies that used the *modify* resource handler, discussed in Section 4.3.2, to migrate virtual machines and alter resource shares. These policies and their interactions are outside the scope of this thesis.

Varun Marupadi led the transition from using an LDAP database for persistent storage to using a MySQL database, as well as completed the integration of NFS/ZFS as a usable Shirako storage device. Varun Marupadi also ported Jaws and GridEngine to use a new guest packaging format developed by Aydan Yumerefendi that permits the dynamic creation and linking of guest-specific service managers within the Automat web portal [185]. Piyush Shivam and Aydan Yumerefendi

led the integration and testing of a Rubis-aware service manager [185] and Lavanya Ramakrishnan spearheaded the integration of a Globus-aware service manager [138], both of which we mention, but do not detail, in this thesis. Aydan Yumerefendi led the development and/or redevelopment of multiple pieces of software including a node agent for easier integration of resources, a model-checker for finding code bugs and validating lease state transitions, and a sophisticated web portal for uploading and testing new management policies [185]. Aydan Yumerefendi has significantly augmented the persistence architecture to include persistence of state other than core slice, lease, and policy state to build the web portal for Automat [185].

The implementations of different guests, resources, and databases have been written and re-written multiple times over the years in order to incorporate updates in resource and guest technologies, new constraints imposed by higher levels of the architecture, and more robust interfaces for improved logging and debugging. Since all of Shirako's various pieces are interdependent a number of people have touched this code; however, Aydan Yumerefendi and I have led the development of these changes over the last 4 years. Matthew Sayler has recently taken Shirako's ad hoc rules for event logging and formalized and rewritten the event logging subsystem to support work on better classification, diagnosis, and repair of failures. David Becker significantly aided in the understanding of multiple data center administration technologies, most notably LDAP and Xen, and provided support in tracking down the root cause of many guest-specific and resource-specific bugs and failures.

The insights in this thesis stem from lessons I learned through my distinct experiences, beginning with the original Cluster-on-Demand prototype in the early fall of 2002 up to the present day, spearheading the implementation of the lease as a programming abstraction for managing guests, resources, and persistence in a deployed system; these experiences required exercising and stressing every aspect of the architecture, including integrating a multitude of specific guests and resources, developing new adaptation policies and monitoring engines tailored for each guest/resource combination, modifying and enhancing arbitration policies to perform guest-specific resource sharing, developing a persistence architecture for lease and policy state, writing resource assignment policies and developing resource-specific configuration directives (*e.g.*, IP address allocation and domain naming), and incorporating new disk images and monitoring software to support guests and sites

beyond the realm of Duke's Computer Science Department. The integrative process I engaged in spanned the entire architecture and allowed me to pursue my own distinct goal of gaining new insights into networked operating system design through the experience of supporting real users using real guests and real resources under multiple different management scenarios for a sustained period of time.

Chapter 2

The Extensible Operating System Approach

"*Because the system must ultimately be comprehensive and able to adapt to unknown future requirements, its framework must be general, and capable of evolving over time.*"

Corbató and Vyssotsky on Multics, 1965

Node operating system researchers have long realized the benefits of a flexible kernel that application developers can extend to add new functionality: our approach to a networked operating system stems from the same motivation [4, 27, 61, 108, 160, 182].

In Section 2.1 we argue that a networked operating system should concern itself only with resource multiplexing and not the definition of programming abstractions that ease guest development but restrict flexibility. Guests are similar to node operating system applications and represent software environments that execute on a hardware platform. We draw parallels between our approach and previous work on microkernels, and outline the design principles of Exokernel, which explored the separation of resource multiplexing from programming abstractions in the design of a node operating system. Section 2.2 describes how the architecture in this thesis extends Exokernel's design principles to a networked operating system.

2.1 Relationship to Node Operating Systems

The landscape of networked computing today is analogous to node operating systems in key ways. There are multiple existing platforms for resource sharing and distributed computing with large user communities. We are again searching for an evolutionary path that preserves and enhances existing programming environments, accommodates innovation in key system functions, and preserves the unity and coherence of a common hardware platform. Finding the elemental common abstractions or "narrow waist" of the architecture that supports all of its moving parts remains an elusive goal.

Node operating systems serve two purposes: they multiplex resources among multiple guest applications and they provide programming abstractions that ease guest development. Programming abstractions represent functionality embedded into a node operating system that simplify

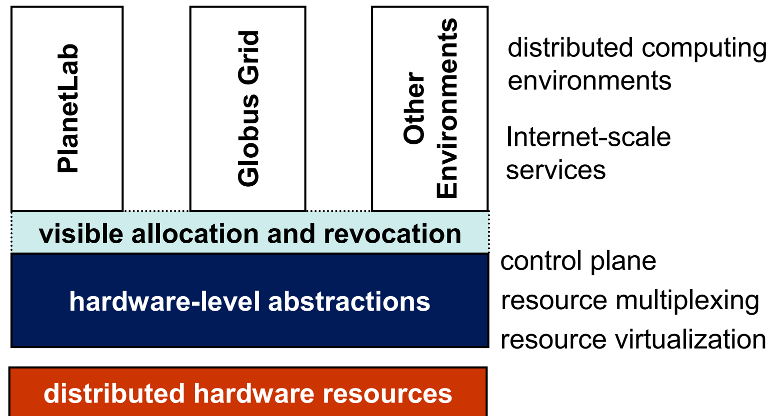

Figure 2.1: Distributed computing environments, such as PlanetLab and Globus Grids, are built above hardware-level resource abstractions exported by resource owners.

guests' interaction with hardware. The implicit assumption is that it is appropriate to embed these programming abstractions since they are useful for the majority of guests.

To support a diverse range of guests, we advocate programming abstractions that focus on resource multiplexing and control. A narrow focus on resource multiplexing is able to accommodate a wide range of scenarios. For example, the GENI initiative proposes an international network testbed facility that multiplexes network resources—including data centers, mobile devices, sensors, storage, and network links—among experimental prototypes for a new generation of Internet applications and services [131]. These network services may eventually define a "new Internet" that is more flexible and open than today's "best-effort" Internet. However, to support new network services, GENI must wrestle with core operating systems issues such as isolation, coordinated multiplexing of a range of hardware resources, and programming abstractions for distributed computing. The "future Internet" must provide a platform for planetary-scale applications not yet imagined, absorb technologies not yet invented, and sustain decades of growth and "disruptive innovation" without fracturing or collapsing under its own weight.

A stated motivation behind both PlanetLab and GENI (and Active Networks before them [174]) is overcoming the "ossification" of the existing Internet that prevents the adoption of new network services [129, 132]. The Internet is not a suitable platform for innovations in high-bandwidth, ultra-reliable, and ultra-secure technologies because it hides/abstracts details and control of resources in the network's interior from the services that execute on the network's edge resources. As a result,

a resource management layer that exposes control of resources is a cornerstone of any new type of Internet-scale operating system that seeks to satisfy the resource needs of all guests.

As depicted in Figure 2.1, we argue that a networked operating system must support multiple guests over collections of networked hardware in such a way that sites can redeploy hardware easily from one guest to another, rather than dedicating static islands of resources to each guest, as is common today. This "pluralist" approach has a familiar justification: given the wide diversity of needs among users, it is useless to wait for one system to "win" [52, 62]. Also, pluralism enables innovation and customization—we want to be able to experiment with multiple approaches to distributed computing, not just one, and to provide a richer and wider set of experiences than any single platform can offer.

2.1.1 Lessons from Microkernels

The problems that researchers face designing an Internet-scale networked operating system, as proposed by GENI, also arise in a different context in the design of node operating systems. The motivation for microkernels and extensible kernels are similar to our motivation for a networked operating system: the "kernel" of the system should not restrict the range of guests it supports. In this section, we review lessons from research on microkernels and highlight the motivations, arguments, and principles behind the design of Exokernel, a variant of microkernel research, that espouses a clean separation of resource multiplexing from programming abstractions [60, 61, 96].

Microkernels are a node operating system structure that shifts some functionality outside of the kernel. Adopting a microkernel structure for a node operating system reduces the size of large monolithic kernels, that contain an array of device drivers, file systems, and other subsystems, to a small core that only coordinates hardware multiplexing and communication between privileged subsystems. The idea underlying microkernels is that shifting functionality outside of the kernel creates a more stable and reliable structure by extracting complex subsystems and executing them in user-space. As a result, subsystems that are known to be unreliable, such as device drivers, are unable to compromise correct system operation.

While early microkernel systems, such as Mach [4], extracted key subsystems from the kernel, the kernel retained control over resource management policy by preventing unprivileged guests from

modifying kernel subsystems. Exokernel is a variant of the microkernel structure that addresses this limitation by advocating a strict division between a node operating system's kernel, which multiplexes resources, and its guests, which implement programming abstractions. The division is a logical extension of the key microkernel design principle that a "...concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality [111]." The Exokernel structure views user-level kernel subsystems as privileged extensions of the kernel that prevent guests from modifying or replacing them, as depicted in Figure 2.2.

2.1.2 Exokernel

Exokernel, as well as other extensible operating systems [27], apply the neutrality principle to a node operating system by designing a kernel that does not impede the implementation and execution of a diverse range of guests by predefining a set of programming abstractions. We review the arguments behind the Exokernel architecture and the minimal set of design principles it defines. In Section 2.2, we relate how an extensible networked operating system applies and extends these arguments and design principles to a new context.

A summary of the argument for Exokernel, taken from the first paragraph of [61], is as follows.

> *"Operating systems define the interface between applications and physical resources. Unfortunately, this interface can significantly limit the performance and implementation freedom of applications. Traditionally, operating systems hide information about machine resources behind high-level abstractions such as processes, files, address spaces and interprocess communication. These abstractions define a virtual machine that applications execute on; their implementation cannot be replaced or modified by untrusted applications. Hardcoding the implementations of these abstractions is inappropriate for three main reasons: it denies applications the advantages of domain specific optimizations, it discourages changes to the implementations of existing abstractions, and it restricts the flexibility of application builders, since new abstractions can only be added by awkward emulation on top of existing ones (if they can be added at all)."*

An Exokernel is a secure hardware multiplexer that allows untrusted guests to use the hardware

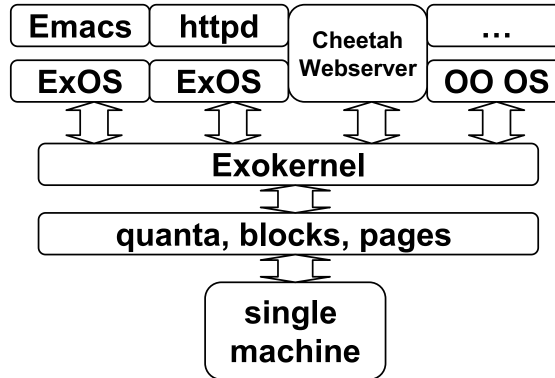

Figure 2.2: An overview of Exokernel's structure from [61]. The kernel acts as a secure multiplexer of hardware, in the form of CPU quanta, disk blocks, and memory pages, for multiple guests. The kernel does not define programming abstractions such as file systems, processes, threads, or virtual memory that restrict guests' use of the hardware. Guests are free to use the hardware directly to define their own programming abstractions, or link against library operating systems that provide a set of common programming abstractions.

directly to implement their own programming abstractions, or link against libraries that implement node programming abstractions at application level. As with other microkernels, Engler et al. argue that the approach results in a more minimal, secure, and reliable kernel compared with monolithic kernels since it does not require implementing complex programming abstractions. The Exokernel philosophy is in keeping with Butler Lampson's hints for computer systems design, as stated below [106].

> *"Dont hide power. ...When a low level of abstraction allows something to be done quickly, higher levels should not bury this power inside something more general. The purpose of abstractions is to conceal undesirable properties; desirable ones should not be hidden. Sometimes, of course, an abstraction is multiplexing a resource, and this necessarily has some cost. But it should be possible to deliver all or nearly all of it to a single client with only slight loss of performance. ...Leave it to the client. As long as it is cheap to pass control back and forth, an interface can combine simplicity, flexibility and high performance by solving only one problem and leaving the rest to the client."*

Exokernel asserts that guests "desire" control of the hardware, and that a node operating system should solve only the "one problem" of secure hardware multiplexing, thereby "leaving the rest" of a node operating system's job to the guest. Research on Exokernel design distilled a minimal set of

design principles to separate programming abstractions from hardware multiplexing.

- **Fine-grained Resource Allocation.** Exokernel allocates resources at the finest granularity offered of the hardware, such as CPU quanta, memory pages, and disk blocks. The choice to offer access to raw partitions of the hardware is in keeping with a strict adherence to the neutrality principle: hiding hardware behind any type of higher-level abstraction hinders flexibility and prevents guest-specific performance optimizations.

- **Visible Allocation and Revocation.** An Exokernel explicitly notifies each guest when it allocates or revokes a resource, arguing that masking any type of resource allocation or revocation is a form of abstraction. Explicit notifications provide guests an opportunity to adapt their behavior in accordance with their resource allotment. For example, a guest that loses a memory page requires a notification of the page's revocation to know that it must swap the page to disk [87].

- **Secure Bindings.** Exokernel uses secure bindings to associate each resource with a single guest and track ownership. Secure bindings prevent competing guests from accessing or modifying a resource by restricting resource access to the owning guest. They do not significantly impact performance because bindings are set only at the time of resource allocation and revocation.

- **An Abort Protocol.** Exokernel uses an abort protocol to break a secure binding if a guest does not voluntarily relinquish control of a revoked resource. The abort protocol preserves the Exokernel's right to revoke any resource at any time.

- **Exposing Names and Information.** Exokernel exposes physical names of hardware resources, such as disk blocks and memory pages, whenever possible to remove a layer of indirection. A corollary to exposing physical names is exposing system-level information that guests cannot easily derive locally, but require to efficiently utilize the hardware, such as revealing the number of hardware network buffers [61].

Principle	*Exokernel*	*Shirako*
Hardware Multiplexing	Exokernel allocates at the finest granularity of the hardware: CPU quanta, memory pages, and disk blocks.	Shirako allocates hardware at the granularity of physical and virtualized hardware devices.
Visible Allocation/Revocation	Exokernel notifies guests when it allocates or revokes a CPU quanta, memory page, or disk block.	Shirako notifies guests when it allocates, revokes, or modifies a physical or virtual hardware device.
Secure Bindings	Exokernel binds guests to hardware and tracks ownership.	Shirako binds guests to physical or virtual hardware devices and tracks ownership.
Abort Protocol	Exokernel uses an abort protocol to break a secure binding.	Shirako uses an abort protocol based on leases, which define a natural termination point for resource revocation.
Exposing Names/Information	Exokernel exposes the names of hardware, such as memory pages and disk blocks.	Shirako exposes the names of slivers, logical units (*e.g.*, virtual machines), and hosts.

Table 2.1: The principles underlying Exokernel and Shirako are similar. Both systems focus on hardware multiplexing, visible allocation/revocation, secure bindings, an abort protocol, and exposing names.

2.2 An Extensible Networked Operating System

As with node operating system kernels, a networked operating system involves two different but intertwined issues: resource multiplexing and programming abstractions. Existing systems for networked resource management, like conventional node operating systems, implement programming abstractions that violate the neutrality principle. For example, distributed operating systems, such as Amoeba [160], extend node operating system abstractions across a collection of machines. Like Exokernel, we argue that a networked operating system should strive to separate these abstractions from physical resource management.

The same principles that govern node operating system design also apply to managing networked hardware. We adapt Exokernel principles to a networked operating system to expose control of collections of resources to guests, giving them the power to manage a subset of resources in a multi-site infrastructure. The key design tenets of Exokernel are also found in our architecture. We explore how the same principles apply to the allocation of collections of physical and virtual machines. Table 2.1 summarizes the relationship between Exokernel design principles and the design principles of Shirako, our prototype networked operating system.

Guests require visible resource allocation and revocation to adapt the dynamics of their resource allotment, as described in Section 4.3.2, and sites require an abort protocol to break a guest's binding

to resources, as described in Section 4.3.3. In Section 5.3, we discuss how to adapt the principles of exposing names and information to manage collections of hardware. In Section 5.4.2, we discuss secure bindings of guests to resources. Despite the similarities to node operating systems and Exokernel, networked resource management is a fundamentally different environment with its own distinct set of characteristics. Below we outline characteristics of a networked operating system that motivate our design principles. We detail the motivation behind each principle in the subsequent subsections.

- **Multiplex Collections of Networked Resources.** The heart of a networked operating system is the allocation of collections of resources linked across a network, and not simply the resources of a single machine. The distributed nature of the resources results in an execution environment that binds guests to resources using network-accessible APIs and not intra-machine software bindings.

- **Allocate Resources across Multiple Sites.** Unlike a node operating system kernel, no single entity in a networked operating system controls all the resources; instead, the system coalesces and multiplexes resources from multiple, independent sites.

- **Coarse-grained Resource Allocation.** Allocating resources at the level of CPU quanta, memory pages, and disk blocks is too fine-grained for a networked operating system. We argue that a coarser-grained approach is more appropriate for guests that control collections of resources. Physical or virtual machines represent coarse-grained hardware allocations that retain a hardware abstraction. Previous systems use the term *slice* to refer to coarse-grained collections of networked resources [29, 23, 73].

2.2.1 Multiplex Collections of Networked Resources

We depict a comparison between Exokernel and an extensible networked operating system in Figure 2.3. The collection of brokers and authorities communicate to coordinate resource allocation using Exokernel's principles [73]. Service managers are akin to library operating systems, and interact with brokers and authorities to coordinate allocation for guests.

A consequence of allocating networked collections of resources is that the actors in the system control the binding of resources across a network, using network-accessible APIs. As a result, the

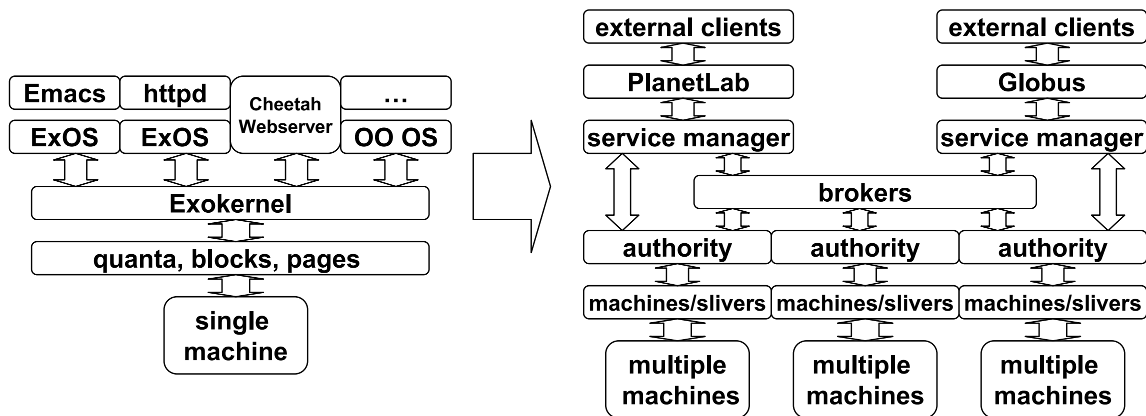

Figure 2.3: The figure compares the structure of Exokernel with a networked operating system. The primary design differences stem from a focus on multiplexing collections of resources from multiple sites at the coarser granularity of physical machines, virtual machines, and slivers. The focus motivates both a physical separation between the service manager, the authority, and the guests and resources they control, and a logical separation of policy between multiple authorities and brokers.

service managers, which control the guests, and the authorities, which control the resources, do not execute on the machine (or machines) they are controlling, unlike a node operating system guest or kernel. The interface exported by each authority must be network-accessible so external service managers can invoke them. Further, interfaces must exist for the authority, itself, to control the allocation and revocation of resources. Virtualization platforms provide the interfaces necessary to sliver resources and present a hardware abstraction.

Another result of the separation of service managers and authorities from the guests and resources they control is a reduced emphasis on the method of resource partitioning. The functions that coordinate resource multiplexing and control are physically separate from the guests and resources being multiplexed. As we describe in Section 4.3.1, it is possible to apply Exokernel principles to networked resources in a general way that is independent of a specific guest or resource.

2.2.2 Allocate Resources across Multiple Sites

Multiple sites own and control networked resources in infrastructures. A networked operating system must coalesce these resources into a coherent system that offers them to guests. We leverage two design elements for a networked operating system that address multi-site resource sharing, as described below.

- **A Lease Abstraction.** Leases are an effective tool for multi-actor resource negotiation, arbitration, and fault tolerance that are compatible with Exokernel's design principles.

- **A Division of Policy and Control.** In order to offer resources to guests from multiple sites, sites must relinquish partial control over their allocation policy to some entity. We examine how the SHARP model for resource peering defines this separation of policy and control, and how the separation is compatible with the neutrality principle.

The Lease Abstraction

We identify the lease as an elemental programming abstraction for multi-actor resource negotiation, arbitration, and fault tolerance. From the start, Exokernel assumes that a single trusted kernel allocates and revokes resources from guests at any time. In a networked operating system, control of guests resides outside of any site's sphere of authority. As a result, negotiation between guests and sites is unavoidable—sites need to know how much resource they are sacrificing and for how long to prevent overcommitting their resources (or to control the degree of overcommitment [73]), and guests require an assurance over the resources they hold to accurately plan for the future. Knowledge of resource commitments is especially important if guests exchange payment for resources. As a result, we examine a design built around a lease abstraction, which serves as the basis for all multi-party resource negotiation and exchange.

Sites and guests use the lease abstraction to agree on the duration of resource use *a priori*. Exokernel has no parallel to the lease abstraction because, unlike a networked operating system, its design is bound to the resources it multiplexes since fine-grained allocation requires resource-specific techniques to ensure good performance. The physical separation of service managers and authorities from the resources they control combined with a focus on coarse-grained allocation permits the management of guests and resources under a single programming abstraction.

Exokernel also does not address the time aspect of resource sharing: guests inevitably use each resource for a limited period of time. Instead, Exokernel implements a programming abstraction that only defines that a resource is given and then taken away. This abstraction is weaker than a lease abstraction and does not have the requisite benefits for negotiation (see Chapter 4), arbitration (see Chapter 6), and fault-tolerance (see Chapter 7).

It is the time element, which is so fundamental to resource sharing, that presents the strongest argument for choosing the lease abstraction as "the tie that binds" a networked operating system together. Despite being fundamental to both resource sharing and distributed system design, we know of no other system that ties together all aspects of resource management, including extensibility, arbitration, and fault tolerance, under a single programming abstraction.

A Division of Policy and Control

We adopt the Sharp protocol for secure and accountable resource sharing between multiple sites. Note that, in contrast to node operating systems like Exokernel, Sharp has no globally trusted core; rather, an important aspect of Sharp is a clear factoring of powers and responsibilities across a dynamic collection of participating actors that communicate to control resources.

Sharp's framework goes beyond Exokernel's policy-free design by placing the arbitration policy for any type of resource in an external brokering service. A Sharp brokering service aggregates resource rights from multiple sites and collectively coordinates their distribution to guests to enable multi-site resource allocation. While Exokernel removes the scheduling and arbitration policy of the CPU from the kernel by including a primitive for processes to yield their processor time to another specific process, it does not include a similar primitive for other resources, such as memory.

Since Sharp brokering services allocate resource leases with a well-defined start and end time, they also have the power to delegate service managers (or other brokering services) the present or future *right to a resource*. Allocating the future right to use a resource is a foundation for implementing sophisticated resource scheduling policies, such as advance reservations or futures markets. Defining arbitration policies for a networked infrastructure composed of independent guests and sites is more challenging than developing policies within a single kernel that owns all resources for all time. In Chapter 6 we show how leases are a basis for a currency system that provides a foundation for market-based allocation policies.

2.2.3 Coarse-grained Resource Allocation

Modern virtualization technology provides the power to bind performance-isolating reservations and/or shares of individual hardware components to each virtual machine. Site administrators, or

trusted software acting on their behalf, may precisely allocate resources to virtual machines along multiple resource dimensions of CPU, memory, I/O bandwidth, and storage. We use the term *sliver* to describe a bundle of resources from multiple hardware components on the same physical machine.

Virtual machine performance depends on its sliver: if a virtual machine attempts to use more resource than its sliver provides, and there are no surplus resources to permit it to burst, then performance isolation dictates that the guest will suffer, but other co-resident virtual machines will receive the resources promised to them (*i.e.*, minimal crosstalk). Slivering may behave unfairly in corner cases [82]; ongoing research is developing techniques for reducing virtualization's performance overhead are ongoing [117]. Characterizing the fundamental performance overheads and slivering capabilities of virtual machines is an open research question. We view the performance overheads of existing virtual machines as an acceptable cost for increased management flexibility, and the slivering capabilities of existing virtual machines as acceptable for a range of scenarios that require precise sliver isolation, as described in [136].

We advocate a structure where sites offer entire physical machines or slivers bound to virtual machines to guests, rather than offering them individual CPU quanta, memory pages, or disk blocks. We sacrifice some of the flexibility and performance improvements Exokernel offers on a single machine to support commodity mechanisms for physical and virtual hardware management. Engler et al. note that a virtual machine implements the perfect Exokernel interface, except that it hides fine-grained resource allocation/revocation from each virtual machine [61]. The benefit of providing this level of precision to guests comes at a cost in complexity. For example, Exokernel requires complex techniques, such as downloading and sandboxing code in the kernel and defining untrusted deterministic functions (or *udfs*), to ensure high performance at the lowest level of multiplexing for the network and disk [96].

Hardware virtualization bypasses the complexities of downloading and sandboxing code inside the kernel (a technique present in both Exokernel [61] and SPIN [27]) by combining the secure presentation of hardware to unaltered guests with the simplicity of masking fine-grained resource allocation decisions. An authority that delegates control of hardware to a guest may expose a network-accessible API to control booting or rebooting a machine, loading software, such as a node operating system kernel, onto a machine, linking the software running on one machine to software

running on an external machine, such as a remote storage device, or binding slivers to a virtual machine.

With coarser-grained allocation, guests that implement programming abstractions that require finer granularity over per-machine resources may install software that exports their hardware at the proper granularity and handles allocation and revocation of machines. Global memory management is an example of a distributed shared memory protocol that provides remote access to memory pages and supports the dynamic addition and removal of machines [63].

2.3 Summary

In this chapter, we parallel our approach to networked operating system design with previous work on node operating systems. In particular, we describe how our approach is reminiscent of Exokernel and its design principles. We outline Exokernel design principles and describe how we extend them to manage networked resources.

Exokernel principles were never widely deployed in commodity node operating systems because the need for increased flexibility at the OS-level never materialized, and the focus on extensibility was feared to be "leading us astray [56]". In retrospect, interposing on fine-grained resource management is expensive, and machine resources became cheap enough to dedicate or partition at a coarse grain. [59] provides a comprehensive description of the barriers to the architecture's adoption. While the Exokernel approach was not adopted for node operating systems its principles are well-suited to systems that manage collections of resources. These systems must accommodate multiple types of guests and resources, as well as a rich set of arbitration and assignment policies. Networked resource management may use a coarser-grained model of resource allocation to alleviate concerns over implementation overhead and complexity that make extensibility difficult for node operating systems.

Chapter 3

Classical Resource Management

"*Operating systems are like underwear–nobody really wants to look at them.*"

Bill Joy

Figure 3.1 depicts a modern software stack. At the lowest level of the stack is the physical hardware, which the programmer cannot alter. A node operating system executes on the physical hardware and multiplexes and abstracts hardware for applications. Virtualization layers now run underneath a node operating system to increase the flexibility of a rigid physical hardware layer by providing server consolidation, configuration and resource isolation, and migration. Finally, applications execute above programming abstractions exported by the node operating system. In particular, middleware is a class of application that defines new programming abstractions, extending those provided by the node operating system.

Virtualization is a building block for programmatic resource management. However, virtualization requires external policies to drive its mechanisms in accordance with guests' needs. Node operating systems manage the resources of individual machines, but OS-level scheduling policies and programming abstractions do not readily extend to networked resource management. Middleware welds together a diverse set of networked resources into a uniform programming environment by masking differences in the lower levels of the stack, but since it runs above node operating systems it cannot predict or control resources allocated by lower layers, unless the node operating system exports that control.

Our approach represents a new layer in the software stack that provides the hooks to map guests, such as middleware and node operating systems, onto physical and virtual hardware. We call this new layer, which exposes rather than masks resources, *underware* to emphasize that it serves as a control plane that operates below middleware, node operating systems, virtual hardware, and physical hardware. Underware is a general term for the network of service managers, authorities, and brokering services that communicate to collectively control resource management functions for software stacks executing on collections of networked hardware components. Note that the

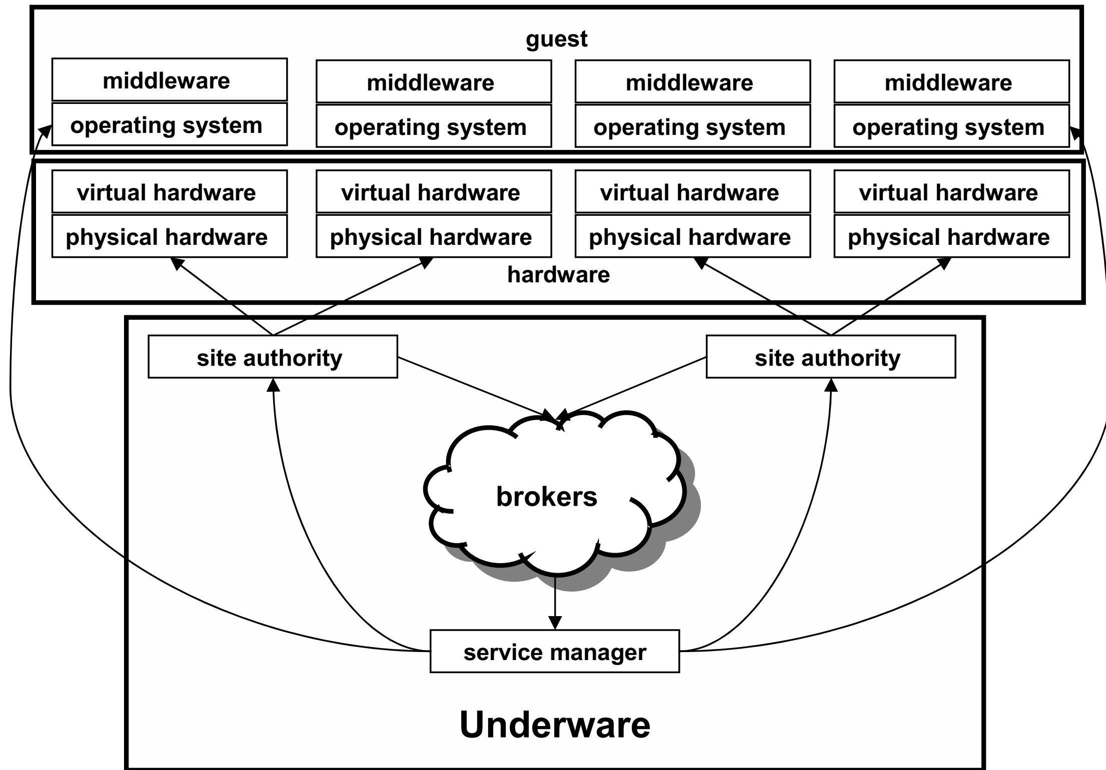

Figure 3.1: Physical and virtual hardware, node operating systems, and middleware are building blocks for a new underware layer that programmatically maps software onto hardware. Authorities control the binding of the resources they own to the service managers that request them. Service managers use network-accessible APIs exported by each authority to control the mapping of software (node operating systems, middleware, etc.) for their guests onto each site's hardware. Brokers represent the cohesive entity that coalesces resources from multiple sites and offers them to service managers.

underware layer includes only those entities that control hardware, and does not include either the physical or virtual hardware controlled by authorities, or the node operating systems and middleware controlled by service managers. We summarize below.

- **Underware.** Underware is a software layer that multiplexes resources by coordinating the mapping of guest software onto collections of networked hardware.

- **Hardware Virtualization.** Hardware virtualization executes beneath node operating systems and provides the illusion of an isolated virtual machine. Sites use hardware virtualization to partition physical resources without sacrificing the flexibility of the hardware.

- **Middleware.** Middleware is built above node operating systems and provides an environment that masks differences in the underlying hardware and node operating system to enable a

homogeneous execution environment.

Below we further detail the capabilities of each layer of the software stack and describe its relevance to a new underware layer.

3.1 Hardware Virtualization

Machine virtualization, first proposed in the mid-1960s [50], experienced a renaissance in the late 1990s with support for x86 processor virtualization [22, 168]. Since that time virtualization has become a common tool for managing machine, network, and storage resources. Virtualization provides mechanisms to create and manage virtual hardware, and offers more management flexibility than physical hardware, which cannot be created or altered after fabrication.

Improvements in virtualization technology act as a catalyst for a networked operating system by enriching the space of possible hardware management options. Each type of virtualization is a building block for partitioning the different resources that comprise an infrastructure, and delivering the resources to guests in a form that does not restrict their use. Slivers enable guests to receive and control entire virtual machines, virtual disks, and virtual network links without conflicting with competing guests. However, virtualization is not a panacea. Networked operating systems coordinate the binding of multiple instances of virtual hardware to multiple guests. Below we briefly outline important work in machine, network, and storage virtualization.

3.1.1 Machine Virtualization

Many different machine virtualization technologies exist that differ subtly in their implementation [22, 25, 54, 153, 168]. Some technologies provide true hardware virtualization [168], others provide paravirtualization [22, 176] which alters the virtual hardware for improved performance, others provide OS-level virtualization at the system call interface [54], and others simply strengthen and enhance existing operating system isolation mechansims [153]. Each of these systems offer a different set of hardware management services and slivering capabilities. We detail common hardware management services in Chapter 5. In our architecture, service managers programmatically control these capabilities for guests.

Improving the management capabilities listed above, as well as introducing new capabilities, is an active area of research. For instance, machines, whether virtual or physical, require root disk images that store persistent node operating system state. The popularity of machine virtualization has motivated the packaging of applications with their operating system in a virtual appliance [147, 148]. Virtual appliances bind an application with its node operating system and bypass operating system installation by executing both as a single unit on a virtual machine. Virtualization also enables new capabilities for configuration debugging [101, 175], intrusion detection [57], and flash cloning [166].

3.1.2 Storage Virtualization

Like machine virtualization, storage virtualization provides a software layer that divides a physical storage device into multiple virtual storage devices. Site administrators use different forms of storage virtualization to quickly create and export network-accessible disks and file systems. Site administrators may also use storage virtualization to mask the physical location of data or provide a consistent view of data spread across multiple independent physical storage devices.

Multiple technologies support the creation and management of virtual pools of storage. Sun's ZFS file system supports snapshotting, cloning, and exporting of entire file systems. Logical Volume Manager (LVM) operates at the block level, and allows the creation of logical disks and block-level snapshots. Filer appliances also offer programmatic snapshotting and cloning at the block-level or file system-level. Node operating systems may access pools of virtualized storage over a network using network storage protocols such as iSCSI (a block level protocol) or NFS (a file level protocol).

Research into storage virtualization for collections of virtual machines is ongoing. Parallax [173] is a block-level storage system designed for large collections of virtual machines that use a small set of distinct root disk images or virtual appliances. Parallax supports fast cloning of disk images and the migration of on-disk data in conjunction with virtual machine migration. Researchers are also investigating methods for loosening the coarse-grained partitioning of disks, which restricts data sharing and complicates security, ease of use, and management, by combining file-based sharing with cloning and snapshotting capabilities in a virtualization-aware file system [133].

Storage virtualization that supports snapshotting and cloning of file systems and disks along with the advent of virtual appliances provide the mechanisms necessary for a networked operating system

to create, alter, and destroy virtual storage devices and bind them to virtual machines on-demand.

3.1.3 Network Virtualization

The goal of network virtualization is similar to that of storage and machine virtualization: create and isolate shared network links. For example, Virtual LANs (VLANs) provide a network of machines the illusion of a dedicated LAN. Network virtualization also enables better testing and experimentation of network applications and protocols. For example, VINI seeks to grant developers the control to create their own private network on a shared physical infrastructure, which includes both edge hosts and internal network links [24].

Virtualization support in switches, routers, and edge hosts increases the possibilities for managing, isolating, and sharing internal network resources, such as bandwidth, in a more controlled fashion than today's "best-effort" Internet. For example, network virtualization technology can enable multiple independent network architectures, rather than the single, global Internet that exists today [52, 62]. Violin investigates linking virtual machines over a virtual network that spans the wide-area—the approach combines machine virtualization with network virtualization to give a guest the illusion of a dedicated, distributed network infrastructure [144]. As with other forms of virtualization, network virtualization provides a networked operating system mechanisms to sliver and allocate network resources that do not exist for physical hardware.

3.2 Middleware

The term middleware was first used in 1968 and the original definition remains a good description: middleware exists "... because no matter how good the manufacturer's software for items like file handling it is just not suitable; it's either inefficient or inappropriate [122]." Middleware now signifies a broad class of systems that layer software on top of the node operating system to create a custom execution environment with its own set of abstractions tailored to the needs of a specific class of applications.

Middleware approaches have long dominated the landscape for networked resource sharing because the only way to innovate before the advent of rich, programmatic interfaces to control physical and virtual hardware was to layer new abstractions on top of node operating systems. With the rise

of virtual hardware and network-controlled physical hardware the assumptions that initially drove networked resource sharing using the middleware approach have now changed. However, we do not need to throw away years spent developing sophisticated middleware that support popular programming models to adopt a new approach; instead, we only need to extend existing middleware to support visible hardware allocation and revocation mechanisms provided by a networked operating system, as discussed in Chapter 9. Below we discuss two important classes of middleware—batch schedulers and grids—that represent guests of a networked operating system that define a programming model.

3.2.1 Batch Job Scheduling

Batch job schedulers are middleware systems that schedule *tasks* on a set of compute nodes. Tasks are computations that execute for some time, produce output, and then terminate. Tasks do not include persistent services, such as multi-tier web applications, which do not fit the task mold since they never terminate. Individual tasks may be complex workflows that define the sequenced execution of multiple, independent sub-tasks, where each sub-task passes its output as input to the next sub-task. Parallel tasks execute on many machines at the same time. For example, MPI is a popular library for developing parallel tasks. Examples of batch schedulers include Sun's GridEngine [75] (SGE), Condor [113], Load Sharing Facility [183] (LSF), and PBS/Torque. Each of these batch schedulers follows a similar model: users submit tasks at a submit host, which then passes the task to a scheduler that decides when and where to execute tasks on one or more compute hosts.

Batch schedulers must deal directly with managing networked resources. However, batch schedulers do not share hardware resources between multiple guest environments—the resource they provide is actually *the service* of executing a task on a set of resources. The scheduling service defines an important programming model for networked resource sharing that redefines the capabilities of the hardware to support tasks, which a system may queue, schedule, and execute. We discuss the integration of the GridEngine batch scheduler with our architecture in Section 9.1.

Batch job schedulers provide a uniform programming model, but since they do not control node operating systems or hardware, they have limited control over resource management. Quality-of-

Service, reservations, and flexible site control are important for batch job schedulers, but they have been elusive in the practice. The problem is that batch job schedulers can only control when to submit tasks to queues or node operating systems, but they cannot predict or control what resources are allocated by the lower layer, unless the lower layer provides those hooks.

3.2.2 Grid Computing

Batch schedulers do not operate across multiple spheres of authority. In some cases, such as with Condor glide-in jobs [37] or LSF multi-cluster [183], the scheduler may allow users of external batch schedulers to submit tasks—these examples represent a shift toward a grid computing model. Batch schedulers, which assume complete control of all available resources, are not grids. Grids focus on standardizing protocols for distributed resource management. The Open Grid Forum (OGF), which was formed in 2006 by merging the Global Grid Forum (GGF) and the Enterprise Grid Alliance, is a community of grid users and developers that define standards for developing grids, such as the Open Grid Services Architecture [70].

The goals of a networked operating system and grids are common, but the approaches are different. We explore how to factor out resource multiplexing from a larger system that implements programming abstractions and environments for distributed computing. Here we focus only on grids built using the Globus toolkit [67], which implements OGF's grid computing standards. The same arguments generally apply to other grid computing systems.

Globus grids support familiar abstractions: they allow users to run their tasks and workflows on somebody else's node operating system. Globus is middleware that runs above node operating systems installed by resource owners and sacrifices control over node operating systems to enable some degree of heterogeneity. It derives from the "metacomputing" idea, which introduces standard protocols and API libraries to weld diverse resources and diverse node operating systems into a uniform execution platform. Globus also provides services to establish a common notion of identity, and a common distributed middleware for routing jobs and scheduling them on local resources.

In practice, grids decentralize resource management across multiple spheres of authority by extending the basic programming model of batch job schedulers. The choice of a batch job scheduling model is not central to grids or their implementation. The focus derives from the needs of users

of computationally intensive workloads, which package their computations as tasks. In contrast to grids, the goal of this thesis is to separate specific programming models from resource management. Grids, like batch schedulers, represent an important programming model that an operating system architecture for networked resource management must support.

3.3 Data Center Operating Systems

Previous work advocates controlling of collections of machines using a *data center operating system* that coordinates their allocation for software. Muse represents an early example of this approach. Muse combines the coordinated allocation of resources to guests with a policy for optimizing revenue and energy consumption in a hosting center [38]. The original conception of Cluster-on-Demand derived from the same motivation as Muse, but focused on mechanisms to drive the allocation and reallocation of physical machines [120]. SHARC is a related system that adopts the approach by advocating coordinated sliver allocation for collections of machines [163]. Muse and SHARC preceded a resurgence in slivering technologies for virtual machines. As a result, both systems use resource containers in node operating systems to sliver and isolate machine resources. The use of resource containers does not provide the flexible hardware management services of virtual machines.

The implementation of data center operating systems differ from distributed operating systems, which redefined each machine's kernel, in their use of preexisting mechanisms for coarse-grained slivering using resource containers [20] in node operating systems rather than redefining low-level kernel mechanisms. Systems for virtual and physical machine management, described below, are types of data center operating systems that use the same approach to coordinate the allocation of virtual and physical machines. The approach is less intrusive than distributed operating systems because it does not require redefining existing middleware or node operating system kernels. Existing data center operating systems as well as recent systems for virtual and physical machine management differ from our approach because they combine specific policies and programming abstractions with specific mechanisms for multiplexing one or more types of resources.

Virtual Machine Management

Systems for managing networked resources in data centers leverage virtual machine technology to take advantage of virtualization's increased management flexibility. Industry products include VMware Virtual Center and Xen Enterprise. The state-of-the-art products provide management consoles that allow site administrators, not guests, to manually control virtual hardware resources. These systems, which offer programmatic interfaces to collections of hardware, are building blocks for a networked operating system.

Other systems that perform virtual machine management include SoftUDC [97], Sandpiper [181], Virtual Computing Labratory [17], Usher [116], SODA [92], In Vigo [5], Virtual Workspaces [99], Condor [146], and PlanetLab [23]. We outline the key aspects of these systems below.

While the systems differ in implementation and their target applications, each system's goal has the same basic element: automating a set of privileged actions, typically performed by a site administrator, so users do not have to deal with the complexity of instantiating and placing virtual machines in a data center. Our architecture differs from these systems by extracting specific resource management policies and usage scenarios (*e.g.*, programming abstractions) from the system and defining a common set of elements for networked resource management based on the neutrality principle. We describe the main points of each system below.

SoftUDC aims to provide a management console for administrators to control and manage an infrastructure using virtualization technology. However, SoftUDC focuses primarily on how site administrators use the console to manage an infrastructure and does not specifically address how guests use the virtualized environment. Sandpiper focuses on specific policies for migrating virtual machines to improve guest performance, and not a policy-free operating system architecture for networked resource management. Our architecture is sufficient to export the mechanisms for service managers to implement Sandpiper's migration policies.

Virtual Computing Labratory (VCL) is a web portal where users may request a virtual machine that VCL configures on-demand. Usher is similar to VCL and provides a command line console for allocating virtual machines to users of a cluster. Neither VCL nor Usher address integration with guests, resource adaptation policies, or support for infrastructures that span multiple sites. In Vigo, Virtual Workspaces, and Condor concentrate on virtualizing resources for Grid computing

environments that support a task execution programming model. As we state above, task execution represents one, of many, programming models that a networked operating system must support. Our architecture supports these programming abstractions, as discussed in Section 9.1. SODA recognizes the importance of programming models other than task execution, and instantiates User-mode Linux [54] virtual machines for persistent services. In contrast, our architecture generalizes to any programming model by supporting hardware for any type of guest.

Physical Machine Management

A networked operating system should also be able to provide mechanisms for manipulating the physical hardware by exporting hardware management services of physical machines. Physical hardware management focuses on controlling physical, not virtual, hardware such as raw machines, disks, switches, and routers. For example, the Dynamic Host Configuration Protocol (DHCP) can take control of a physical machine which boots into Intel's Preboot eXecution Environment (PXE) to transfer and invoke a custom kernel provided by the remote DHCP server. Once booted, the custom kernel may rewrite a machine's local disks or configure a machine to boot using a remote disk via NFS or iSCSI.

Mechanisms also exist to boot machines remotely using programmable power strips to drive the physical machine reallocation process [15, 39, 177], use Wake-on-LAN to programmatically power machines up and down [38], or dynamically regulate CPU frequency to meet infrastructure-wide energy targets [139]. Blade enclosures now provide network-accessible backplanes that obviate the need for PXE, programmable power switches, or Wake-on-LAN to manage physical machines using remote network protocols. Site administrators may also dynamically modify common management services to control each machine's view of its environment. These services include DNS for resolving host names, NSS for determining file-access permissions, and PAM for user authentication.

Systems that manage physical infrastructure include Oceano [15] for SLA-compliant management of web applications, Rocks [126] for quick installation of customized Linux clusters, and Emulab [177] for configuring network emulation experiments. Emulab also supports instantiating virtual machines for network experiments [142, 58]. Oceano, Rocks, and Emulab each implement important programming models for web applications, machine imaging, and network experimentation, respectively. Physical machine management and these programming models are realizable within

our architecture. We discuss the integration of physical machine management with our architecture in Chapter 8.

3.3.1 PlanetLab

PlanetLab virtualizes the resources of over 700 machines from 379 distinct clusters located on 6 continents by acting as a brokering service that establishes relationships with sites spread across the world. To join PlanetLab, sites permanently relinquish control of at least two machines by registering them with PlanetLab Central and installing a PlanetLab-specific OS and disk image. PlanetLab users may then access these machines, subject to PlanetLab allocation policies, to test and deploy new network applications on a global scale. PlanetLab defines a programming model around a distributed virtualization abstraction, called a slice: guests receive a virtual machine presence on one or more machines in the federation. However, slices provide no assurance over the underlying resources bound to each virtual machine and no notifications when a slice's resource allotment changes due to contention or failure.

In contrast, we advocate that sites control when, how much, and how long resources are made available to federations like PlanetLab using a lease abstraction, and provide the mechanisms to programmatically repurpose hardware for new uses by supporting guest control of the software stack. Our architecture encourages further participation in PlanetLab by enabling infrastructures to dynamically donate and withdraw their machines as local conditions dictate, reflecting a view that sustainable federation will only occur when systems do not force sites to permanently relinquish their machines and do not bind machines to a specific programming model. In Chapter 9, we demonstrate PlanetLab as one example of a guest running over our architecture. We also discuss an integration of our architecture with Plush [9]: software originally developed for PlanetLab that implements a programming model for distributed application management without multiplexing resources. We describe how we generalize common programming abstractions for distributed application management and apply them to resource management in a networked operating system. We discuss PlanetLab's programming model in detail and how it integrates with our architecture's policy-neutral principle in Chapter 9.

3.3.2 Utility Computing

Utility computing (or cloud computing) is a variant of networked resource management that focuses on selling computing as a service: in this model, customers rent resources as they need them. Amazon's Elastic Compute Cloud (EC2) is an example of an operational computing utility [74]. EC2 customers programmatically create Xen virtual machines as they need them without purchasing and maintaining their own dedicated infrastructure. Amazon's Simple Storage Service (S3) allows customers to store data, such as virtual machine disk images, on Amazon's infrastructure. Amazon meters the I/O bandwidth to and from S3, as well as the number of virtual machine hours clocked on EC2, and charges customers monthly based on their usage.

The XenoServer Open Platform [85, 103] is a proposal for a global utility computing framework based on Xen virtual machines. The platform proposes multiple utilities, such as EC2, as well as resource discovery and payment systems to enable coordinated locating and purchasing of virtualized resources. Our architecture may serve as a foundation for building and enhancing utility computing platforms. In Chapter 8 we explore the benefits of the integrating our policy-neutral architecture with EC2 to enable the leasing of EC2 resources.

3.4 Elements of Distributed Systems

A system for sharing networked resources is inherently distributed. Authorities, service managers, and brokering services, along with the resources under their collective control, execute at different points in a network under different spheres of authority. As a result, networked operating system design encounters common problems from distributed systems design. Specifically, our architecture applies and extends research into distributed resource arbitration and reliability. We review related work in these areas below.

3.4.1 Resource Scheduling and Arbitration

A centralized scheme for resource scheduling and arbitration in a distributed infrastructure composed of self-interested guests and sites requires sites to relinquish control of their resources. It is unlikely that sites will relinquish this control *en masse* unless proper incentives exist to outweigh the impact allocation decisions have on each site's operational costs [38, 119, 139]. Markets represent a natural

approach to making decentralized resource scheduling and arbitration decisions among self-interested actors. Markets encourage sites to contribute resources to infrastructures by providing an incentive, such as money or the promise of future resource usage, to contribute, as well as a basis for sites to make allocation decisions to guests under constraint.

The systems community has flirted with market-based allocation regularly over the years [34, 154, 156, 169, 180]. Unfortunately, market design is complex and places a heavy burden on the user to manage currency, define utility and bidding functions, and dynamically adjust resource usage in accordance with prices. Mapping currency to utility is a heuristic process that requires users to continuously monitor resource prices. Sites must deal with the complexity of defining expressive bidding languages and auction protocols for users. Furthermore, optimally clearing all but the simplest auctions requires solving NP-hard problems. Shneidman et al. present a summary of the challenges markets must address to become a viable resource allocation mechanism [152].

One problem Shneidman et al. discuss is a lack of a viable virtual currency system for computational markets. Chapter 6 presents a virtual currency system which associates currency with a lease term and recycles currency back to the user at the end of the term. Importantly, the virtual currency system does not define a resource management policy. Users lease currency in exchange for resource leases. Adjusting the currency's lease term provides a configurable tradeoff between proportional-share scheduling and a market economy independent of the auction. The currency serves as a foundation for accepted forms of allocation, such as proportional-share, as well as evolving, experimental forms of market-based allocation.

Proposals such as Karma [165], SWIFT [159], and CompuP2P [83] use currency to address the free-rider problem in a peer-to-peer setting where anonymous users enter and leave the system at a high rate. The design of currency in peer-to-peer systems reflects the model that trust is only by reputation. Industry initiatives in utility computing are based on cash and rely on recycling of currency within the larger common market [178].

There have also been recent proposals for community virtual currency. Tycoon is a market-based system for managing networked resources [105]. Tycoon uses a price anticipating auction protocol in which all bidders for a resource receive a share that varies with contention [64]. They prove that the protocol always yields a Nash equilibrium when auctions are strongly competitive, competition

is stable, and bidders adapt to recent history according to stable utility functions. However, in Tycoon, the result of each auction does not assure the buyer a predictable resource share. Resource assignments in Tycoon vary continuously with contention, emphasizing agility over stability and assurance. In our architecture, leases (discussed in Section 6.2) provide an opportunity to balance the stability of knowing your resource assignment and your currency holdings, with the agility of being able to alter your holdings over time, as discussed in Chapter 6.

3.4.2 Leases and Fault Tolerance

A network partition may interrupt communication between authorities, service managers, brokering services, guests, and hardware at any time. Servers may also crash or fail at any time. Like other distributed systems, a networked operating system must address network and server failures. Leases are a useful tool for building fault-tolerant distributed systems that are resilient to failures [107].

Variants of leases are 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. Web Services Resource Framework (WSRF) uses leases as a tool for distributed garbage collection of state [68]. The technique of robust distributed reference counting with expiration times also appears in Network Objects [28], and subsequent systems—including Java RMI [179], Jini [167], and Microsoft .NET—have adopted it with the lease vocabulary. While our architecture benefits from the lifetime management property of leases, we also use leases for resource management.

Mutual exclusion. Leases are useful as a basis for distributed mutual exclusion, most notably in cache consistency protocols [78, 114]. 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. Fault-tolerant distributed systems combine this

use of leases with replicated state machines [149] and quoroms or a primary-backup structure [31] to provide data availability in the face of server failures.

Resource management. Our architecture, like Sharp [73], uses the lease as a foundation for managing infrastructure resources. In Sharp, the use of leases 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, as described below.

- 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. In contrast, it is common with lease-based mutual exclusion for the server to issue a *callback* to evict the lease holder if another client requests the resource.
- 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.

3.5 Summary

This chapter ties together resource management at all layers of the software stack using a new layer that coordinates the mapping of software onto networked hardware. We review systems for networked resource management that manage multiple software stacks and describe how they combine resource management policies and programming model with resource multiplexing. Finally, we show how our architecture relates to classical issues in distributed systems research including resource arbitration and the use of leases for reliability.

Chapter 4

A Networked Operating System Architecture

"Perfection is reached not when there is no longer anything to add,

but when there is no longer anything to take away."

Antoine de Saint-Exupéry

This chapter outlines the key elements of a networked operating system architecture based on a lease abstraction. We first review the relevant aspects of the SHARP resource peering framework [73], which was also based on a lease abstraction. We then outline extensions to the lease abstraction in SHARP including: guest/resource independence, visible resource allocation, modification, and revocation, and an abort protocol. Finally, we illustrate how the lease abstraction can apply to multiplex machine resource types used in Cluster-on-Demand and other systems [39].

4.1 Background

Our leasing architecture derives from the SHARP framework for secure resource peering and distributed resource allocation. The participants in the leasing protocols are long-lived software servers, called *actors*, that interact over a network to manage resources. The term actor is meant to imply that the software servers are independent and self-interested. Actors represent different trust domains and spheres of authority that motivate a focus on negotiation and well-formed contracts. We discuss the impact of actor self-interest in Chapter 6 in relation to market-based resource allocation.

- Each guest has an associated *service manager* actor that monitors the demands and resource status of one or more guests, and negotiates to acquire leases for the mix of resources needed to host the guests. Each service manager requests and maintains leases on behalf of one or more guests, driven by its own knowledge of guest-specific behavior and requirements.

- An *authority* actor controls allocation for an aggregation of resources controlled by a single sphere of authority, and is responsible for enforcing isolation among multiple guests hosted on the resources under its control.

- A *broker* actor maintains inventories of resources offered by authorities, and matches service manager requests to resources from its inventory. 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. The original SHARP framework refers to brokers as agents: we only use the term agent to describe a broker that operates under the sphere of authority of one or more sites and/or guests. In this case, we say that the broker is an agent acting on behalf of the corresponding sites and/or guests. We also refer to brokers as *brokering services* to emphasize that they are software servers, and that they may cooperate and replicate to support more robust brokering.

SHARP, which was initially prototyped for PlanetLab [23], defines a secure protocol for authorities to lease resource rights over time to brokering services and service managers. We inherit the slice terminology from the original SHARP work and note that the same term also describes the distributed virtualization abstraction that exists in PlanetLab [23]. To avoid confusion, for our purposes, a slice is simply a collection of resources leased by a single guest at any time, and is a generalization of the distributed virtualization abstraction defined in [23]. Note that a service manager may control one or more guests and, hence, may also control one or more slices.

4.1.1 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 associated 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. 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 architecture represents attributes associated with each lease as properties,

as discussed in Section 4.3.1. The intent is that the resource attributes quantify capability in a guest-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 [164]. The level of underlying isolation provided by a resource (especially a virtualized one) is not fundamental to our architecture, as long as the service manager and the authority mutually agree on the nature of the resource. For example, both parties may agree to a lease for a group of virtual machines that span sites where each virtual machine receives a continuously varying share of the processor time, memory allotment, and network link of a physical machine in accordance with competition from co-located virtual machines, as with PlanetLab [23] and VServers [153].

4.1.2 Sharp 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. A distinguishing feature of Sharp brokers is that they have power to coordinate resource allocation. 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 authorities control the *assignment* of specific resource units at the site to satisfy requests approved by the brokers. The decoupling present in Sharp balances global coordination (in the brokers) with local autonomy (in the authorities).

Figure 4.1 depicts the Sharp 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 an authority. The ticket specifies the resource type and the number of units granted, and the interval over which the ticket is valid (the term). Authorities 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

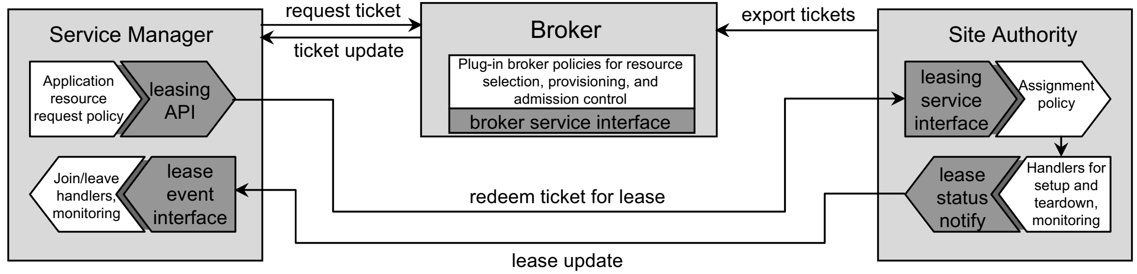

Figure 4.1: Summary of protocol interactions and extension points for the leasing system. A guest-specific service manager uses the lease API to request resources from a SHARP broker. The broker issues a ticket for a resource type, quantity, and site location that matches the request. The service manager requests a lease from the owning 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 guest. Plug-in modules for *setup* and *join* event handlers are applicable to different resource types and guests, respectively.

authority. This chapter focuses on the design principles for a policy-neutral networked operating system. Since our architecture conforms to SHARP, the delegation and security model applies directly. The mechanisms for accountable resource contracts are described in more detail in [73]. The delegation model is suitable to support broker networks of any topology, ranging from a network of per-site brokers to a deep hierarchy of communicating brokers.

We highlight two consequences of SHARP's brokering model. First, defining brokers that coordinate resource allocation by leasing resource rights is compatible with the neutrality principle: the resource type may specify any resource including physical and virtual hardware. Second, since brokers use leases to allocate the future right to a resource they are a suitable foundation for designing advanced resource scheduling and arbitration policies, such as advanced reservations. Chapter 6 explores a currency system that sites and brokering services may leverage to implement different policies that provides incentives for actors to contribute resources.

4.2 Shirako and Cluster-on-Demand

Shirako is a prototype implementation of our leasing architecture. Shirako includes a server to represent each actor that implements interfaces for extension modules to configure different guests and resources. The core of each actor is an actor-specific library that governs the behavior of resource leases. Site developers use the authority library to multiplex different resources underneath the lease abstraction on behalf of service managers. Likewise, guest developers use the service manager

library to instantiate different software environments using the lease abstraction to configure leased resources on demand. The lease abstraction also encapsulates inter-actor lease protocol interactions by defining the methods for ticket requests, redeems, and renewals.

Brokers have a similar library for developing arbitration policies that is independent of specific guests or resources and does not require hooks for integrating specific guest and resource technologies. At the end of this chapter we illustrate how a Cluster-on-Demand authority integrates with Shirako to manage physical and virtual machines. In the next section we use Cluster-on-Demand to illustrate the design principles—guest/resource neutrality, visible allocation and revocation, and an abort protocol—and how they apply to managing networked resources. In Chapter 5, we extend the approach with additional design principles—exposing names and secure bindings—that include sliver allocation. In Chapter 8, we give examples of other resources we integrate with the authority library and in Chapter 9 we give examples of guests we integrate with the service manager library.

The lease abstraction manages state storage and recovery for each actor, and mediates actor protocol interactions. The lease abstraction defines primitives, not policy, for developing guest-specific service managers. In Chapter 9, we present several specific guests and the policies they use to monitor their load, adapt to changing conditions, and handle software-related failures. Each actor invokes lease interfaces to initiate lease-related actions at the time of its choosing. In addition, actor implementations associate extension modules with each lease, which the lease abstraction invokes in response to specific events associated with state changes. In Chapter 7, we describe a distributed lease state machine which governs the state transitions that raise specific lease events. For example, an event may include transferring a resource in or out of a *slice*—a logical grouping of resources held by a guest.

Figure 4.1 summarizes the separation of the lease abstraction on each actor from different actor management policies. Each actor has a *mapper* policy module that it invokes 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

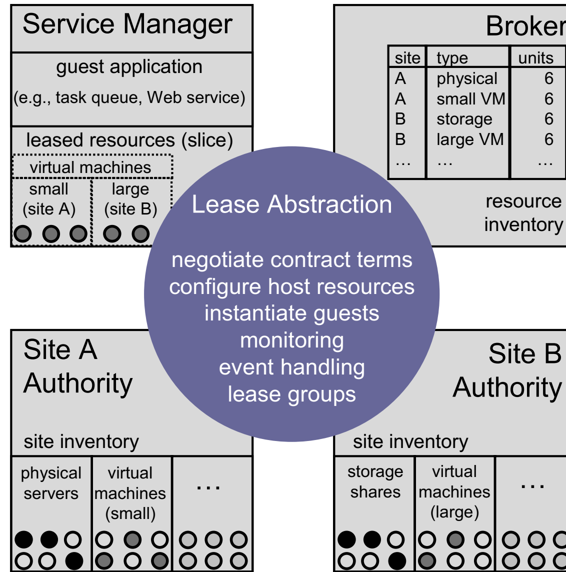

Figure 4.2: An example scenario that illustrates Sharp and COD using a lease abstraction. The scenario depicts a guest acquiring machines from two sites through a broker. Each site maintains an authority server that controls its physical machines, and registers inventories of offered machines with the broker. A service manager interacts with a broker to lease machines from two different authorities on behalf of its guest. The lease abstraction is resource-independent, guest-independent, and policy-neutral, and masks all protocol interactions and lease state maintenance from the service manager controlling its guest, the authority controlling its resources, and the broker mediating between the two.

units from its inventory to fill lease requests that are backed by a valid ticket from an approved broker.

Figure 4.2 depicts a simple example of the benefit of combining COD and Sharp using Shirako and the lease abstraction: a service manager leases two types of virtual machines from two different sites using a broker and instantiates its guest on them. Following the neutrality principle, the lease abstraction exposes only elemental resource sharing primitives including contract negotiation, site resource configuration, guest instantiation, event handling, guest monitoring, and lease grouping to the service manager and site. In the next section we outline the design principles for lease abstraction and how the architecture embodies each principle.

4.3 Design Principles

We apply the neutrality principle by separating the lease abstraction from any dependencies on specific policies, guests, and resources. We combine three different design principles to accomplish this separation.

- **Guest/Resource Neutrality**. The lease abstraction is sufficiently general to apply to other resources, such as bandwidth-provisioned network paths, network storage objects, or sensors. As a result, we define a narrow interface that divides the lease abstraction from the guests and resources that use it. Actors also exchange type-specific configuration actions for each lease to control the setup or teardown of resources for use by the guest, and persist lease state as a foundation for fault tolerance. To provide guest/resource independence, the lease abstraction stores guest-specific and resource-specific state using opaque property lists.

- **Visible Allocation, Modification, and Revocation**. Guests must have an opportunity to know about changes to their leased resource holdings over time to adapt to changing conditions. For example, a guest may need to respond to load surges or resource failures by leasing additional resources, or it may need to adjust to contention for shared resources by deferring work or gracefully reducing service quality. As a result, the lease abstraction exposes resource allocation, modification, and revocation to the service manager and authority.

- **An Abort Protocol**. Leases define a natural termination point for service managers and sites to coordinate vacation of leased resources. After a lease expires sites are free to unilaterally teardown leased resources.

4.3.1 Guest/Resource Neutrality

The architecture decouples the lease abstraction from specific guests and resources using two techniques: a handler interface that hides guest and resource configuration from the lease abstraction, and a technique for exchanging context-specific configuration information using opaque property lists. We describe both techniques below.

Handler Interface

The lease abstraction integrates with different guests and resources using a narrow *handler* interface. Service managers and authorities register handler modules with each lease that define guest-specific and resource-specific configuration actions invoked in response to lease events. Although the interfaces at the service manager and authority are identical, to avoid confusion we refer to the handler

Function	*Description*	*Example*
join	Executed by a service manager to 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	Executed by a service manager to modify an existing guest resource.	Modify for a guest batch scheduler notifies the scheduler that a virtual machine's sliver size has changed.
leave	Executed by a service manager to remove a resource from a guest.	Teardown for a guest batch scheduler issues directives to the batch scheduler's master to remove a compute machine from the available pool.

Table 4.1: The lease abstraction for the service manager implements the guest handler interface. Service manager developers may add or modify guest handlers to support new types of guests.

Function	*Description*	*Example*
setup	Executed by an authority to setup a new resource for a site.	Setup for a virtual machine includes creating a new root disk image and initiating the boot process.
modify	Executed by an authority to modify an existing site resource.	Modify for a virtual machine may migrate the virtual machine to a new physical host and/or change the isolated sliver of host CPU, memory, and bandwidth.
teardown	Executed by an authority to destroy a site resource.	Teardown for a virtual machine includes deleting a root disk image and destroying a virtual machine.

Table 4.2: The lease abstraction for the site authority implements the resource handler interface. Site administrators add new resource handlers or update existing resource handlers to lease different types of resources.

used by service managers as the *guest handler* and the handler used by authorities as the *resource handler*.

Each guest and resource handler includes three entry points that drive configuration and membership transitions in the guest as resource units transfer in or out of a lease. A description of each entry point for the guest handler is shown in Table 4.1 and the description of each entry point for the resource handler is shown in Table 4.2. We summarize the execution of the handlers and the sequence in which they execute below.

- The authority invokes a *setup* action to prime and configure each new resource unit assigned to a lease by the mapper. The authority notifies the guest and issues the lease when all of the setup actions for a lease complete.

- 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 *join* guest event handlers (and the *leave* guest event handler) may interact with other guest software components 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

across a distributed application.

- The authority or service manager may invoke a *modify* action if the nature of a resource changes on a lease extension. We discuss lease extensions further in the Section 4.3.2.
- 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.
- At the end of the lease an authority issues a *teardown* action for each resource unit to make the resources available for reallocation.

Using Properties

To maintain a clean decoupling between the lease abstraction and specific resources and guests the we store all guest-specific and resource-specific information in opaque *property lists*, which are sets of *(key, value)* string pairs. Below we discuss the role of properties for context-specific information exchange and state storage.

First, actors exchange context-specific information to guide allocation and configuration actions in the guest and resource event handlers. For example, a guest may need to pass specific requirements for configuring a resource to a site, which the service manager must communicate to the authority when redeeming a ticket for a lease. Similarly, an authority must pass information about each resource unit to a service manager in order for it to incorporate each resource into the guest. Property lists attached to message exchanges are opaque to the lease abstraction on the peer. Their meaning is a convention among the policies and handlers at the service manager, broker, and authority. Property sets flow from one actor to another, through the policies on each of the steps and protocol exchanges depicted in Figure 4.1, and down to the corresponding guest and resource handler.

- *Resource properties* attached to tickets give the attributes of the assigned resource types. Brokers attach resource properties to each ticket sent first to the service manager and subsequently to the authority. The lease abstraction at the authority passes these properties directly to the resource handler. Note that the authority assignment policy may append to the resource properties before sending them to the resource handler to guide resource configuration.

- *Configuration properties* attached to redeem requests direct how an authority configures resources. Service managers attach configuration properties to each ticket sent to the authority. The lease abstraction at the authority passes the configuration properties directly to the resource handler.

- *Unit properties* attached to each lease define additional attributes for each resource unit assigned. Authorities attach unit properties to each resource unit in lease notifications sent to the service manager. The lease abstraction at the service manager passes these unit properties directly to the guest handler.

The lease abstraction on each actor also serializes its state to a *local property list* and commits it to a persistent store, such as a database, before passing the properties to the resource or guest event handler. Service managers may use local properties to attach and persist arbitrary data associated with a guest or lease. In Chapter 7, we describe a methodology for tolerating actor server failures where lease state transitions trigger local commits of lease state to a database.

The lease abstraction includes an interface with functions to `save` state to the local property list and `reset` state from the local property list as a foundation for our methodology for tolerating failures. Other first-class data structures in the architecture persist state using the same basic property list approach including slices as well as the mapper modules that implement different adaptation, arbitration, and assignment policies.

4.3.2 Visible Allocation, Modification, and Revocation

In our architecture, authorities and service managers specify the nature and length of resource use in a lease contract and authorities programmatically notify service managers when the resources specified in the contract change. Service managers leverage visible allocation, modification, and revocation of resources to inspect the state of guests, determine how to handle a change in resource allotment, and alter the guest's configuration to incorporate the addition, modification, or loss of resources. For example, a service manager controlling a network service supporting its own clients may wish to gracefully degrade all clients' service quality, provide differentiated quality of service to preferred clients, or seek and request available resources from elsewhere in the networked infrastructure to satisfy demand.

In our prototype, changes in a lease take effect only at the end of the previously agreed term. We note that restricting changes to lease boundaries is a subtle policy choice. Examining the policy implications of a lease abstraction that allows for renegotiation or revocation before lease expiration is outside the scope of this thesis. The lease abstraction must support visible allocation, modification, and revocation regardless of the strength of the lease contract. We leverage the principle of visible resource allocation, modification, and revocation for guests in four distinct ways: lease flexing and modifying, visible handler invocation, guest lease handlers, and lease groups.

Flexing and Modifying

First, peers may incorporate changes to the lease at renewal time by *modifying* each unit's underlying characteristics or *flexing* the number of resource units. Supporting resource modification on lease renewals gives guests the opportunity to adjust to changes in an individual resource's capabilities or attributes. For example, if we treat a machine as a resource then hotplugging a new IDE hard disk constitutes a fundamental change in the machine's nature.

Resource flexing has the benefit of not forcing a service manager to vacate an entire lease and replace it with a smaller one, interrupting continuity, if a lease's units must be shrunk on a lease extension. Shrinking leases on extensions requires using properties to pass context-specific information using the lease abstraction, since a shrinking extend requires a general method for service managers to transfer the name of victim units to relinquish to the authority. Section 5.3.4 discusses this use of properties for generalized victim selection.

The protocol to extend a lease involves the same pattern of exchanges as to initiate a new lease (see Figure 4.1). The service manager must obtain a new ticket from the broker by marking a request as extending an existing ticket named by a unique ID. Lease renewals maintain the continuity of resource assignments when both parties agree to extend the original contract. A lease extension makes explicit that the next holder of a resource is the same as the current holder, bypassing the usual *teardown/setup* sequence at the authority on lease 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.

Function	*Description*	*Example*
onExtendTicket	Executed by a service manager immediately 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	Executed by a service manager once all resources in a lease are active.	A guest may issue a directive to begin a parallel computation, as in the case of an parallel MPI task.
onCloseLease	Executed by a service manager before the guest leave handler's execute.	Download and store data from one or more leased machines.

Table 4.3: The lease abstraction for the service manager implements the guest lease handler interface to give service managers an opportunity to modify leases when their state changes.

Visible Guest/Resource Handler Invocation

Second, the *guest handler* described above, represents a form of visible allocation, modification, and revocation for guests since the lease abstraction at the service manager invokes each guest event handler after the corresponding asynchronous notification from the authority. The principle of visible allocation, modification, and revocation also extends to the lease abstraction on the authority. The service manager notifies the authority to redeem a new lease, extend an existing lease, or close an expired lease. In response to a service manager's notifications the authority executes each lease's corresponding resource event handler to setup, modify, or teardown resources. In particular, a site may use the modify event handler on lease extensions to adjust the isolated resource slivers bound to a leased virtual machine (*i.e.*, change the nature of the resource). In Chapter 5, we show how authorities may augment this use of the modify event handler by allocating slivers across multiple leases.

Guest Lease Handlers

Third, 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 implementation of the lease abstraction within the service manager library upcalls a guest lease handler function before a lease state change occurs. However, as opposed to the guest handler, the service manager invokes the functions on a per-lease, and not per-resource unit, basis. 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 4.3 lists examples of how 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).

Lease Groups

Fourth, we augment the lease abstraction on the service manager with a grouping primitive that defines a sequence for guest handler invocations. Since the service manager specifies properties on a per-lease basis, it is useful to obtain separate leases to support development of guests that require a diversity of resources and configurations. Configuration dependencies among leases may impose a partial order on configuration actions—either within the 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.

We extend the lease abstraction to enable service managers to enforce a specified configuration sequencing for lease groups. The lease abstraction 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 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 lease abstraction upcalls a service manager-defined method before transitioning out of the blocked state; the upcall gives the service manager an opportunity to manipulate properties on the lease before it fires, or to impose more complex trigger conditions.

For example, recent work observes that loose synchronization primitives define a useful trigger condition for sequencing guest instantiation on resources that span a volatile wide-area network [10]. Lease groups are a generalization of task workflows that sequence the execution and data flow of multiple tasks for batch schedulers and grids [184] as well as other workflow techniques for instantiating distributed guests on an existing set of resources [9]. In contrast to task workflows,

lease groups define sequenced execution of distributed guests, which may include tasks, across a set of allocated resources.

4.3.3 An Abort Protocol

An authority may unilaterally destroy resources at the end of a lease to make them available for reallocation if it does not receive a notification from a service manager, although we note that the decision of when to destroy resources without a service manager notification is a policy choice that is outside the scope of this thesis. An authority is free to implement a restrictive policy that destroys resources immediately at the end of a lease, a relaxed policy that destroys resources only when it must reallocate them, or a policy that destroys resources at any time in between. Unilateral resource teardown by an authority is reminiscent of Exokernel's abort protocol.

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 in Exokernel and others [13, 61]. As long as actors loosely synchronize their clocks, a service manager need only invoke its *leave* handlers for each lease immediately prior to termination time. As described in Chapter 7, service managers notify authorities when they vacate leased resources which may then issue their resource teardown handlers.

4.4 Example: Allocating Virtual Clusters to Guests

In this section we describe how an authority uses the lease abstraction to share resources in the form of machines. The authority is inspired by Cluster-on-Demand (COD) which was originally intended to allocate and configure *virtual clusters* from a shared collection of servers [39]. Each virtual cluster comprises a dynamic set of machines and associated hardware resources assigned to each guest at a site. The original COD prototype allocated physical machines using 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 coordinate resource usage across multiple sites. Extensive experience using the COD prototype led to the SHARP framework, and subsequently to the extensible leasing architecture we outline in this thesis.

4.4.1 Integrating Machine Resources

A COD authority defines resource handlers to configure different types of machines. The resource handlers use machine-specific configuration and unit properties to drive virtual cluster configuration at the site, as well as guest deployment. The authority selects the resource handler to execute for each lease redeemed by a service manager based on each lease's resource type and attached configuration properties. The *setup* and *teardown* event handlers execute within the site's trusted computing base (TCB). The original COD prototype was initially designed to control physical machines with database-driven network booting (PXE/DHCP). The physical booting machinery is familiar from Emulab [177], Rocks [126], and recent commercial systems. The resource handler controls boot images and options by generating configuration files served via TFTP to standard bootloaders (*e.g.*, grub).

A COD authority drives 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 machines have read access to an LDAP directory describing the containing virtual cluster. Guest machines configured to run Linux may use an LDAP-enabled version of AutoFS to mount NFS file systems, a PAM module that retrieves user logins from LDAP, and an NSS module that controls file-access permissions.

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

In addition to the resource handlers, COD includes classes to manage IP and DNS name spaces at the slice level. The authority names each instantiated machine with an ID that is unique within the slice. It derives machine 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 machine is assigned to it. Public address space is limited; the system may treat it as a managed resource using

the lease abstraction, as described in the next chapter. In our deployment the service managers run on a control subnet with routes to and from the private IP subnets. COD service managers overload configuration properties to specify victim virtual machines by IP address on a shrinking lease extension, giving guests the power to choose which machine an authority revokes in a lease block.

4.4.2 Support for Virtual Machines

As a test of the architecture, we extend COD to manage virtual machines using the Xen hypervisor [22]. The extensions consist primarily of a modified resource handler and extensions to the authority-side mapper policy module to assign virtual machine images to physical machines. The new virtual machine resource handler 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 machine, 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; binding pre-defined shares of CPU, memory, and bandwidth to a virtual machine (see Section 5.3.1); and initiating the boot. The authority writes some configuration-specific data onto the image, including a user-supplied public key and host private key, and an LDAP path reference for the containing virtual cluster. The *setup* event handler may also assign shares of a machine's resources to a virtual machine to provide an assurance of performance isolation, guided by resource properties attached to each ticket by the broker. Furthermore, a COD authority may use the modify resource event handler to alter the assigned sliver on lease extensions. We discuss sliver allocation in detail in the next chapter.

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 4.4: Selected properties used by Cluster-on-Demand, and sample values.

4.4.3 Defining Machine Properties

Configuration actions are driven by the configuration properties passed to the resource handler. Table 4.4 lists some of the important machine properties. These property names and legal values are conventions among the guest and resource handlers for COD-compatible service managers and authorities. Service managers may employ various shortcuts 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. For extend/renewal operations, servers may reuse cached configuration properties, bypassing the cost to retransmit them or process them at the receiver. 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. 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.

- *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 to enable secure, programmatic access to the leased nodes using SSL/SSH. The service manager generates a key-pair and passes the public key as a configuration property. The site's *setup* event handler writes the public key and a locally generated host private key onto the machine image, and returns each host public key as a unit property.

After the authority configures all machines within a lease it notifies the service manager, which then executes the selected *join* event handler for each resource unit within the lease, passing the resource unit's properties to the handler. The *join* and *leave* event 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. For example, in the case of a COD authority, the TCB does not include root access to the machine (which the guest controls), but does include assignment of a machine IP address since a COD authority does not virtualize the network.

In the case of COD, the unit properties returned for each machine 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 any available server executing on the machine (*e.g.*, `sshd` and `ssh`) to install and execute arbitrary guest software. Note that any server, such as `sshd`, executing on the machine that the service manager connects to must be pre-installed on the image, pre-configured for secure machine access, and started at boot time by the COD authority.

After the *join* event handler completes the service manager transitions the machine to an active state. Once a machine is active further management of the guest (*e.g.*, monitoring, adaptation, failure handling) depends on the implementation of each service manager.

4.5 Summary

This chapter presents a networked operating system architecture derived from the SHARP framework for secure resource peering. We present the lease abstraction and its design principles:

guest/resource independence, visible allocation, modification, and revocation, and an abort protocol. We then describe the use of the architecture and the lease abstraction to build and enhance Cluster-on-Demand, a reconfigurable data center which represents one example of a class of existing networked machine management systems.

Chapter 5

Exposing Names and Secure Bindings

"Intelligence is not to make no mistakes,

but quickly to see how to make them good."

Bertolt Brecht

Our architecture defines a single programming abstraction—the lease—for managing multiple types of guests and resources, and exhibits three design principles necessary to be independent of resource management policy: guest/resource independence, visible resource allocation, modification, and revocation, and an abort protocol. The architecture leverages the SHARP framework to support brokering and secure delegation of resource rights to third parties. However, we show in this chapter that the original conception of SHARP is insufficient for allocating virtualized hardware. We examine extensions to our architecture and SHARP based on the principles outlined in Section 2.2 that enable authorities and brokers to allocate and control virtualized hardware.

A goal of transitioning COD to use the lease abstraction described in the previous chapter is to support the allocation of virtual, as well as physical, machines using a single guest programming abstraction. Guests and sites have the option of choosing whether or not to use physical machines, which offer control of the raw hardware, virtual machines, which combine slivering with a set of hardware management services, or a combination of the two. Virtualizing hardware by combining slivering with hardware management capabilities, such as virtual machine migration, enables resource management control that is more powerful and flexible than physical hardware management. The slivering and hardware management capabilities of virtual machines, virtual networks, and virtual storage offer fundamental tools for resource management. As a result, any system that controls virtualized environments must address the allocation and binding of slivers.

5.1 Example: Virtual Machines

We focus primarily on the use of slivering and hardware management services in the context of virtual machines, although these capabilities are independent of a specific virtualization technology.

For instance, virtual storage devices permit slivering storage into statically partitioned virtual block devices or file systems, and exporting hardware management services, such as storage snapshotting and cloning. We use the term *host* to refer to a physical hardware device, such as a physical machine or physical block device, and the term *logical unit* to refer to a virtual hardware device, such as a virtual machine or virtual block device, that executes on the host and supports sliver binding and hardware management services.

We refer to each individual dimension of a sliver, such as CPU capacity or memory allotment, as a *grain*, and use the number of grains to refer to the relative amount of a single resource dimension. For example, actors may equate 50 grains of CPU capacity to a 50% CPU share while equating 200 grains of memory to 200MB of memory; in each case, the grain represents a unit of allocation specific to the hardware. The use of the term grains is meant to decouple the partitioning and isolation strategies of different resource dimensions from the specification of slivers: regardless of the partitioning technology, actors represent slivers as multi-dimensional vectors of numerical grain values.

The architectural principles and mechanisms in this chapter are independent of the fidelity of any specific slivering mechanism. Fidelity is a metric that quantifies the accuracy of a slivering mechanism over time. For example, a low-fidelity slivering mechanism that specifies a virtual machine should receive 30% of the processor time over a given interval may only provide 20%. Improving mechanisms to achieve high-fidelity and low-overhead slivering is an active area of research [82]. Existing virtual machine slivering mechanisms have been shown to be useful for controlling service quality for software services, such as a multi-tier web application [136].

5.2 Overview

Sliver binding raises a number of issues about the division of control in a networked operating system. For example, a service manager request to grow or shrink a sliver bound to a logical unit may trigger a migration, if there are not enough resources on a logical unit's current host to satisfy the request. Since brokers approve requests, they must convey information about the old and new host to the authority that conducts the migration since slivers and logical units exhibit *location dependencies* that bind them to a specific host. When allocating aggregations of slivers and logical units, brokers

must communicate to authorities the information necessary to preserve location dependencies.

The original research on Sharp and COD does not address how to delegate control of hardware management and sliver binding to service managers. COD virtual clusters were initially designed for the allocation of entire hosts [39, 120], and while previous Sharp systems [72, 73] address sliver allocation for individual logical units, they do not address the flexible allocation of collections of slivers or the delegation of hardware management to service managers. To demonstrate this point, we initially consider the benefits and drawbacks of a simple solution—site-assigned computons—that requires no changes to the original frameworks and protocols. After considering this solution, we conclude that Sharp and COD need explicit mechanisms to accommodate sliver binding and hardware management. We discuss a series of extensions that provide the mechanisms by explicitly naming individual hosts, slivers, and logical units at the broker and authority.

We extend Sharp and COD to address the sliver allocation and management capabilities of virtualized hardware, and frame our discussion of the principles around the naming information present in Sharp tickets. However, the principles of exposing names and secure bindings are independent of the Sharp ticket construct and the authority/broker policy division. The principles enable mechanisms that are necessary for service managers to reference slivers and logical units at a single authority without broker interaction. In particular, the mechanisms discussed in Section 5.3.3 and Section 5.3.4, which discuss augmenting tickets issued by a broker, and the mechanisms discussed in Section 5.4, which discuss secure binding of slivers to logical units, are also necessary in an architecture where the broker and authority operate under the same sphere of authority.

The discussion in this chapter also applies to authorities that internally separate their arbitration/provisioning policy from their assignment policy, similar to the separation of policy defined by a Sharp broker. An internal separation is useful to permit an authority to define the arbitration/provisioning and assignment policies, which may have conflicting goals, independently, as described in previous work [81].

The information conveyed in Sharp tickets is a useful basis for framing the discussion, since tickets, allocated by brokers, and leases, allocated by authorities, combine to convey information to service managers about their allocated resources. Brokers and authorities do not expose names for hosts, slivers, and logical units to service managers in Sharp. Service managers cannot reference

individual slivers or hosts or differentiate between them to control the binding of logical units to hosts or slivers to logical units. The result is that while service managers are able to lease sets of resources in aggregate, they are not able to operate on individual resources within each lease. For example, Section 5.3.4 describes how the ability to reference a sliver enables a service manager to select victim slivers for a lease that is shrinking on an extension—the capability is crucial if a service manager values one sliver within a lease over another. Since service managers cannot name each individual logical unit or sliver, authorities also cannot export control of individual hardware management services for a logical unit, such as virtual machine hardware management functions (*e.g.*, save/restore, migration). We discuss exposing names and secure bindings, summarized below, in the following sections.

- **Exposing Names.** Specifying names for hosts and slivers in Sharp tickets, allocated by brokers, communicates location dependencies to the authority and accommodates mechanisms for controlling sliver allocation.
- **Secure Bindings.** Logical units and slivers are separate entities. Naming logical units in leases allows authorities and service managers to treat them separately, enabling service managers to control the binding or unbinding of multiple, potentially remote, slivers to a single logical unit. Naming logical units at the authority interface also permits an authority to expose access to hardware management services by separating the naming of the logical unit from the slivers bound to it.

5.3 Exposing Names

To understand why Sharp tickets do not accommodate flexible sliver allocation and arbitration, recall from Section 4.1, that each Sharp ticket defines two distinct numerical values—a resource type and a resource unit count—and consider an example of an authority assignment policy for host allocation: the policy maintains a one-to-one mapping between each resource unit represented in a redeemed Sharp ticket and each host. To fulfill a ticket redeemed by a service manager, the assignment policy need only select and configure a single host of the proper resource type for each of the resource units specified in the ticket, as shown in Figure 5.1. Now consider an example of an assignment policy for logical units with slivers using the same Sharp ticket representation. What

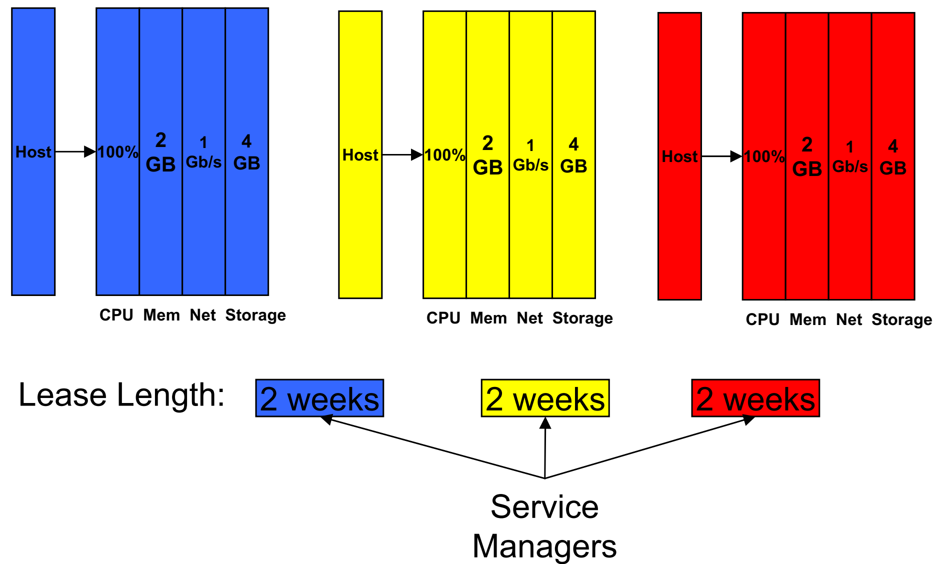

Figure 5.1: SHARP tickets specify a resource type and resource unit count. COD associates each resource unit with a single host.

should the resource type represent? Does it refer to the type of host, as in the previous case, or does it refer to a specific grain of a sliver, such as a CPU? What should the resource unit count represent? Does the resource unit count refer to the number of logical units, similar to the previous case, or to the number of grains of a sliver, such as 50 grains of CPU? Authorities and brokers must resolve these questions to implement sliver assignment and arbitration policies, respectively.

This section outlines a progression of candidate solutions within each subsection to answer these questions and discusses the benefits and drawbacks of each candidate. Each candidate solution provides a successively greater degree of freedom for guests to control resources. We give a brief description of each candidate solution below, before providing a more detailed discussion.

- **Site-assigned Computons.** Computons pre-define fixed size quantities for each grain that forms a sliver. With site-assigned computons an authority associates a computon with a resource type. The approach simplifies sliver allocation by reducing the flexibility a broker has to determine size of slivers it allocates.

- **Broker-managed Hosts.** Brokers, instead of sites, can determine sliver allocation by augmenting the information in SHARP tickets to specify each sliver's size as a multi-dimensional vector of numerical grain values. However, supporting broker-managed hosts by augment-

ing Sharp tickets with a vector of grain values constrain's sites' freedom to control sliver placement.

- **Broker-guided Colocation.** Brokers can allocate tickets for multiple slivers if, for each allocated sliver, they provide a name for the sliver's logical host. Providing a name for the host gives authorities the information necessary to map a logical host to a host and to determine a feasible sliver-to-host mapping.

- **Broker-guided Colocation with Sliver Naming.** Naming each sliver in a ticket allows a broker to allocate the different grains that comprise a sliver in separate leases. The flexibility affords a separation of grains that do not have the same lease lifetime, such as CPU and storage, into different leases since a service manager is able to reference previously allocated slivers in subsequent requests.

5.3.1 Site-assigned Computons

The first candidate solution we consider is site-assigned computons, depicted in Figure 5.2. Computons have been proposed as an abstraction for simplifying the allocation of slivers—they pre-define the number of grains along each dimension of a sliver [3]. With site-assigned computons, a site divides its hosts into computons and then delegates the power to allocate computons to brokers. As a result, resource requests from service managers must conform to the computon sizes defined by the site and offered by the broker.

Computons reduce complexity by predetermining the quantity of each grain that comprises a sliver and restricting sliver allocation to one dimension at a cost in flexibility. To illustrate, consider a site operating a single physical machine with a 4.0 GHz CPU, 8 gigabytes of RAM, and a gigabit I/O channel. A site may choose to partition its single physical machine into 4 computons. In this case, the site associates each computon with a resource type that represents the equivalent of a 1.0 GHz CPU, 1 gigabyte of RAM, and 256 Mb/s I/O channel.

Site-assigned computons require no extensions to the original Sharp ticket representation. With site-assigned computons, the resource unit count in the Sharp ticket represents the number of different logical units allocated and the resource type represents the type of logical unit and its computon. We use resource unit and logical unit interchangeably, since each extension below associates the re-

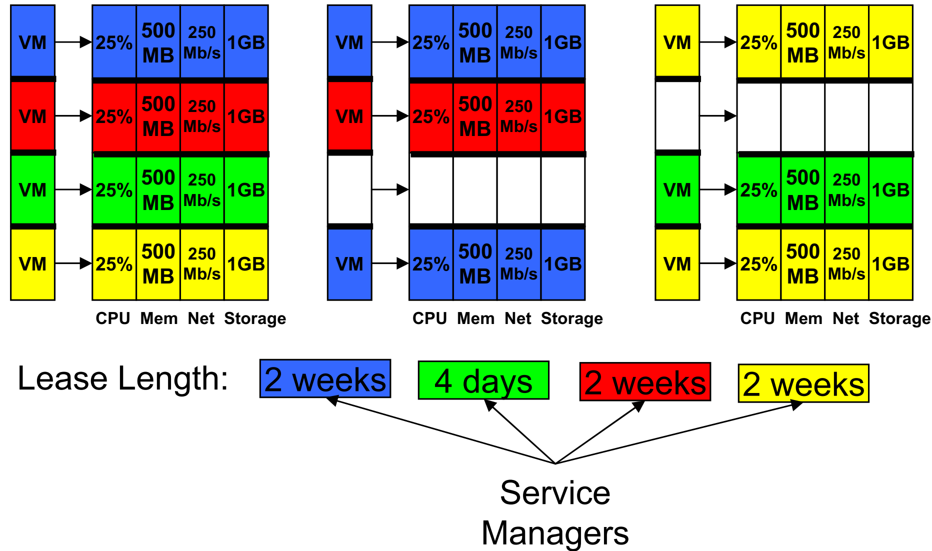

Figure 5.2: Each color represents a different lease. A computon is a predefined number of grains that comprise a sliver. To support logical unit colocation using computons, an authority associates a resource type with a computon and a resource unit count with a number of logical units. An authority assignment policy colocates multiple computons and logical units on each host.

source unit count with the number of logical units. We use the computon approach to support sliver and logical unit assignment to allocate virtual machines in the COD example in Section 4.4 without altering SHARP tickets.

To support sliver allocation in SHARP using only the resource type and resource unit count, authorities associate a resource type with a computon size, and a resource unit count with a number of distinct logical units bound to computons. The assignment policy for computons is trivial since it defines a fixed one-to-one mapping between each resource unit of a ticket and each logical unit bound to a computon. Initially, the simplicity of computons proved beneficial: the representation of SHARP tickets and the assignment policy did not need to change to allocate logical units and slivers since a computon-aware assignment policy is a simple extension of the initial host assignment policy. Instead of the assignment policy defining a one-to-one mapping between each resource unit and each host, it defines an n-to-one mapping between each resource unit and each host, where n is the site's predefined number of each computon of a specific type per host.

Site-assigned computons accommodate flexible sliver allocation if sites predefine a range of different size computons for hosts and associate the different computons with different resource types. The SHARP ticket protocol does not preclude service managers from altering their resource type on

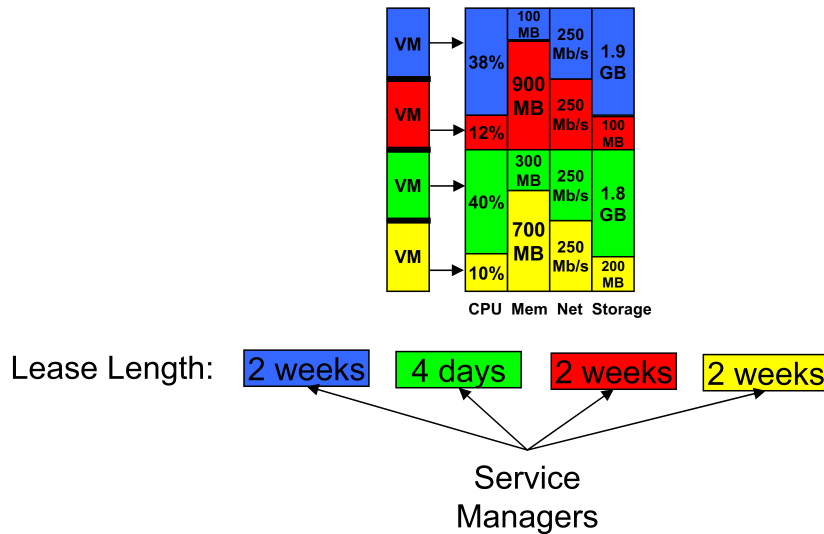

Figure 5.3: Authorities may permit brokers to determine the quantity of each grain that forms a sliver as long as each ticket includes information specifying the host. SHARP accommodates host specification if brokers specify a host by associating it with a resource type.

or before lease extensions, so brokers and authorities may support service managers that request to alter their slivers within the bounds of a site's predefined computons. The next section discusses the benefits of delegating the power to determine sliver allocation to a broker.

5.3.2 Broker-managed Hosts

We now consider a series of new naming and binding mechanisms that build on each other, starting with a candidate solution, called *broker-managed hosts*, in this subsection. In contrast to site-assigned computons, the broker-managed hosts approach allows the broker, as opposed to the site, to define and allocate slivers. The approach has two advantages relative to site-assigned computons. First, it gives brokers the power to satisfy service manager sliver requests precisely. Work on feedback control of web applications demonstrates that a service manager controlling a multi-tier web application, such as Rubis [36], adapts individual grains of CPU and memory allotment at precise granularities, not known *a priori*, to regulate service quality in response to load fluctuations [136]. To support this type of dynamic sliver allocation, we extend SHARP tickets to allow a brokers to allocate the grains of a sliver.

In order to support broker-managed hosts, we augment SHARP tickets to specify a single vector

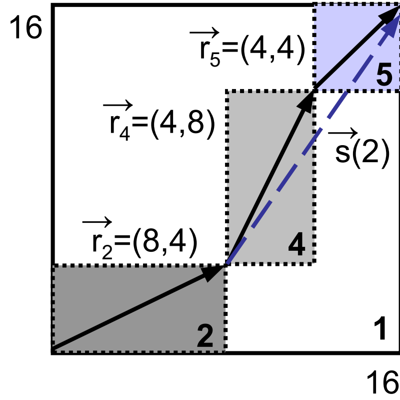

Figure 5.4: Brokers use vector addition to represent the two-dimensional packing of slivers onto hosts. The figure depicts a host with slivers that comprise 16 grains in two different dimensions. A broker carves tickets a, b, and c for slivers from the host.

of numerical grain values within each ticket, in addition to specifying the resource unit count and the resource type, as depicted in Figure 5.3. Brokers and authorities use resource properties (see Section 4.3.1) attached to tickets to store and retrieve this information: authorities attach the quantity of each grain to source tickets exported to brokers, and brokers attach the quantity of each grain in tickets allocated to service managers. Authorities then delegate the power for brokers to determine the partitioning of each host into slivers and allocate them to service managers, as depicted in Figure 5.3.

Figure 5.4 depicts a broker's representation of a set of numeric-valued grains in a vector, $\vec{r_i}$, for each ticket i. The number of grains and their granularity is a property of the host. The site specifies them in the source tickets it issues to brokers. For a source ticket j we refer to the dimension attribute vector—the space available in each unit—as $\vec{s_j}$. The broker "carves" tickets from its sources to fill requests without violating capacity constraints of the sources along each dimension.

Consider the example in Figure 5.4, which depicts three single-unit tickets with varying $\vec{r}$-vectors (slivers) carved from one host of a source with $\vec{s} = (16, 16)$. The total allocated from the host is given by vector addition: the host is full when $\sum \vec{r} = \vec{s}$. We define the space remaining on the host after ticket i is assigned to it as $\vec{s}(i)$. The next assignment of a ticket j to the same host is feasible if and only if $\vec{r_j} \leq \vec{s}(i)$.

Allowing brokers to manage hosts by allocating slivers introduces a conflict between the broker

and the authority. For each redeemed ticket, the authority must determine the assignment of slivers to hosts. To address the problem, we examine the information a broker stores in each SHARP ticket in order to guide the authority assignment policy to a feasible mapping of slivers to hosts. One method for providing mapping information without further augmenting each ticket is for authorities to associate each host with a different resource type. As a result, the SHARP ticket implicitly encodes the name of the host by specifying the resource type, and the assignment policy assigns the sliver to the one host associated with the ticket's resource type. Note that augmenting SHARP tickets to include additional information does not free the broker from performing the initial mapping of slivers to hosts, which is a variant of the NP-hard Knapsack problem if the broker associates each sliver with a value and seeks to maximize value [81]. Kelly gives a full treatment of the Knapsack problem and its relation to resource allocation, and demonstrates that even simple variants of the problem are NP-hard [100]. We discuss the authority assignment problem and its impact below.

Constructing a Feasible Assignment

Delegating control of sliver allocation and assignment to a broker requires the authority assignment policy to reconstruct the broker's mapping of slivers to hosts. Consider a SHARP ticket, which treats all resource units of a given resource type as interchangeable. In this setting, the assignment policy is a simple n-to-one mapping: an authority maps each resource unit to a host with an available computon. However, if a broker controls sliver allocation, the n-to-one mapping no longer holds since the broker is free to allocate the grains of a sliver in any quantity. As service managers redeem tickets for slivers, the authority assignment policy must construct a mapping of slivers to hosts. Reconstructing a broker's mapping of slivers to hosts is as hard as the NP-hard bin packing problem if the ticket only encodes the resource type, the resource unit count, and the amount of each grain that forms a sliver. We call this problem the *authority assignment problem.*

The formal definition of the bin packing problem is as follows: pack n objects, each with weight w, into the minimum number of bins of capacity c [46]. To prove that the authority assignment problem is NP-hard, consider an authority that receives a stream of tickets redeemed by service managers, where each ticket encodes a set of logical units each with an assigned sliver with a weight w_i that the authority must map to hosts (*e.g.*, bins). Without loss of generality we consider slivers composed of a single grain. Consider an authority that is able to solve the authority assignment

problem and map logical units to a set of m hosts given a set of n logical units and their weights w_i for $i = 1$ to n. To solve the bin packing problem, the authority need only repeat the process for $m - j$ hosts for $j = 1$ to $m - 1$—the solution to the bin packing problem is the minimum value of j for which the authority is able to find a mapping of logical units to hosts. Since an algorithm that solves the authority assignment problem also solves the bin packing problem we conclude that the authority assignment problem is NP-hard.

The classical bin packing problem considers a single dimension—an authority assigning slivers composed of multiple grains must consider multiple dimensions. Furthermore, the authority must deal with an online variant of the problem where each redeemed ticket alters the total number of logical units, which may necessitate both a new packing and a set of migrations that achieve the new packing.

Migration Freedom

In addition to constructing a feasible assignment, authorities may also alter the initial placement of slivers to hosts using *migration*. Authorities have an incentive to retain the freedom to assign and migrate slivers for load balancing. Previous work demonstrates that load balancing affects aggregate site power consumption and total cost of ownership [38, 119]. Sites may also leverage migration freedom to perform probabilistic host overbooking [164] or gain the flexibility to route around hardware failures. A design goal is to leave this flexibility in place.

An authority is free to change the mapping of a resource type to a host at any time as long as it reflects any change in the mapping by migrating all slivers from the old host to the new host if both hosts are identical. Changing the resource type to host mapping does not affect the complexity of the assignment policy since the mapping of slivers to hosts remains unchanged. Since the authority is overloading the resource type to represent both the physical host and its characteristics, an authority that alters the resource type to host mapping must ensure that the physical characteristics of the new host are equivalent to the physical characteristics of the old host.

An authority is free to deviate from the sliver to host mapping defined by each ticket if it is able to construct a feasible sliver to host assignment and a migration plan that specifies the sliver migrations necessary to shift to a feasible mapping. For example, an authority that deviates

from the colocation defined by a broker may find it impossible to satisfy a new ticket redeemed by a service manager. In this case, the authority must have a plan to migrate slivers to a feasible assignment. Constructing at least one feasible sliver-to-host assignment is trivial since each ticket encodes a logical mapping of each sliver to a specific resource type, which identifies a specific physical host. However, generating a migration plan that defines the minimal number of migrations to shift from one feasible sliver assignment to another feasible sliver assignment is a variant of the NP-hard migration planning problem [84].

Solutions exist if there is enough stable storage to cache instantiated logical units to disk, although caching logical units, such as virtual machines, to stable storage may impact performance and the total number of migrations may be unacceptably high. The problem is tractable for authorities at low utilizations, since there exist many different feasible sliver assignments; however, in the general case the problem requires exponential time. Simple heuristics can balance the tension between migration freedom and migration overhead. For example, an assignment policy that reserves space for each sliver at two hosts, the host defined in the ticket and a host of the site's choosing, permits an authority to freely assign slivers to hosts while retaining a trivial single hop migration plan as long as the site is less than 50% utilized. We note that the power of load balancing to reduce power consumption and total cost of ownership or perform resource overbooking is most important at these low utilizations [38, 119].

5.3.3 Broker-guided Colocation

We now consider a candidate solution where *brokers guide colocation* by extending tickets from Section 5.3.2 to specify a set of host names in each ticket, in addition to the sliver size specified by broker-managed hosts. The solution builds on the advantages of broker-managed hosts—flexible sliver allocation and migration freedom—by enabling service managers to request tickets for a set of slivers that span multiple hosts rather than a single host, as required in the previous subsection. Broker-guided colocation requires brokers to augment Sharp tickets to convey a logical name for the host of each sliver allocated in a ticket. The assignment policy requires the information to guide the assignment and migration of multiple slivers in a single ticket.

A naive method for augmenting tickets to include host naming information is for brokers to

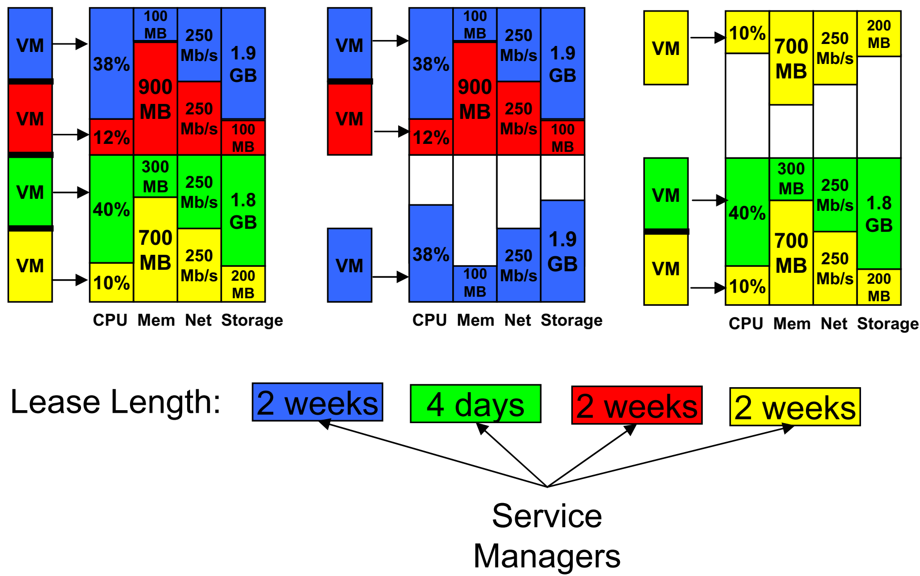

Figure 5.5: Augmenting each ticket with a list of host names permits brokers to allocate multiple slivers within each ticket. Each color represents a different lease.

associate a distinct logical *host name* for each individual resource unit allocated at a site: each ticket stores a host name for each resource unit (*e.g.*, slivers) allocated, as well as a resource type and a set of resource properties specifying the sliver size for each resource unit, as depicted in Figure 5.5. As service managers redeem tickets, an authority associates each host name with a single physical host and allocates a single sliver of the specified size to each of the hosts referenced in the ticket. Note that the list of host names in a ticket may include duplicates if a broker allocates two slivers to the same host. The host name is a logical reference to a host at the site, and the site may alter the mapping between a host name and a physical host at any time.

Either the site or broker may initially define the host names. The only constraint is that there is a unique host name for each physical host, such as a physical machine. For example, a site may define host names in the initial source ticket exported to the broker. Alternatively, a broker may define host names on-the-fly as it fulfills sliver requests; in this case, the site constructs a table mapping unique host names to hosts as service managers redeem tickets. Our prototype uses globally unique identifiers to ensure host names are unique and allows a broker to generate host names on-the-fly; each authority maps host names to physical hosts as service managers redeem tickets.

Defining host names that identify a host for each resource unit in a ticket enables the allocation

of multiple slivers in a single ticket and provides a degree of migration freedom at the authority without forcing it to solve the authority assignment problem from Section 5.3.2 to reconstruct a broker's colocation decisions. However, naively storing n host names in a ticket that allocates n resource units redefines the relationship between ticket size and the resource unit count in SHARP: rather than the size of each SHARP ticket being a constant, the size of each ticket scales linearly with the number of allocated resource units.

The tradeoff between ticket size and naming is fundamental: enabling a range of control mechanisms for service mangers requires that brokers and authorities independently expose the names of each host and, as we describe below, each sliver and logical unit. We note that while the size of SHARP tickets is constant relative to the resource unit count, each lease, as described in the previous chapter, returned by an authority to a service manager contains per-unit properties that scale linearly with the number of resource units.

A linear relationship between the size of the lease and the resource unit count is unavoidable if a service manager requires a method for contacting each individual logical unit, which requires a unique name, such as an IP address, to coordinate communication. If we restrict sliver allocation to a single lease then there exist optimizations for creating sublinear representations of the sliver to host mapping, as described in [80], that do not require augmenting each ticket with n host names. These representations reduce space by treating multiple hosts as a single pool, and conveying group sliver allocation as a deduction from this pool. If there is enough idle space on each host in the pool then group sliver allocation requires only a constant size ticket, since the ticket need only convey that one sliver should be drawn from each of n hosts in the pool.

As a broker allocates slivers, the pools fragment into multiple smaller pools. In the worse case, the broker must allocate n slivers to n different pools to satisfy a request; in this case, the ticket size is linear with the resource unit count.

5.3.4 Broker-guided Colocation and Sliver Naming

There is a benefit to exposing the name of each host, regardless of the ticket size: service managers may separate the allocation of different grains of a sliver into multiple leases. In this section we extend broker-guided colocation from the previous subsection by augmenting SHARP tickets to include the names of slivers in addition to the names of hosts. Naming slivers is beneficial for two

reasons. First, guests are able to separate the different grains of a sliver into different leases. For example, service manager requests for storage leases, which tend to be on the order of days, weeks, and months, are distinct from requests for CPU share leases, which tend to be on the order of seconds, minutes, and hours.

Second, a guest is able to convey preferences to a broker for each grain independently. For example, a guest that requires a lease extension for both storage (*e.g.*, to store a virtual machine's root disk image) and additional CPU share (*e.g.*, to improve performance) is able to separate the requirements into two broker requests. The capability allows a guest to separate requests that serve different needs. For example, a rejected request for additional CPU share only slightly degrades guest performance, whereas a rejected request for storing a root disk image terminates a virtual machine.

To cover the need to specify different grain preferences in our prototype, we initially tagged each requested grain with a simple boolean `elastic` attribute that specified a binary preference for each grain using request properties. If a service manager sets `elastic` to be true for a grain then the broker assumes that allocation of the grain is not critical, so it is free to allocate less than the requested value, as in the case for requesting additional CPU. In contrast, if a service manager sets `elastic` to be false then the broker must either allocate the requested grain in full or not at all, as in the case for requesting storage for a virtual machine root disk image. The `elastic` attribute is a binary representation that captures two different preferences a service manager is able to convey for the individual grains that comprise a sliver. Rather than conveying preferences for individual grains, an alternative approach splits the allocation of grains that comprise a sliver across multiple leases. This approach removes the need for individual preference semantics, as we discuss next.

In order to independently lease the different grains of a sliver we expose a sliver name for each resource unit of a lease and associate each sliver name with a host name, as depicted in Figure 5.6. Brokers generate unique sliver names for initial service manager sliver requests. Service managers use the initial sliver names generated by the broker in subsequent requests for additional grains for a particular sliver. If a ticket allocates n resource units then the broker specifies n host names and a mapping from each host name to each of n sliver names. In this model, the host name maps to a host at the authority as before, and the sliver name maps to a logical unit at the authority.

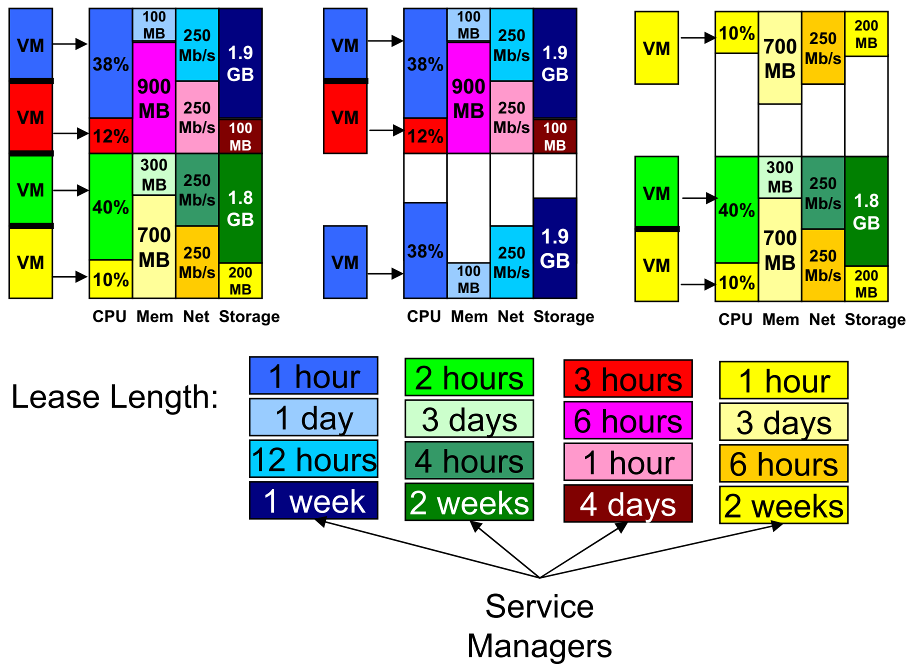

Figure 5.6: Naming individual slivers permits brokers to separate the different grains of a sliver into different leases, enabling a range of mechanisms that require sliver identification, including generalized victim selection, lease forking and merging, and host selection.

To request additional grains for a block of leased logical units, a service manager appends the names of a set of existing slivers to a ticket request. If a request is for n logical units, then the service manager attaches n sliver names to the ticket, and the broker allocates the additional grains from each host mapped to each existing sliver. Service managers are able to request multiple slivers within a single ticket, even if a broker has previously allocated grains for a subset of the slivers. For example, a service manager may attach only s sliver names to a request for n logical units, where $s < n$; in this case, the broker simply generates $n - s$ new sliver names, maps them to hosts, and returns the s existing sliver names and the $n - s$ new sliver names in the ticket.

Broker may need to alter the sliver name to host name mapping if the current logical host of a sliver does not contain enough grains to satisfy an additional request. Authorities must consent *a priori* to this type of broker-driven migration of slivers since they must account for these unexpected sliver migrations in their migration plan. If an authority does not consent then the broker cannot fulfill the request for the sliver.

Grouping Slivers

As described above, naming slivers enables service managers to request the grains of a sliver across multiple leases. Authorities require a mechanism for associating the binding of leases that comprise a sliver to a single logical unit. A simple grouping mechanism, similar to lease groups used by service managers in Section 4.3.2 to sequence the invocation of logical units and the binding of slivers, is sufficient for this purpose.

For example, when an authority detects a new sliver name it records the sliver name, instantiates a new logical unit, and binds the logical unit to the grains specified by the sliver. The authority treats each subsequent ticket that contains the sliver name as an ancestor of the instantiated logical unit, and uses the relationship to pass properties of the logical unit to the resource handler that binds subsequent slivers. As with the lease groups in Section 4.3.2, resource handlers for slivers must reference the host and logical unit properties, such as the internal IP address of a virtual machine monitor and a virtual machine name, to alter the grains bound to a logical unit.

The primary difference in this grouping mechanism and the mechanism described in Section 4.3.2 is that brokers specify the predecessor relationships for each sliver, and not for each lease. However, the architecture does not preclude brokers or authorities from restricting service managers to altering the sliver size of entire blocks of leased logical units in unison; in this case, the grouping mechanism closely resembles an authority-side implementation of the lease grouping mechanism in Section 4.3.2, where tickets that specify existing sliver names encode the specification of a *setup*, as opposed to a *join*, predecessor.

The reason for the granularity difference (*e.g.*, per-sliver as opposed to per-lease) in the two grouping mechanisms is that service managers use the lease groups from Section 4.3.2 to configure blocks of virtual machines for a guest and not individual slivers. Since we have not found a need to alter the configuration of a guest or its virtual machine due to the allocation of a sliver, we have not found a need for a service manager lease grouping mechanism at a granularity finer than a lease. For brokers and authorities, which handle the allocation of slivers to virtual machines, a finer granularity grouping mechanism allows service managers to request individual slivers for a single virtual machine in a block of requests.

Note that separating the grains of a sliver into different leases does not prevent leases for multiple

resource units—host and sliver names are sufficient for an authority to map the grains specified in a lease to the correct host and sliver. We retain the key property of Sharp tickets: each ticket specifies a lease for a block of equivalent resource quantities (*i.e.*, the size of the sliver). The Sharp delegation model relies on uniform resource quantities within a lease. However, broker-managed colocation with sliver naming eliminates the property of Sharp tickets that dictates that the units of a lease are interchangeable. Interchangeable units are not central to Sharp's delegation model, as described in [80], and are necessary to separate the lease lifetime of different grains, and enable new service manager control mechanisms, as described below.

Generalized Victim Selection

In many cases, guests need to select which sliver to revoke for a given lease. For instance, a batch scheduler forced to shrink the number of units in a lease due to contention has an incentive to revoke an idle sliver, rather than a sliver that is a week into executing a month-long task. Control of victim selection is especially crucial if a guest does not have mechanisms to suspend/resume work on a sliver or migrate the execution of work to an equivalent sliver. In Section 4.4, we describe how a COD guest overloads configuration properties, and selects victim virtual machines using their private IP address.

Naming slivers using an IP address is not ideal for three primary reasons. First, an IP address is an attribute of a specific logical unit (*e.g.*, a virtual machine) and not a sliver. As we see in the next subsection, naming logical units allows authorities to bind multiple slivers to a single logical unit. Furthermore, the model should be general enough to support victim selection for other logical units, such as virtualized storage partitions or networks, which have slivering capabilities, but are not typically associated with IP addresses. As a result, not naming slivers in tickets requires each assignment policy to define its own sliver naming scheme. Second, even assuming virtual machine allocation, an authority may not allocate each virtual machine its own IP address if a single leased virtual machine acts as a gateway to a virtualized private network with an IP namespace unknown by the site [24]. Third, a virtual machine may contain one or more private and/or public IP addresses at any given time. As a result, the service manager and authority must agree on conventions for naming machines based on IP address.

In Section 4.4, we circumvented this problem by associating each virtual machine with one and

only one IP address and dictated that service managers name each virtual machine by its IP address. Sliver naming enables a general mechanism that is not bound by the networking substrate to select specific victim slivers to revoke for each lease.

Lease Forking and Merging

Service managers have the power to request blocks of slivers in a single lease. However, after a broker grants a lease, the service manager is free to renegotiate the conditions of the lease at any time. Service managers require sliver naming to renegotiate with a broker to fork a lease into two separate leases with different conditions. For instance, a batch scheduler that leases a block of slivers to execute multiple tasks may need to renegotiate the lease of a single sliver running an important computation. The new lease conditions may increase the number of grains that comprise the sliver or lengthen the lease term to provide an assurance that the task completes before a specified deadline. The capability is especially important if the service manager cannot migrate the task to a new sliver. In this case, the service manager does not need to renegotiate the conditions of all of the slivers in the original lease.

Naming slivers allows the service manager and broker to convey which sliver to fork into a new lease, alleviating the need to restrict lease renegotiation to all the slivers within the lease. The opposite of lease forking is lease merging. Service managers or brokers may wish to renegotiate with each other to consolidate slivers from different leases into a single lease to reduce the overhead of lease maintenance if the slivers have the same size and lease term. As described in Section 8.4, the overhead of lease management is the primary metric that determines performance of service managers, authorities, and brokers.

Host Selection

Naming slivers accommodates a broker that reveals information to each service manager about the mapping of slivers to hosts. A broker may choose to reveal the information for two primary reasons. First, the information gives service managers the power to specify a host for a requested sliver. A service manager may want to specify hosts if it is leasing "best-effort" slivers that have no reserved capacity, and only use excess capacity; in this case, hosts with no other colocated slivers are most desirable. A host with no other colocated slivers is also desirable if a service manager is planning for

future growth of the sliver to meet anticipated demand. If the authority permits broker migration, a service manager may request that a broker migrate a sliver to a host that has no colocated slivers.

Second, pushing information to the service manager to control host selection simplifies the allocation problem in the broker. Brokers that hide the information must allow service managers to convey their preferences for multiple sliver possiblities and map those preferences onto one or more hosts by solving a variant of the Knapsack problem. In the most general case, a broker that does not expose information must allow service managers to specify resource requests as utility functions that the broker then uses to optimize global resource allocation. A broker that exposes information, such as host selection, has the option of pushing this complexity to the service manager, which is in a better position to decide its local allocation choices given enough information, at some cost in global utility. For example, instead of the broker deciding which host a sliver maps to, a service manager has the freedom specify the host to sliver mapping it desires.

Sliver naming and host selection ensures that the architecture accommodates either type of broker—one that optimizes for global utility or one that exposes control for each service manager to optimize their local utility.

Aggregation vs. Separation

Naming hosts and slivers enables new control mechanisms that allow service managers, brokers, and authorities to separate the different grains that form a sliver into different leases. Naming information also allows actors to reference specific slivers and hosts within a lease. Exposing these names has two drawbacks: scalability and complexity. We discuss scalability in Section 5.3.3. SHARP tickets provide enough information to perform the same lease operations on sets of slivers and hosts in aggregate. However, the lack of naming prevents lease operations that treat grains, slivers, and hosts separately, as described above. The naming information gives service managers, brokers, and authorities more flexibility, but the flexibility comes at a cost in complexity. Actors must reason about, track, and operate on individual slivers and hosts, rather than entire leases. It is important to note that exposing this information does not force service managers, brokers, and authorities to use it, and does not preclude the aggregate model in SHARP.

5.4 Secure Bindings

The previous section details how broker-managed colocation along with naming hosts and slivers enables useful mechanisms. In this section, we discuss how naming logical units enables a general method for authorities and service managers to bind to, and control, the hardware management services of each logical unit. We then compare and contrast two approaches for managing logical units: an approach that only exposes hardware management services to authorities and an approach that delegates control of hardware management services to service managers using logical unit names. We explore these two approaches using virtual machines and physical machines as the examples of logical units and hosts, respectively.

5.4.1 Logical Unit Naming

Naming slivers enables mechanisms that eliminate subtle embeddings of resource management policy. The policies include the inability to separate the lease lifetime of different grains of a sliver, the inability to select victims of a shrinking lease, the inability to renegotiate the lease conditions of specific slivers, and the inability for service managers to select specific hosts for each sliver. In this section we extend broker-guided colocation with sliver naming to also include the naming of logical units.

Logical units and slivers are distinct entities: slivers represent the portion of a host's resources allocated to a logical unit, whereas a logical unit is a software layer that serves as an execution environment bound to sliver resources. In this subsection we discuss how disassociating slivers from logical units enables the binding and unbinding of multiple slivers to and from a logical unit over time, as depicted in Figure 5.7.

Broker-managed colocation with sliver naming described in Section 5.3.3 equates slivers and logical units: each sliver has a unique name and is bound to one and only one logical unit. As a result, the integration of COD described in Section 4.4 names sliver/virtual machine combinations using the virtual machine's private IP address, which is a configuration attribute that is unknown to the broker. Naming logical units enables an authority to offer mechanisms to service managers that treat slivers and logical units independently. Figure 5.7 illustrates the separation between virtual machine execution environments, one example of a logical unit, and a set of leased slivers that are

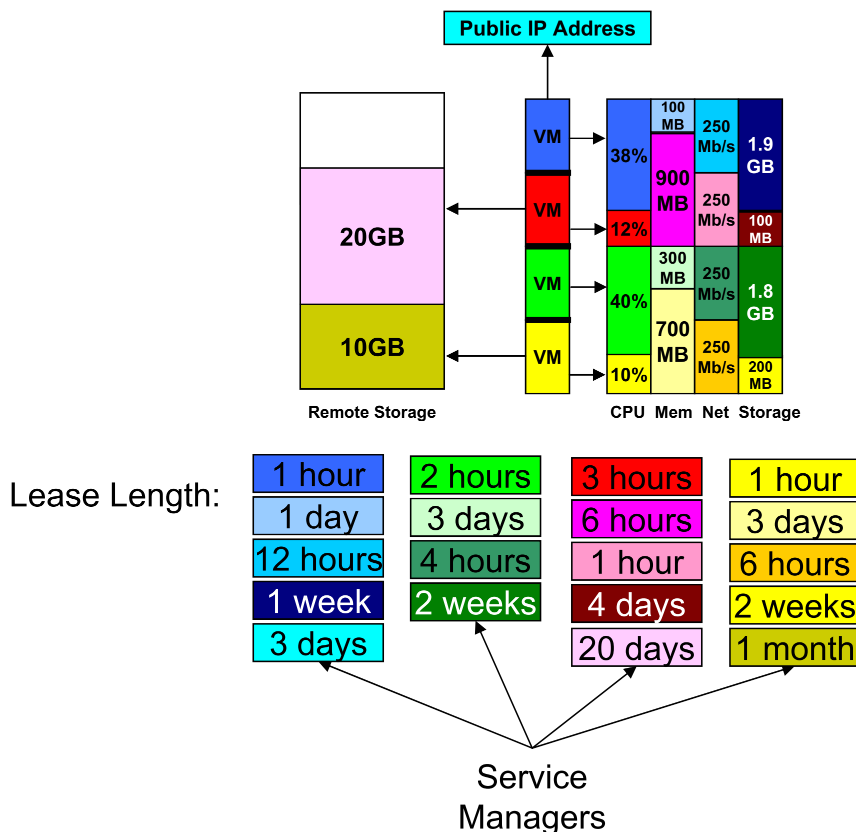

Figure 5.7: Naming logical units separately from slivers allows authorities to bind/unbind multiple slivers to a logical unit. These slivers may include remote slivers bound to different hosts, such as a remote storage server exporting a virtual machine root disk image. The scheme is general enough to also apply to logical resources, such as scarce public IP address space.

bound to that environment.

We give two examples of mechanisms that require a distinct name for each logical unit below. For each example, we give examples of the mechanism using virtual machines.

Remote Sliver Binding

Virtual machines bind to slivers on multiple hosts at distinct locations. This mechanism is useful for authorities that store virtual machine root disk images and other critical software on slivers of virtualized block-level or file-level partitions from one or more storage servers. Binding a virtual machine to a storage sliver is necessary to support remote root disk images. Remote sliver binding also enables authorities to use the lease abstraction to manage storage separately from other virtual machine resources, such as CPU and memory. This flexibility is necessary to store unused virtual

machine images until a guest or user requires them.

Virtual machine technology also exists that binds together slivers from multiple physical machines to present the illusion of a symmetric multiprocessor. Virtual Iron's VFe virtualization platform is an example of a virtual machine platform that combines slivers from multiple physical machines to provide the illusion of a single scalable symmetric multiprocesser.

Binding Logical Resources

Techniques for binding multiple slivers to a logical unit are general enough to allocate the logical resources of an authority. For example, this mechanism is useful for authorities that control a limited public IP address space. Authorities may represent each public IP as a resource type with an associated sliver and host, and delegate the power to arbitrate public IP addresses to a broker. Service managers may then lease public IP addresses and bind them to virtual machines using sliver naming and lease grouping mechanisms. The authority-side binding of a public IP address to a virtual machine may configure edge proxies to route traffic to and from the private IP address of the virtual machine, as is the case with Amazon's EC2 [74]. Our current prototype binds public IP addresses by logging into a virtual machine using a preinstalled key and configuring the virtual machine with the public IP address directly.

5.4.2 Delegating Hardware Management

Authority resource handlers for virtual machines in Section 4.4 define machine configurations that encode sliver binding as well as the specification of a root disk image and associated software packages. Figure 5.8 depicts the approach taken in Section 4.4: the authority reserves control of the hardware management functions of each virtual machine. As a result, the service manager selects a root image from a template of options and the authority creates the virtual machine using the *setup* handler. The authority transfers control of the virtual machine by configuring `sshd`, or an equivalent secure login server, with a public key the service manager transfers using a configuration property. The service manager has no ability to access a leased virtual machine's hardware management functions.

However, guests require a diverse range of different machine configurations and software packages for operation. To offer multiple machine configurations to guests, one approach is for site

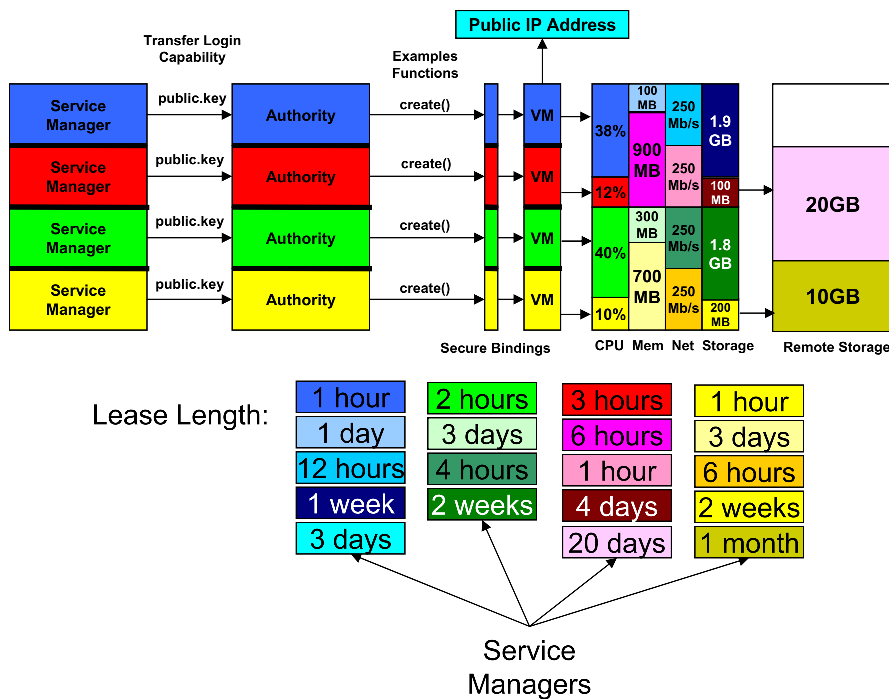

Figure 5.8: The figure depicts an authority that reserves control of the hardware management functions of each virtual machine. The service manager selects a root image from a template of options and the authority creates the virtual machine using the *setup* handler. The authority transfers control of the virtual machine to a service manager by configuring `sshd`, or an equivalent secure login server, with a public key, which the service manager transfers using a configuration property.

administrators to reserve control over the definition of machine configurations inside the resource handlers, as in Section 4.4. To retain this control and accommodate a diverse range of guests, site administrators may either create many different resource types and associate these types with different machine configurations, or overload configuration properties and write scripts beneath the resource handler interface that allow service managers to direct their own machine configuration.

Another approach is for authorities to permit service managers to upload new resource handlers and approve or sandbox them, similar, in principle, to how SPIN [27] downloads and sandboxes user-defined kernel extensions. This approach advocates neutrality through policy injection rather than policy removal. The policy injection approach requires site administrators to enumerate different possible machine configuration alternatives, and publish those alternatives for external service managers to develop against. Supporting a diverse range of guests by defining machine configurations or writing scripts to guide configuration is difficult if there are a numerous set of configurations. As a

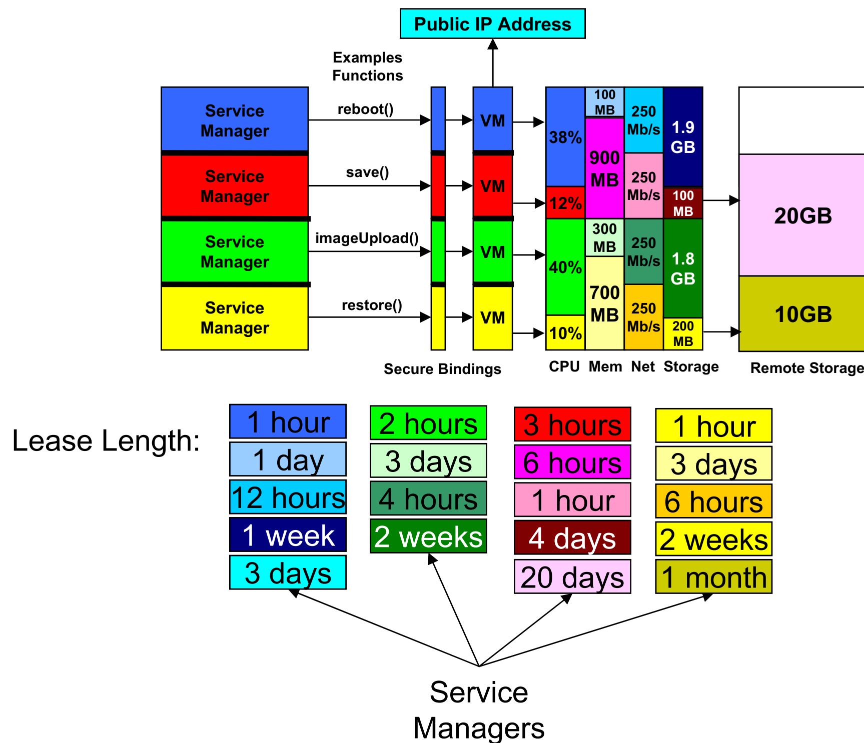

Figure 5.9: The figure depicts four service managers executing example hardware management functions for their leased virtual machines. Secure bindings permit service managers or authorities to access functions that control the hardware management services of a logical unit, such as a virtual machine. Delegating control of hardware management functions to service managers alleviates authorities from defining multiple software configurations and allows authorities to focus solely on hardware multiplexing and configuring access control for secure bindings.

result, we examine an alternative approach below that accommodates any number of guest-specific configurations by delegating a machine's management and configuration to the service manager leasing the machine.

5.4.3 Virtual Machine Examples

Delegating management of virtual machines to guests gives guests the same power as a site administrator on their subset of the infrastructure's resources, as depicted in Figure 5.9. Figure 5.9 illustrates four service managers invoking hardware management functions for their leased resources exported by an authority. The authority retains only the power to place slivers and bind/unbind slivers to virtual machines. Authorities delegate to guests the privilege to invoke hardware management services exported by an authority for each named virtual machine. This alternative does

not require site administrators to write resource handlers to support different machine configurations. In contrast, authorities limit resource handlers to controlling the immutable aspect of each resource—the binding of slivers to machines and the delegation of other management functions to the service manager.

The resource handler controls sliver binding and the machine's destruction at the end of each lease (*e.g.*, the abort protocol). Aside from assigning a physical machine for each sliver, sliver binding represents the only aspect of virtual machine instantiation and configuration that an authority controls. The design follows the Exokernel approach to limit privileged authority operations to resource multiplexing. Delegating management enables the holder of a lease access to hardware management services, such as those provided by virtual machines. Authorities that delegate machine management provide service managers control of the underlying hardware using a set of network-accessible APIs. Below we outline a list of hardware management services that service managers may find useful or even necessary. For each of these services we use "machine" to indicate that the service applies to both physical and virtual machines.

- **Upload a custom machine root disk image/kernel.** Authorities may expose a service to upload a custom root disk image with a custom kernel to a leased storage partition. The advent of virtual appliances as a standard mechanism for distributing applications with their operating system and root disk image makes this functionality a necessity for many guests.

- **Download a virtual machine root disk image.** Authorities can expose an external mechanism to download the contents of leased storage, including a virtual machine root disk image. The mechanism is similar in principle to Amazon's Simple Storage Service [74].

- **Instantiate a new virtual machine.** Authorities can expose a service to boot a new virtual machine given one or more sliver names that meet a minimum set of requirements for CPU, memory, bandwidth, and storage.

- **Bind/unbind a sliver to/from a virtual machine.** Authorities can expose a mechanism outside of the leasing protocols to alter the binding of slivers to virtual machines, as described in the previous section. For instance, a service manager may bind a root disk image on a local or remote storage server to a virtual machine prior to booting.

- **Shutdown or reboot a machine.** Authorities can expose an external interface to shutdown or reboot a virtual machine if network communication to the machine is impossible due to a machine crash, network partition, or network misconfiguration. This control is also useful for recovering from guest software and configuration failures.

- **Place a physical machine in a low-power idle state.** Physical machines have mechanisms, such as Wake-on-LAN, that allow site administrators to place them in a low-power idle state to conserve power and wake them up as needed. An authority that charges service managers for energy usage may allow service managers to place machines in an idle state..

- **Regulate the CPU frequency of a physical machine to reduce power.** Physical machines also have mechanisms to adjust the CPU frequency in accordance with resource demand to conserve energy. Authorities that expose this control to service managers enable them to meet system-wide power targets, as described in [139]. Service managers may also use regulation of their CPU sliver grain to produce soft power states, as described in [121].

- **Remote kernel debugging.** Node operating system developers often use virtual machines to ease the kernel development process. Virtual machines allow developers to substitute using a serial line for remote kernel debugging with a network connection attached to the virtual machine's virtual serial line. Authorities are free to expose this control to service managers to allow remote kernel debugging for leased virtual machines.

- **Console access to a virtual machine.** Debugging a corrupt root disk image or kernel requires service managers to have console access to a virtual machine that is unable to complete the boot process and service a network connection. Authorities may expose console output and console control to service managers.

- **Snapshot a machine root disk image.** Storage virtualization technologies allow snapshots, immutable copies of a storage device at an instant in time, to preserve data from an important instance in time. Authorities can give service managers the control to snapshot leased storage.

- **Clone a machine root disk image.** Storage virtualization technologies also allow snapshot clones that service managers may alter. Cloning snapshots is a common tool for quickly generating new machine root disk images from a template. An authority may delegate cloning

to a service manager that leases additional storage for a clone. The service manager may then bind the sliver specify the clone to a new virtual machine.

- **Reset a machine and its root disk to an initial state.** A service manager may wish to reset a machine that becomes corrupted to a fresh state. This is especially useful for developing new guests. Root disk images cloned from templates provide an easy mechanism for resetting a machine: delete the current root disk image, create a new clone from the template, and boot the machine.

- **Suspend, save, and download a virtual machine memory image.** Most virtual machines have the capability to save their in-memory state to disk. Authorities can allow service managers to save this state to leased storage and/or download the in-memory state.

- **Upload, restore, and resume a virtual machine memory image.** Service managers may wish to upload and restore the in-memory state of virtual machines. Authorities can allow service managers to upload previous virtual machine images and restore them.

- **Migrate virtual machines/data within a site.** A service manager may wish to shift data between two leased storage partitions or shift virtual machines between leased slivers on different hosts. The authority can expose a mechanism to perform these migrations without requiring service managers to go through the download/upload process.

- **Migrate virtual machines/data between sites.** A service manager may wish to shift data or virtual machines to storage partitions or slivers at a different site. The authority can expose a mechanism to coordinate cross-site migration if the service manager delegates the authority an appropriate authentication mechanism: a variation of standard Sharp tickets is a suitable authentication mechanism to delegate this control.

- **Migrate slivers.** The previous two mechanisms address migration of virtual machines between two slivers leased by a service manager. A site may also expose the control to migrate slivers. This control is most useful for "best-effort" virtual machines that only use excess resources at each machine. In this case, a service manager may want to migrate a "best-effort" sliver to a physical machine with the greatest amount of excess resource without coordinating with a broker. Delegating control of sliver migration to service managers is problematic since

it introduces a conflict with each site's incentive to place slivers to balance load and each broker's incentive to allocate slivers to maximize global performance.

The mechanisms above represent only a small subset of the hardware management services possible with physical and virtual machines. Other hardware management services may aid in intrusion detection [93], flash-clone new virtual machine's from the memory of existing virtual machines [166], or enable configuration and kernel debugging [175, 101]. Future work may enable more extensible virtual machines which allow sites to delegate each service manager the control to upload and link their own custom hardware management services or define their own custom hardware layer [86].

A consortium of industry virtualization leaders, including Dell, HP, IBM, Microsoft, VMware, and XenSource, have taken a step towards a common virtual machine layer with the Open Virtual Machine Format Specification (OVF) which describes a portable and extensible format for packaging and distributing virtual machines to any platform. The OVF format includes methods for verifying the integrity of virtual machine appliances from third-party vendors and enables tools to wrap virtual machines with meta-data that contains information necessary to install, configure, and run the virtual machine. The end goal of the OVF standard is to make virtual machines independent of the virtualization platform.

5.4.4 Delegation using Leases

The primary contribution of this thesis is not to demonstrate that authorities can export these existing hardware management services to service managers, but, instead, to demonstrate that the leasing architecture combined with sliver and logical unit naming can support service manager control of any existing or future hardware management service.

Authorities may use the *setup* and *teardown* resource event handlers to modify key-based access control to specific hardware management services from above. As with secure bindings in Exokernel, authorities only need to modify the access control once at the beginning of a lease and once at the end of a lease. Within a lease, a service manager is free to execute any hardware management services approved and exported by the authority for each named logical unit in its lease.

5.4.5 Guest Failure Handling

Exporting hardware management services to service managers has the benefit of unburdening authorities from addressing guest failures. In contrast, authorities delegate the power to allow each service manager to introspect on their own leased hardware, reboot a machine, upload a new kernel and/or root disk image, or alter a root disk image while a machine is shutdown. These mechanisms are crucial for guests to diagnose and recover from software failures without requiring authority or site administrator intervention.

From experience, we have seen that the primary source of failures in COD does not derive from bugs in the leasing protocols or physical or virtual hardware failures, but from software failures of guests. Without the proper mechanisms to allow guests to correct failures, the burden of correcting them devolves to the authority and site administrator, which know little about the hosting guest software. We note that delegating this control is not a panacea for failures, but it does enable an avenue for failure correction that does not involve the authority or site administrator.

5.5 Exposing Information

As authorities and brokers expose more control to service managers there is a need for them to expose more information so service managers can utilize their new control. Examples from above that address the allocation of "best-effort" virtual machines exposes information to the service manager to allow it to control the binding of each sliver to a physical machine. For instance, the broker may expose the assignment of slivers to hosts so a service manager may select an appropriate host, or the authority may expose the control to migrate slivers, without the broker's knowledge, to allow a service manager to migrate a "best-effort" sliver to a host with the most excess resources.

Determining the level of information to expose to service managers is orthogonal to defining an architecture that does not constrain service manager's use of the hardware. We note that in many cases leveraging control requires exposing information about the hardware platform. For example, Planetlab exposes nearly all information about resource usage to third-party services that allow guests to make queries about available resources [124] and freely migrate to machines with the most excess resources [125]. However, some authorities and brokers may consider this type of allocation information privileged and restrict exposing both the information and control of sliver migration

to service managers, as in Amazon's Elastic Compute Cloud [74]. Ultimately, the decision of how much control to delegate and information to expose to service managers rests with each site.

5.6 Summary

In this chapter we identify naming of hosts, slivers, and logical units as a design principle of a networked operating system and explore the benefits and drawbacks of different alternatives and approaches to naming. The naming schemes of authorities and brokers determine the mechanisms that they can offer to service managers. SHARP did not name hosts, slivers, or logical units. As a result, ticket size is independent of the number of allocated units, but the naming scheme prevents mechanisms that operate on individual slivers and/or logical units. The original naming scheme is appropriate if these mechanisms are unnecessary. Our goal is not to determine the appropriate granularity of naming, but to demonstrate the importance of naming in a networked operating system, show that our architecture accommodates a range of approaches, and describe the impact of naming on the mechanisms the architecture is able to support.

We show how SHARP tickets accommodate different naming schemes in a brokered architecture by augmenting tickets with names for hosts, slivers, and logical units. The naming scheme is general enough to apply to any virtualized resources, including network and storage virtualization, that offers hardware management services and slivering capabilities. Finally, we show that naming virtual machines enables sites to export hardware management services to service managers for their leased machines.

Chapter 6

The Lease as an Arbitration Mechanism

"I can imagine no society which does not embody some method of arbitration."

Herbert Read

Examples of guests that share an infrastructure include content distribution networks, computational grids, hosting services, and network testbeds such as PlanetLab. The growing reach and scale of these systems exposes the need for more advanced solutions to manage shared resources. For example, the tragedy of the PlanetLab commons is apparent to any PlanetLab guest or observer, and there is now a broad consensus that networked resource management needs resource control that is strong, flexible, and fair. While overprovisioning is an acceptable solution in centrally controlled environnments, networked environments composed of multiple self-interested actors require incentives to contribute so the collective system is sustainable.

Market-based resource control is a logical next step. The Internet has already grown convincing early examples of real, practical networked systems structured as federations of self-interested actors who respond to incentives engineered to induce a desired emergent global behavior (*e.g.*, BitTorrent [47]). At the same time, grid deployments are approaching the level of scale and participation at which some form of market-based control is useful or even essential, both to regulate resource allocation and to generate incentives to contribute resources to make them self-sustaining.

Service managers and brokers define adaptation and arbitration policies, respectively, that interact and negotiate to guide the flow of resources to and from multiple service managers. In our architecture these policies are pluggable—service manager and broker developers define them and plug them into the respective library implementation. Policy flexibility is necessary to accommodate a diverse set of site and guest objectives [79]. Markets are a natural, decentralized basis for arbitrating resource usage in networked systems composed of a community of resource providers and consumers. In this chapter, we demonstrate that leases are an arbitration primitive that is able to define a range of different arbitration policies, including market-based policies that define incentives necessary to avoid a "tragedy of the commons." In market-based allocation, a service

manager and broker negotiate exchanges of currency for resources. We leverage active research on market-based task services [16, 32, 44, 90, 134] to demonstrate the power of leasing currency to bridge the gap between proportional-share allocation and markets, by arbitrating resource usage between two independent market-based task services from our previous work [90].

6.1 Overview

Market-based arbitration defines incentives and price feedback for guests to self-regulate resource usage [34, 154, 156, 169, 180]. However, the use of computational markets for resource arbitration is still an active area of research. Below we summarize the complexities that computational markets and a networked operating system must address. These complexities amount to multiple policies that actors must define to ensure both an efficient global resource allocation and an efficient local resource allocation for each actor.

- Service managers, which control guests, such as batch schedulers, that require resources to service external clients, must determine how many resources they require to make their clients "happy." A service manager must first associate a notion of happiness, or utility (*e.g.*, a value that quantifies happiness), with a level of performance. The service manager must then associate a desired performance level with a quantity of resource; only then can the service map overall utility to a resource allotment.

- A service manager must purchase resources at some price. In a market, price is denominated using currency. The service manager must determine the price it is willing to pay for resources to achieve the desired level of performance, and hence utility. The price the service manager is willing to pay may depend on future events, such as the replenishment of currency, expected load increases, or competing service managers. Purchase price also determines future currency holdings and impacts future allocation decisions.

- An authority (or broker) uses an auction to set prices and clear bids. Auctions are well-studied in the literature and come in many varieties. Incentive-compatible auctions are important to ensure that services bid their true valuations. Clearing all but the simplest auctions require solving NP-hard problems. In particular, combinatorial auctions which clear bids for bundles

of resources, such as CPU share, memory allotment, and storage, are related to the Knapsack problem [100].

- The service manager, authority, and broker must continually repeat these steps to adapt as conditions (*e.g.*, competition, load, prices) change over time.

Market-based allocation is an active area of research. An arbitration primitive for a policy-neutral system must also be able to support more conventional means of arbitration, such as static priorities or proportional-share allocation. An lease-based arbitration primitive is a general mechanism that associates actors with identifiable currency tokens, which service managers, brokers, and authorities use to arbitrate usage under constraint. Our goal is to demonstrate the properties of leases as a basis for arbitration—we do not explore the full range of arbitration policies possible with this mechanism, such as combinatorial auctions for slivers, or how these policies interact with adaptation and assignment policies.

Service managers lease currency to brokers and authorities in the same way that authorities and brokers lease resources to service managers. We present a *self-recharging virtual currency* model: once the lease for currency ends the service manager is free to lease the currency to another broker or authority. Section 4.1 describes how our architecture derives from Sharp. Previous Sharp-based systems restrict arbitration to barter exchanges, which rely on consumers to identify mutual coincidences of needs or enter into transitive barter arrangements: a common currency is an important step toward an efficient market economy in which resource control is open, flexible, robust, and decentralized [45, 73].

We use the market-based task service to highlight the many different policy issues that arise with resource arbitration and markets, including mapping performance to resource allotment, mapping resource allotment to utility, mapping utility to currency bids, and dealing with uncertainty of future resource availability. Due to the market complexities, we use simple policies to show the benefits of a lease-based arbitration primitive: a service manager that plans future resource usage over a limited time horizon can achieve higher utility with self-recharging currency than using either a strictly proportional-share or market-based approach.

6.2 Self-Recharging Virtual Currency

The use of self-recharging virtual currency is an instance of the large class of market-based resource control systems in which self-interested actors use their currency to obtain resources at market prices, subject to their budget constraints (*e.g.*, [19, 34, 105, 154, 158, 169, 170]). *Currency* is denominated in some common unit that constitutes a standard measure of worth ascribed to goods. Each unit of currency is possessed exclusively by some system actor, who may transfer it to another actor in exchange for goods.

We use a virtual currency—called *credits*—rather than a fungible cash currency (*e.g.*, dollars) to arbitrate resource usage. Our premise is that external control over resource allocation policy is crucial in systems that serve the needs of a networked infrastructure. In a virtual currency system, consumers are funded with credit budgets according to some policy. As with other virtual currencies, a global policy may distribute and delegate credits in a decentralized or hierarchical fashion to meet policy goals [169, 170]. In contrast, cash markets allow real-world wealth to dictate resource access policy, and currency allotted by a global policy can be diverted to other purposes. We envision that credits may be purchased or redeemed for cash (or other goods) as a basis for a cash utility market or to reward contributors.

Money is any currency that functions as a store of value. The holder of money has full control over when and how it is spent. Money economies depend on a balance of transactions to recycle currency through the economy. Once money is spent, the spender permanently relinquishes ownership of it. Actors may run out of money, leaving resources idle even while demand for them exists (*starvation*). As they spend their money, they must obtain more, either by offering something else of value for sale, or by means of an income stream from an external source (*e.g.*, the state).

Actors may save or *hoard* income over time to increase their spending power in the future, and may use that accumulation to corner the market, manipulate the price of resources in the system, or starve other users. Some systems (*e.g.*, [43, 154]) limit hoarding by bounding savings or discourage it by imposing a demurrage (a tax on savings). Money economies are prone to cycles of inflation and deflation caused, in part, by fluctuations in the circulating money supply.

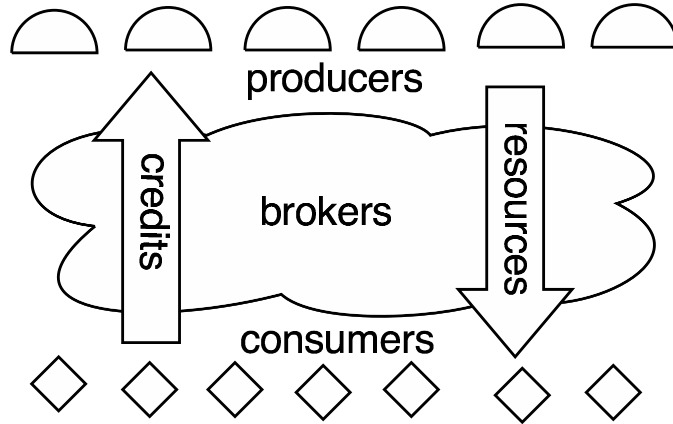

Figure 6.1: Flow of credits and resources in a networked infrastructure. Leasing rights for resources flow from sites down through a broker network to consuming service managers. Brokers allocate resource rights using any policy or auction protocol: payments in credits flow up through the broker network.

6.2.1 Rationale

As stated above, we outline a currency that is self-recharging: the purchasing power of spent credits is restored to the consumer's budget after some delay. In effect, the credits *recharge* automatically, ensuring a stable budget but bounding the opportunity for hoarding. The purpose of self-recharging currency is to eliminate reliance on fragile mechanisms to recycle currency through the economy. In particular, it is not necessary for each consumer to contribute in proportion to its usage, although a global policy could punish free riders by draining their credit budgets if desired.

Self-recharging currency refreshes each consumer's currency budget automatically as it is spent—credits recharge after a fixed *recharge time* from the time they are spent. The automatic recycling of credits avoids the "feast and famine" effects of money economies (hoarding and starvation). A consumer with a budget of c credits can spend them according to its preferences, substituting resources or scheduling its usage through time according to market conditions, but it can never hold contracts or pending bids whose aggregate face value exceeds c. The consumer can never accumulate more than c credits, and it can spend up to c credits in any interval equal to the recharge time.

Self-recharging credits are analogous to a money economy in which the state provides each actor with a steady flow of income over time, while imposing a 100% income tax to prevent hoarding. Actors may purchase goods by contracting to divert some portion of their income for the duration of the recharge time in exchange for the good: in effect, the state gives a tax credit for mort-

gaged income. The incentive to conserve income by delaying purchases is adjustable based on the recharge time. With a recharge time of zero there is no incentive to conserve, and the system reverts to proportional-share allocation. As the recharge time grows, additional mechanisms must be introduced to recycle funds, as in a money economy.

In a perfect market, each consumer is assured access to a share of resource value proportional to its share of the wealth. With self-recharging currency, this assurance applies to any interval whose duration is the recharge time. However, the purchasing power within any interval depends on the movement of prices through time. Self-recharging currency emphasizes the importance of adjustable incentives to schedule resource usage through time, in contrast to many previous systems which balance only instantaneous supply and demand.

Self-recharging currency is not a new idea: Lottery Scheduling [170] is one simple and popular example of a system with self-recharging currency, and the self-recharging credit currency model was inspired by Sutherland's 1968 proposal for a futures market in computer time [158]. Lottery scheduling and Sutherland's proposal are two points in the self-recharging currency design space: we generalize self-recharging currency to continuous, rolling, brokered, multi-unit futures auctions.

Lottery Scheduling [157, 170] uses a form of self-recharging currency that recharges on every time quantum. A lottery scheduler fills demand in proportion to instantaneous currency holdings; its single-resource single-unit auction protocol is a simple random lottery, making it efficient enough for fine-grained scheduling (*e.g.*, at CPU quanta). However, Lottery Scheduling is not a market: consumers receive no price signals and cannot defer their purchasing power into the future since they have no incentive to defer a resource request in response to spot shortages.

In contrast, self-recharging credits are redeemable for any resource available for sale according to prevailing prices within some time window. With a short recharge time the system is similar to Lottery Scheduling in that it approximates proportional-share scheduling and provides no incentive for saving. Longer recharge times provide more incentive for service managers to plan their resource usage over time, but begin to resemble a money economy with increasing risk of hoarding and starvation.

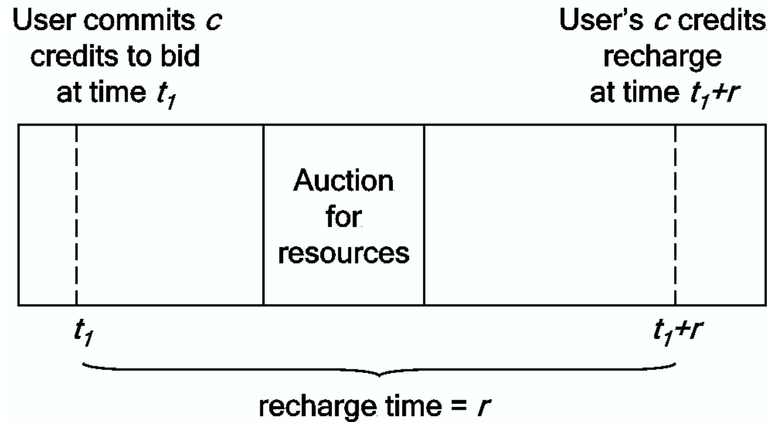

Figure 6.2: The *credit recharge rule* returns credits to the buyers after a configurable recharge time r from when the currency is committed to a bid in an auction, or if a bidder loses an auction.

6.2.2 The PDP-1 Market

Self-recharging currency was first introduced by Sutherland in 1968 under the name "yen" to implement a self-policing futures market for a PDP-1 computer [158]. Although the paper is widely cited, we are not aware of any subsequent published work that generalizes this form of virtual currency. We now describe credits in the PDP-1 market, then outline how we apply the concept to a continuous market for multiple resources.

Consumers in the PDP-1 market bid yen for time slots on the computer during the next day. In essence, the system runs an open ascending English auction for each time slot.

- Bidders commit their currency at the time it is placed for bid; the currency is held in escrow and is not available for concurrent bids. The protocol is compatible with an open (*i.e.*, exposed bids) or closed (*i.e.*, sealed bids) auction with a fixed call-time for bids.
- A high bid preempts a low one since only a single unit of each good is for sale; preemption is immediately apparent to the loser, which can reclaim its yen. If a subsequent bid preempts an existing bid or if a bidder cancels a bid, the currency is immediately available to the bidder.
- Each bidding period determines the allocation for the next day. The bidding period ends before the beginning of the day; once the bidding period ends no user is allowed to change bids.

- The market *recharges* the winning bid's currency after the resource is consumed (*i.e.*, after the purchased slot expires). The yen then become available for auctions for the following day.

6.2.3 Generalizing the PDP-1 Market

We extend the PDP-1 market to a networked market with multiple rolling auctions. As in the PDP-1 market, the bidder commits credits to each bid, and these credits are transferred at the time of the bid.

The crux of the problem is: *when to recharge credits spent for winning bids?* The PDP-1 market recharges credits when the purchased contract expires, which occurs 24-48 hours after the auction, depending on the time of day for the purchased slot. It is easy to see that the PDP-1 credit recharge policy is insufficient for a market with continuous rolling auctions. If credits spent for a contract recharge as soon as the contract expires, then a dominant strategy is to bid for instant gratification so that spent credits recharge sooner. The value of scheduling usage through time would be diminished, and the market would revert to a proportional-share as in Lottery Scheduling [170]. In general, a variable recharge time disadvantages buyers whose credits expire further in the future. The effect is negated in the PDP-1 market because the buyer always receives the recharged credits in time to bid for the next auction.

To maintain a consistent recharge time across multiple concurrent auctions, we enforce the *credit recharge rule*: spent credits recharge back to the buyer after a fixed interval—the *recharge time*—from the point at which the buyer commits the credits to a bid. The recharge time is a global property of the currency system. The credit recharge rule has three desirable properties:

- It encourages early bidding, yielding more accurate price feedback to other bidders, depending on the auction protocol.
- It discourages canceled bids, since shifting credits to another bid delays recharge of the credits.
- It encourages early bidders to bid higher, to avoid incurring the opportunity cost on any credits returned by the auction protocol for losing bids. The opportunity cost incurred by bidders is higher the earlier they commit their bids.

These properties address known limitations of many common auction protocols. For example, open ascending auctions with fixed call times as used in the PDP-1 market (Section 6.2.2) encourage predatory late bidding just before the auction closes, a property that the PDP-1 market has in common with Ebay. Also, open auctions tend to minimize profit to the seller unless potential buyers are motivated by uncertainty to expose their willingness to bid high: this problem is a key motivation for sealed-bid auctions. These concerns have also motivated the design of auction protocols such as the Hambrecht OpenBook auction for corporate bonds [2].

One implication of the credit recharge rule is that it is possible for a buyer to receive credits back before a contract expires, and thus it is possible for a buyer to hold pending bids and contracts whose face value exceeds its credit budget c. This occurs rarely if the recharge time is long relative to the lease terms, as in the PDP-1 market. Whether or not it occurs, it is easy to see that the key property of the PDP-1 market continues to hold: a consumer can commit exactly its budget of c credits in any interval whose duration is the recharge time. A decision to spend or save credits now does not affect the consumer's purchasing power in future intervals.

The credit recharge rule has a second implication for brokered auctions: credits passed to a broker expire to preserve the currency balance when they revert to the consumer's budget after the recharge time. As a result, brokers have an incentive to spend credits quickly. To simplify management of the currency, our prototype imposes a *binding bids* rule for credits: *credits committed to a bid become unavailable until they recharge.* Bidders cannot cancel or reduce their bids once they are committed. In this way, it is never necessary for a broker to return escrowed credits (*e.g.*, to a losing or canceled bid) after the broker has spent them.

The currency model is well-matched to *price-anticipating* auction protocols as in Tycoon [64, 105], in which brokers accept all bids but the service rendered varies according to contention. The self-recharging currency model is independent of the auction protocol: our experiments use an auction similar to Tycoon in which the broker allocates resources in proportion to bids. The protocol is simple and incentive-compatible for bidders; however, due to the credit recharge rule, the bidder's use of the credits is lost until they recharge regardless of the desired number of resources allocated. The credit recharge rule does not constrain the auction protocol. For instance, brokers may use common incentive-compatible auctions, such as a generalized Vickrey auction.

6.3 Using Credits in a Simple Market

In this section we demonstrate the benefit of self-recharging currency for arbitrating resource usage in a simplified market setting. We show that in this setting a service manager that plans well over the recharge time may achieve higher utility than using strict proportional share or market-based allocation.

Markets must define specific policies, described in Section 6.1, to ensure effective allocation. To demonstrate important properties of self-recharging currency we insert simple policies for each of these policies. While the policies lack the complexity of a true market, they allow us to separate the properties of the currency system from the properties of specific policies. Since our goal is not to define optimal bidding strategies, but to demonstrate properties of the self-recharging virtual currency model, the bidding strategies have knowledge about the future workload embedded into them and the auction protocols are simple. We do not examine bidding strategies that dynamically adjust bids based on the workload. Nothing in the currency model prevents the use of any type of auction, including combinatorial auctions [128].

We use a previously studied market-based task service, described below, that associates a utility function with each task. We then define static, unchanging service-specific and workload-specific bidding strategies for two competing market-based task services, and use a simple proportional-share auction, inspired by Tycoon, to divide resources between the two services. We vary the the recharge time of the currency and the planning horizon of the service manager bidding strategy to demonstrate the effects on the global utility of both services. Self-recharging currency's effect on global utility is independent of the different policies that define a market.

We use a simulator from our previous work to experiment with multiple market-based task services bidding for resources to satisfy their workloads using self-recharging virtual currency [90]. In our prototype, since all actors abide by the principle of visible allocation, modification, and revocation, they know when they receive resources in exchange for currency. Actors may transmit immutable properties of self-recharging virtual currency that encode identity and quantity using context-specific property lists. Furthermore, self-recharging currency leverages SHARP's secure resource delegation model to prevent actors from counterfeiting currency or recharging currency early.

Next, we present the market-based task service in detail. In particular, we outline the task

service's formulation of utility for tasks and the scheduling of tasks on the available resources in the service.

6.3.1 A Market-based Task Service

Batch job scheduling systems address resource arbitration with a combination of approaches including user priority, weighted proportional sharing, and service level agreements that set upper and lower bounds on the resources available to each user or group [75, 113, 183]. These scheduling mechanisms have largely been translated, with few adaptations, for use on the grid. Market-based approaches refine this capability by enabling users to expose the relative urgency or cost of their tasks, subject to constraints on their resource usage.

In value-based [33, 40] or *user-centric* [44] scheduling, users assign value (also called yield or utility) to their tasks, and the system schedules to maximize aggregate value rather than to meet deadline constraints or a system-wide performance objective such as throughput or mean response time. In economic terms, value maps to currency where users "bid" for resources as a function of their value, currency holdings, and competing bids. The system sells resources to the highest bidder in order to maximize its profit. This approach is a foundation for market-based resource management in infrastructures composed of multiple sites.

Here we use market-based scheduling in grid service sites based on user bids that specify value across a range of service quality levels. Representations of task value create a means to dynamically adjust relative task priority according to the workflow of the organization (*e.g.*, giving jobs higher priority when other activities depend on their completion). For example, the results of a five-hour batch job that is submitted six hours before a deadline are worthless in seven hours. The scheduler must balance the length of a task with both its present value and its opportunity cost. In short, it must balance the risk of deferring a task with the reward of scheduling it. The task scheduling model extends the treatment of task scheduling for linearly decaying value functions in the Millennium cluster manager [42, 44].

Background

We consider a *service market* in which each site sells the service of executing tasks, rather than hardware resources. In a service market, clients and servers negotiate contracts that incorporate

some measure of service quality and assurance as well as price. For example, clients may pay more for better service, and servers may incur a penalty if they fail to honor the commitments for service quality negotiated in their contracts. We present the simplest type of service: a batch task service. A market-based task service is based on three key premises, as described below.

- Batch tasks consume resources but deliver no value until they complete.
- A task submission specifies a resource request (service demand) to run the task, and correctly specifies the tasks's duration if its resource request is met.
- Each task is associated with a user-specified *value function* (utility function) that gives the task's value to the user as a function of its completion time. The next section discusses these value functions, which are used to specify bids and service contracts.

The customer and the task's service manager agree on the expected completion time and value, forming a contract. If the site delays the task beyond the negotiated completion time, then the value function associated with the contract determines the reduced price or penalty. As a result, task service managers must consider the risks inherent in each contract and scheduling choice.

Value Functions

Each site makes scheduling decisions based on the task value functions. These functions give an explicit mapping of service quality to value, exposing information that allows each task service to prioritize tasks more effectively. Value-based scheduling is an alternative to scheduling for deadline constraints, which give the system little guidance on how to proceed if there is no feasible schedule that meets the constraints (*e.g.*, due to unexpected demand surges or resource failures). Value functions also serve as the basis for a task service to define bids on resources in an underlying resource market, as we demonstrate in the next section with self-recharging virtual currency.

A value function specifies the value of the service to the user for a range of service quality levels—in this case, expected task completion times. The key drawback of the user-centric approach is that it places a burden on users to value their requests accurately and precisely. The more precisely users can specify the value of their jobs, the more effectively the system can schedule their tasks. The formulation of value functions must be simple, rich, and tractable. We adopt a generalization

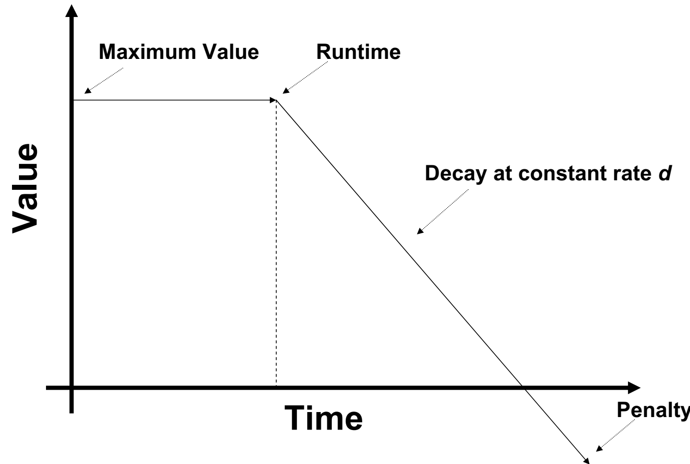

Figure 6.3: An example value function. The task earns a maximum value if it executes immediately and completes within its minimum run time. The value decays linearly with queuing delay. The value may decay to a negative number, indicating a penalty. The penalty may or may not be bounded.

of the *linear decay* value functions used in Millennium [44, 42], as illustrated in Figure 6.3. Each task i earns a maximum value $value_i$ if it completes at its minimum run time $runtime_i$; if the job is delayed, then the value decays linearly at some constant *decay* rate $decay_i$ (or d_i for short). Thus if task i is delayed for $delay_i$ time units in some schedule, then its value or yield is given by:

$$yield_i = value_i - (delay_i * decay_i) \tag{6.1}$$

These value functions create a rich space of policy choices by capturing the importance of a task (its maximum value) and its urgency (decay) as separate measures. Tasks are more urgent if expensive resources (such as people) are waiting for them to complete, or if they must be finished before real-world deadlines. The framework can generalize to value functions that decay at variable rates, but these complicate the problem significantly.

A scheduler is driven by a sequence of task arrival times (*release times*) $[arrive_0, ..., arrive_i]$ and task completion/departures. It maintains a queue of tasks awaiting dispatch, and selects from them according to some scheduling heuristic. Once the system starts a task, it runs to completion unless preemption is enabled and a higher-priority task arrives to preempt it. RPT_i represents task i's

expected Remaining Processing Time—initially its $runtime_i$.

Two common scheduling algorithms are First Come First Served (FCFS), which orders tasks by $arrival_i$, and Shortest Remaining Processing Time (SRPT), which orders by RPT_i. We consider the value-based scheduling problem under the following simplifying assumptions:

- The processors or machines for each task service manager are interchangeable. Preemption is enabled; a suspended task may be resumed on any other processor. Context switch times are negligible.
- The task service never schedules a task with less than its full resource request—tasks are always gang-scheduled using common backfilling algorithms with the requested number of processors. For simplicity we assume that the resource request is a single processor or machine.
- The predicted run times $runtime_i$ are accurate. There is no interference among running tasks due to contention on the network, memory, or storage. We do not consider I/O costs [26] or exceedance penalties for underestimated runtimes. Users may leverage recent work on active learning to learn correct estimates for task runtimes on a given set of resources [151].

A candidate schedule determines the expected next start time $start_i$ and completion time for each task i. The completion time is given as $start_i + RPT_i$ assuming the task is not preempted. Thus, the expected value $yield_i$ for $task_i$ in a candidate schedule is given by Equation 6.1 and the delay for the expected completion time:

$$delay_i = start_i + RPT_i - (arrival_i + runtime_i) \tag{6.2}$$

Scheduling based on linearly decaying value functions is related to well-known scheduling problems. Total Weighted Tardiness (TWT) seeks to minimize $\sum_i d_i T_i$ where d_i is the weight (or decay) and T_i is the tardiness of taks i. A task i is *tardy* if it finishes after a specified deadline. We focus on the variant in which each task's deadline is equal to its minimum run time, so that any delay incurs some cost. If penalties are unbounded, then this problem reduces to Total Weighted Completion Time (TWCT), which seeks to minimize $\sum_i d_i C_i$ where d_i is a task's weight (equivalent to decay) and C_i is its completion time in the schedule [18].

The offline instances of TWT and TWCT are both NP-hard. We use online heuristics for value-based scheduling [33, 40] in a computational economy: the best known heuristic for TWCT is Shortest Weighted Processing Time (SWPT). SWPT prioritizes tasks according to the task's d_j/RPT_j, and is optimal for TWCT if all tasks arrive at the same time. Our task service schedules tasks based on SWPT and uses the measures of task utility to formulate bids to a resource provider (or broker).

Previous studies give no guidance on how users value their jobs, since no traces from deployed user-centric batch scheduling systems are available. We use the workloads from [44] where value assignments are normally distributed within *high* and *low* classes: unless otherwise specified, 20% of jobs have a high $value_i/runtime_i$ and 80% have a low $value_i/runtime_i$. The ratio of the means for high-value and low-value job classes is the *value skew ratio*. However, we note that the specification of the workload does not have a significant impact on the results.

6.3.2 Effect on Global Utility

The previous section outlines a market-based task service that associates a utility function with each task: these task services execute tasks on a set of available machines. We now consider a scenario with two competing task services bid for machines using self-recharging virtual currency. The bidding strategy for each task service is workload-specific. The first task service, denoted as *Steady*, executes a steady stream of low-value tasks, while the second task service, denoted as *Burst*, executes intermittent bursts of high-value tasks. We choose the two workloads to demonstrate the capability of self-recharging currency to achieve a higher utility than proportional-share by shifting resources to the *Burst* service only when it requires them. The name of each task service implies both its workload and its bidding strategy.

The intuition behind the experiments is rooted in the fact that with low recharge times there is no incentive for a service to withhold a bid, even if it does not have any load; larger recharge times provide the incentive to withhold bids when a service has no load since currency acts as a store of value. This results in higher global utility if two services are able to plan their bids over time and they have different resource needs at different times, as is the case with *Steady* and *Burst*.

Defining an Auction

A market requires an auction protocol to decide the allocation of resources based on the received bids. We use an auction that clears bids by allocating a proportion of the total machines at a site to each service manager according to the proportion of each service managers currency bid relative to the total submitted currency bid. Thus, if the two services bid currency in demoninations c_{steady} and c_{burst} for a single auction, then *Steady* receives $machines_{total} * c_{steady}/(c_{steady} + c_{burst})$ and *Burst* receives $machines_{total} * c_{burst}/(c_{steady} + c_{burst})$. Auctions in our experiments occur every time period a_t, and each auction reallocates every available machine in the pool. Our simulated site allocates resource leases of length a_t.

The auction allocates multiple machine units, but does not allocate bids for slivers (*e.g.*, CPU, memory, or I/O bandwidth). The auction is similar to Lottery Scheduling, except that currency is taken away for the duration of the recharge time and not immediately returned to the task service.

Defining Bidding Strategies

The bidding strategies of *Steady* and *Burst* are static—neither strategy adjusts bids dynamically, as is the case in a traditional market. The bidding strategy for *Steady* evenly splits its currency across every auction within the recharge time window. We dictate that auctions occur at fixed times evenly spaced within the recharge time window. In our experiments, auctions occur every 100 simulated minutes and the recharge time varies as a fixed multiple of 100 minutes. The bidding strategy is appropriate since *Steady* plans *a priori* for a steady stream of low-value jobs that does not require changes to the bids.

The bidding strategy for *Burst* evenly splits its currency across a small window of auctions whenever it has queued tasks. The *Burst* service does not bid when there are no queued tasks. In contrast to *Steady*, *Burst*'s bidding strategy assumes that any queued tasks are part of a fixed-size burst of high-value task submissions. The bidding strategy for *Burst* is appropriate if it can obtain enough machines over its small window of bids to satisfy the burst. However, if it cannot attain enough machines, then the self-recharging currency model dictates that it must wait at least one recharge time interval to bid for additional machines to finish its remaining tasks. If the *Burst* service does not receive enough machines then it either planned its bids poorly (*i.e.*, spread them

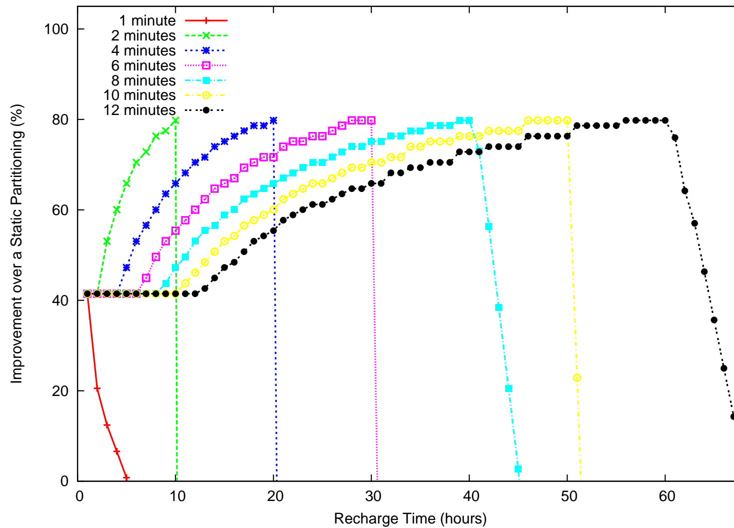

Figure 6.4: Two market-based task services, called *Burst* and *Steady*, bid for resources to satisfy their load. As the recharge time increases, the global utility increases until the point at which the task services' static bidding strategies are too aggressive: they run out of currency before they finish executing their task load. The result is the global utility falls below that of a static partitioning of resources. The bidding window for *Burst* (each line) is the expected length of time to complete a *Burst* of tasks, according to its static bidding strategy. As the bidding strategy becomes more conservative (*i.e.*, lower numbers) relative to the recharge time the allocation achieves a higher global utility despite longer recharge times.

out over too many windows) or it does not have enough currency to purchase enough machines to satisfy the burst.

As exemplified by the bidding strategies of *Steady* and *Burst*, all market-based services must plan both their resource usage and bids over time in accordance with their load to ensure an efficient allocation of resources. A benefit of self-recharging currency is that it clearly defines the planning horizon for service managers—before making a bid a service manager need only account for the impact a bid has on its currency holdings over a single recharge interval. After the recharge interval the service manager is free to use the currency again, and past decisions have no residual impact. In our experiments, we focus on the effect on global utility as the recharge time changes relative to the static bidding strategies of *Steady* and *Burst*. These changes in the recharge time impact global utility because they affect the efficacy of each service's static bidding strategies.

Experimental Setup

Figure 6.4 shows the effect on the global utility as a function of the recharge time as the two services compete for resources in a simulated market setting.

Bidding Window	*Currency*
1 auction	500
2 auctions	250
4 auctions	125
6 auctions	83
8 auctions	63
10 auctions	50
12 auctions	42

Table 6.1: The bidding window is the number of auctions *Burst* bids when it has queued tasks. The bidding window from Figure 6.4 dictates the amount of currency bid for each auction. The table above shows the amount of currency *Burst* bids for each bidding window. The large the window the lower the amount of currency bid, since its total currency holdings are split across more bids.

Steady's workload consists of a task stream of 2000 tasks with exponentially distributed inter-arrival times of 1 simulated minute and exponentially distributed durations of 100 minutes. Each task has a value function as described in Section 6.3.1 with a normally distributed decay rate with a mean of 2 and a standard deviation of 1; both the value skew and the decay skew are 1.

Burst's workload consists of 10 fixed size batches of tasks that arrive every 1000 simulated minutes with 100 tasks of duration 100 minutes in each batch. Each task has a value function with a normally distributed decay rate with a mean of 20 and a standard deviation of 1; both the value skew and the decay skew are 1.

Thus, the value of executing a task in the *Burst* workload has 10 times the value as executing a task in the *Steady* workload. The total number of available machines in the infrastructure is 100 and each service manager has 500 credits. To compute global utility we calculate the sum of the yield, using the equations from Section 6.3.1, of all tasks completed by both services. Auctions are held every 1 minute ($a_t = 1$).

Each line of Figure 6.4 represents a different *bidding window* for the *Burst* service. The bidding window is the expected length of time for *Burst* to satisfy a burst of high value tasks. Note that the bidding window does not represent the actual time it takes to satisfy the burst of high-value tasks. If the *Burst* service has any queued tasks, it bids its currency holdings divided by the bidding window, evenly splitting its currency across all bids during the expected time period of the burst. The y-axis represents the improvement in global utility over a static partitioning of 50 machines to each service and the x-axis is the recharge time. Table 6.1 shows the amount of currency *Burst* bids for each auction in the bidding window.

Conclusions

We draw multiple conclusions from the experiment. First, we observe that a recharge time of 1 minute, which is equivalent to proportional-share scheduling, achieves a higher global utility than statically partitioning resources between the two task services. The result is expected since the services do not bid if they have no queued tasks; in this case, they leave resources for the service that has the dominant workload. In a true market, with a recharge time equal to the auction period a_t, there is no incentive for a service to withhold a bid, even if it does not have any load; larger recharge times provide the incentive to withhold bids when a service has no load since currency acts as a store of value.

Second, for any given bidding window of *Burst* we observe that as the recharge time increases the global utility to a maximum point after which it experiences a steep drop. The increase in utility demonstrates a key property of the recharge interval—with short recharge times the *Steady* service is able to allocate more currency to bid for each auction, reducing the machines allocated to the *Burst* service during its task bursts. As the recharge time increases, the *Steady* service must decrease the currency per bid per auction to ensure that its currency is evenly split across all the auctions in the recharge interval. In contrast, the *Burst* service's bids remain constant since the burst size does not change with the recharge time. As a result, the *Burst* service receives an increasingly larger share of the machines to satisfy high-value tasks, resulting in a steady increase in global utility.

The *Burst* service takes at least 2 auction periods (*i.e.*, two minutes), given its workload, to satisfy a task burst if allocated all of the available machines in the pool. When the recharge time exceeds the length of a burst the assumptions built into the bidding strategy no longer hold: there is more than one burst per recharge interval so the service must save currency. Poor planning outside the range of the recharge interval in the bidding strategy has a significant impact on global utility. We see the impact of poor planning in Figure 6.4 as the global utility decreases drastically. *Burst* does not have enough currency to satisfy subsequent bursts of tasks, which occur before the recharge time, because it spends all of its currency on the first burst.

For example, if *Burst* extends its bidding window to 2 minutes, then the *Burst* service bidding strategy expects the currency to recharge before the next arrival of tasks, which occurs every 10 auction periods (or minutes). As a result, once the recharge interval exceeds 10 minutes the global

utility drops below the global utility of a static partitioning, since the *Burst* service has no currency to bid after the second burst of tasks.

If the bidding window is 4 minutes, then the *Burst* service expects the currency to recharge after 2 bursts finish since the service saves currency after finishing its first task burst in 2 minutes to process another burst. Spreading bids across 4 auctions reduces the value of each bid. As a result, the larger bidding window achieves less global utility than a bidding window of 2 minutes because the service gains less machines, due to its lower bid, to complete its task burst. However, a bidding window of 4 minutes allows the service to continually increase utility until the recharge time is 20 minutes (as opposed to 10 minutes) since the it is able to save enough currency to process 2 bursts.

The experiment demonstrates that the ability to plan usage (and bids) over time is an important part of a successful market. Poor planning relative to the recharge time can result in global utility that is worse than a static allocation. As the recharge time increases services must carefully budget their currency to account for load increases that may occur before a recharge. The recharge interval defines the period that a service must plan its usage: a short recharge time is similar to proportional-share scheduling which imposes less risk from the consequences of starvation and poor planning, while a long recharge time resembles a true market economy with all of the inherent risks of planning incorrectly.

Infrastructures will continue to require short recharge times as long as the policy questions for markets remain unanswered. We show here that increasing the recharge time beyond proportional-share improves global utility if service managers plan their usage over the recharge time. We conclude that a lease-based currency model improves global utility beyond proportional-share allocation and may incorporate advances in market policies as they develop by adjusting the recharge time.

6.4 Summary

Markets define incentives for actors to contribute resources to a networked operating system and provide a means to arbitrate resource usage. This chapter outlines a new form of currency, called self-recharging virtual currency, as a lease-based resource arbitration primitive for networked resource management. We demonstrate one way to use the primitive to define a market for task scheduling as one example market.

The recharge time of the currency is a configurable parameter that allows a market to set the appropriate window for consumers to plan their usage: a short recharge time resembles proportional-share scheduling where bidders require little planning, while a long recharge time resembles a true market economy where users must carefully plan their usage over time to satisfy both present and future demands. We show that self-recharging currency can increase global utility relative to a proportional share economy if bidders plan effectively over the recharge time; if bidders do not plan effectively the global utility may be significantly less than a static allocation of resources.

Chapter 7

Leases and Fault Tolerance

"*Our greatest glory is not in never falling, but in rising every time we fall.*"

Confucius

This chapter examines a methodology for addressing faults and failures in a networked operating system that leverages the lease's role in building reliable distributed systems [107]. We focus on failures of service managers, brokers, and authorities and the network that connects them. Failure recovery for guests and resources requires guest-specific and resource-specific detection, diagnosis, and repair policies. The architecture also enables service managers and authorities to absorb guest and resource failures by requesting additional resources; however, specific policies for detection, diagnosis, repair, and replacement are outside the scope of this thesis.

Addressing actor failures is particularly important due of the scope of the architecture: correct operation of networked hardware affects any guest that uses the hardware. For example, guests that coordinate emergency services, such as Internet 911, must ensure sustained operation in the presence of network failures. Resilience to failures is difficult, in part, because the architecture distributes control of the hardware across loosely coupled actors that communicate to control allocation. These actors and the network that connects them, may periodically experience transient outages. For example, power outages may cause one actor server to fail, while other actors continue operation, a network failure may cause two disconnected actors to continue operation despite the lack of a viable communication channel, or authorities and service managers may become disconnected from the resources and guests they manage. We use the term *disconnection* to describe the transient outages each actor experiences throughout the lifetime of the system.

7.1 Approach

With respect to actor failures, the goal of a networked operating system is to simply make forward progress, despite transient disconnections due to actor and network failures. The network of

independent actors that comprise the system must remain in a consistent state despite transient disconnections, and must be able to reincorporate previously disconnected actors, resources, and guests back into the collective system. The result is sustained operation despite one or more failures in the system.

Individual actors may use conventional distributed systems techniques, such as replicated state machines [149] or the primary-backup approach [31], to replicate themselves to ensure high availability in case of a crash. To ensure sustained operation, each actor in a networked operating system must assume that a peer actor may become disconnected at some time due to conditions outside of its control (*e.g.*, wide spread network outage). Additionally, since actors are independent they cannot rely on peers to conform their behavior for the benefit of other actors or the system. Each individual actor must rely on the leasing protocols and its own local behavior to ensure a consistent global system state.

Each actor operates independently of the other actors and is self-interested. We assume that accountability is sufficient to prevent fraudulent behavior of an actor. A full treatment of accountability as a mechanism to prevent fraudulent behavior or detect byzantine or strategic behavior is outside the scope of this thesis. The actor failures described in this chapter assume that actor accountability is sufficient to prevent actors from not following the leasing protocols. As a result, we assume that actors are fail-stop.

Actors may experience arbitrary delays in communicating with peer actors or processing requests, but we assume that sent messages are eventually delivered unless the sender fails before delivery or the receiver fails permanently. In this chapter, we use concrete examples from the COD authority in our discussion of failures: authorities use virtual machine monitors to instantiate virtual machines for service managers, and service managers instantiate guests on top of these virtual machines. The techniques are applicable to a range of resources and guests, as described in Section 7.4.3.

7.1.1 Overview

The lifetime management property of leases ensures that a permanently failed actor releases any remote actor state bound to the lease. In this chapter, we discuss how to ensure that an actor can recover from a failure and reincorporate itself back into the collective system. We achieve this

property by defining a self-synchronizing lease state machine that commits local actor state changes to persistent storage and ensures that all state transitions are idempotent.

These concepts extend to any asynchronous distributed system composed of independent actors whose loose coupling complicates coordinating updates to shared state. In this model actors issue requests to peers (actors, guests, or resources) to execute operations that change remote state, but are free to control and manipulate their own local state. These concepts apply to systems that share networked resources because they involve independent actor servers, resources, and guests. We note that distributed system designers in practice have proposed changing the organization of their systems to adhere to a similar design pattern, as indicated in [88]. However, our work is motivated not by infinite scalability, but by a loose coupling of actors.

Networked operating system actors represent an important example of this type of distributed system, which comprises a group of servers that request pair-wise state changes and operation execution. For example, service managers request state changes by a broker (*e.g.*, issue a ticket), state changes by an authority (*e.g.*, configure a resource), and state changes on each resource (*e.g.*, execute guest software) where the remote actor executes one or more operations to complete the state change. Likewise, an authority that requests state changes on each resource (*e.g.*, setup the resource) and state changes by the service manager (*e.g.*, notification of a lease) follows the same model.

7.1.2 Interacting State Machines

Actors (service managers, brokers, and authorities) form a distributed system that consists of multiple servers that communicate over an unreliable, wide-area network to negotiate the acquisition of resource leases for guests. Service managers and authorities may behave in a volatile wide-area network or a more robust local-area network. For example, PlanetLab Central acts much like a single authority for resources it controls at numerous locations globally. Actors model each lease as a state machine that interacts with peer actor state machines. Each actor communicates with its peers to advance the state of each lease with actor communication occuring on a per-lease basis. The different actors in the system implement distinct, asynchronous lease state machines to guide lease state changes. We describe each actor's lease state machine below.

Figure 7.1 illustrates typical state transitions for a resource lease through time. The state for

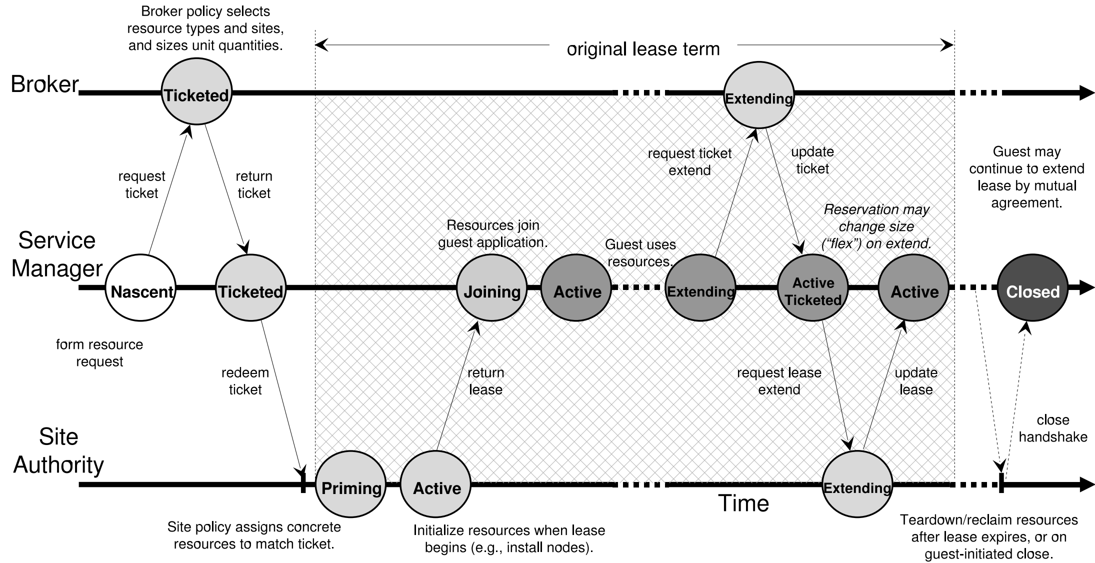

Figure 7.1: Interacting lease state machines across three actors. A lease progresses through an ordered sequence of states until it is active; delays imposed by policy modules or the latencies to configure resources may limit the state machine's rate of progress. Failures lead to retries or to error 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.

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 lease abstraction on each actor. 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 policy modules or handler executions and/or generate a message to another actor. All lease state transitions trigger a commit of modified lease state and local properties to an external repository, as described in Section 7.4.2. The service manager lease state machine is the most complex because the 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.

7.2 Actor Disconnections

Actors request lease state changes from peer actors to guide the allocation of resources and the instantiation of guests. Actors may unexpectedly fail or go offline for a variety of reasons including power outages, software bugs, or scheduled server upgrades. To ensure sustained operation, actors must be able to come back online, recover their previous state, and continue normal operation without interrupting the state of the rest of the system. We discuss the challenges associated with actor disconnections from other actors and from the guests and resources they control, and the approaches we use to address these challenges.

A disconnected actor cannot communicate with peer actors or with the guests or resources under its control. Thus, disconnections can cause an actor to become out of sync with the rest of the system. For example, the state of a disconnected actor's peers and the resources under its control may change during the period of disconnection: a lease may expire or a pending action on a resource may complete. An actor disconnection may prevent the actor from receiving lease state change notifications from its peers.

Disconnections are often transient—actors restore communication at some time in the future, either by re-starting after a failure or by correcting a network partition. If a disconnection is not transient and an actor does not restore communication then the lifetime management property of leases dictates that all leases eventually expire and transition other actor's state to the pre-lease state without the need for direct communication. Systems frequently use leases for garbage collection of state when disconnections occur [28, 68, 167, 179]. We now describe how the architecture addresses transient disconnections.

7.2.1 Actor Stoppage and Network Partitions

There are two primary reasons for an actor to become disconnected from the rest of the system: the actor server goes offline or the network becomes partitioned. Actors address network partitions at the RPC layer; message exchanges with peer actors, remote resources, or remote guests may be periodically retried until an actor restores network communication or a lease expires. As described in the previous section, we strive to keep the actor state machine simple: it does not include states to represent failed cross-actor, cross-resource, and cross-guest communication. The lifetime

management property of leases eliminates the need for setting arbitrary RPC-level timeouts that may depend on a particular resource or the network location of a peer actor.

We focus on disconnections due to actor stoppages: there are two phases required to recover an offline actor as described below.

- In phase one, which we term *local recovery*, the actor restores its local state to its last state using information from a persistent store, such as a database.
- In phase two, called *lease synchronization*, the actor independently synchronizes the local state of each lease, which may be stale due to the passage of time or unknown external events, with the current state of the lease; lease state changes may occur at remote resources, guests, or peer actors.

7.2.2 Local State Recovery

Phase one involves restoring the pre-failure state of a failed actor server. As described in Chapter 4, each actor commits local lease state to a persistent store, such as a database, using opaque property lists. Actors store sufficient lease state to recover from a failure. Actors cache state in memory during normal operations in order to process requests (*e.g.* execute arbitration, assignment, and provisioning policies). The cached lease state is written to the persistent store to ensure that actors can regenerate all cached, in-memory state after a failure. However, despite caching, it is important for actors to be able to regenerate all state not present in the persistent store whenever a failure occurs.

We use the term *local recovery* to describe the process of rebuilding an actor's cached, in-memory state from the persistent store. Local recovery includes the process of regenerating state not stored at the time of failure. Most other objects stored in the actor database rarely (if ever) change. Deciding what objects to write to the actor database, and when, is an important part of local recovery and lease synchronization.

Many of the stored objects are static and include information about each actor such as service manager slices, site resource inventory, user information, and peer actors. Each lease is the primary dynamic object stored by each actor that changes state over time. Each actor may manage hundreds or more simultaneous leases, including leases that interact with multiple distinct service managers,

authorities, and brokers. As described in the previous section, all actor interactions are based on an asynchronous, distributed lease state machine that advances each lease through a series of distinct states on three actor servers.

The local state of a lease dictates the action taken by an actor when an event occurs for that lease. For instance, lease events may include an incoming lease message from a peer that requires an actor to take an action (*e.g.*, instantiate resources), or trigger a timing-related event (*e.g.*, a lease end or lease extension). It is crucial that each lease be in the correct state when a lease event occurs; the correct operation of the distributed lease state machine depends on it. As a result, every time the state of a lease changes an actor writes the updated lease to a database to enable recovery if the actor fails. Lease updates to a local database are transactional.

7.3 Lease Synchronization

Storing and updating lease information in a local database ensures that an actor can recover its last committed local state. After recovery, the actor contains the same state as it did at the time of failure. However, local recovery is not sufficient for resuming operation in the wider system since each lease's state at peer actors may change while an actor is inoperable. As a result, we require the *lease synchronization* phase of actor recovery.

For instance, while an actor server is inoperable time may pass, causing the server to miss timing-related events. While an inoperable actor server cannot trigger timing events, a peer actor's lease state machine may alter the lease's state. Actors do not stop when a peer actor fails—they remain operational and may trigger lease events. However, the lease state machine ensures that an inoperable actor halts the progression of the lease state until the lease expires. This property of the lease state machine is beneficial since it bounds the number of transitions an inoperable actor misses. To a peer actor, an inoperable remote actor appears the same as a network partition: communication is impossible and lease state transitions which require an action by a failed actor will remain blocked within the RPC layer until the lease expires.

Likewise, during an actor server failure the resources under the control of the actor may change state. An action issued by an actor on a resource before a failure may complete or may, itself, fail while the actor is inoperable. The state of each resource under an actor's control changes

independently of the actor.

The recovery architecture of actor servers takes a multi-pronged approach to lease synchronization, as outlined below.

- Actors log all lease state transitions before performing any actions associated with the lease transition. Logging enables recovery of lease state for a failed actor. Actors perform write-ahead logging to ensure that transitions are not lost in a failure.
- Lease states may indicate a pending lease state transition (*e.g.*, a remote, asynchronous action issued to a resource or peer). We design the recovery model so all pending lease requests are reissued on recovery. Actors must reissue requests since the write-ahead logging to the local database is not transactional with respect to remote state change. As described in Section 7.3.2, the design ensures reissued requests are idempotent. Alternative models that use distributed transactions require shared infrastructure, such as a transaction coordinator, that a loose coupling of actors cannot rely on.
- The lease state machine of peer actors and remote resources must gracefully accept duplicate requests correctly. From the perspective of a failed actor the requests must be idempotent so it may issue multiple requests without affecting the end result.

All actor message requests must be idempotent to ensure that duplicate requests are suppressed and do not affect the operation of the system. Actors wait for peers to accept and log all actions. If an actor fails or becomes disconnected before receiving a lease update, indicating a successful completion, then the actor retransmits the request on recovery. Lease updates from clients in response to requests are asynchronous—actors may issue lease updates without explicit requests. As with NFS [95], before responding to an actor a peer must commit an update to stable storage indicating the successful completion of an actor's request. NFS servers implement idempotency of peer requests to ensure statelessness. Unlike NFS, however, all actors [1] store important state about their lease holdings that must be persistent and recoverable. As a result, any actor server can recover its lease holdings after a failure.

[1] Unlike in NFS, each actor acts as both client and server.

Actor servers must apply idempotency in two different ways—multiple interacting actors may issue duplicate lease messages and a single actor may issue duplicate resource actions. In both cases, the peer actor or resource suppresses the duplicate request. Phase two of recovery focuses on using idempotency as the tool to synchronize the local state with the current state of each lease held by a failed actor. Next we discuss the two parts of lease synchronization: synchronizing with the state of peer actor servers and synchronizing with the state of a resource or guest.

7.3.1 Actor Recovery

To accomplish actor recovery we ensure that actors whose leases fall out of sync may catch up by retransmitting requests once they come back online. To synchronize the state of each lease an actor simply recovers its local state, and then examines the state of each lease and retransmits any operations that were in-progress at the time of the failure. As with ensuring other forms of idempotency, cross-actor message exchanges may contain sequence numbers that allow peers to suppress duplicate messages.

The design mitigates the impact of actors that fail to execute the leasing protocols to individual leases; thus, a divergence of one actor only affects the leases co-held by its peers. Misbehaving actors must also operate within the context of the lease state machine—peer actors do not recognize any action that occurs outside the normal cycle of states. The design restricts the effect of divergent peers to individual leases. Actors that experience state changes outside of the normal lease state machine are free to stop processing a lease and engaging in exchanges with peer actors.

7.3.2 Guest/Resource Recovery

Self-synchronizing state machines combined with per-lease, actor message exchanges address the problem of actors becoming disconnected from each other; however, actors also use messages to interact with guests and resources over the network, and actors may become disconnected from these resources. In this case, lease state not only includes persistent state stored on disk, but also the operational state of a resource or guest.

One aspect central to service manager and authority design is their physical separation from the guests and resources they manage. Authorities control resources remotely, such as Xen's control

domain, and service managers control the guest in a symmetric fashion by controlling software on remote resources, such as software executing on a Xen virtual machine. Brokers simply mediate the allocation of resources at the authorities to the service managers and do not remotely communicate with any resource or guest. As a result, we focus only on service manager actors and authority actors in this section.

When recovering locally actors must determine the state of resources under their control. For instance, at the time of failure or disconnection an actor may have outstanding operations issued for a machine. An authority, which may manage thousands of machines, may be creating hundreds of virtual machines when a failure occurs, or a service manager may be deploying a guest on virtual machines spread across different authorities.

Determining the outcome of these actions is critical to recovery. Interactions with guests and resources occur through the guest and resource handler interfaces. The guest handler provides the *join*, *modify*, and *leave* methods. Likewise, a authority contains a handler for each type of resource. A authority's handler includes the *setup*, *modify*, and *teardown* methods. As described in Chapter 4, these methods shield the complexity of the lease abstraction on both actors from the specifics of a particular guest or resource.

The Toggle Principle

The basic principle of our approach to addressing disconnections from guests and resources is that the type-specific resource handlers should follow the *toggle principle*—all resource handler methods are idempotent. While guest and resource recovery differs from actor recovery, the toggle principle follows the principles of lease synchronization: all cross-network message exchanges are idempotent. In an authority, the lease abstraction upcalls the resource handler to *setup* and *teardown* each resource unit at the start and end of a lease. In the service manager, the lease abstraction upcalls the guest handler to attach/detach leased resources to and from the guest. From the perspective of either actor, on recovery each resource is in either of two basic states: on or off. Both handlers hide intermediate states from the lease abstraction since revealing intermediate states would only serve to complicate the lease state machine.

Although handlers execute asynchronously from the lease abstraction, the lease abstraction does

invoke handlers for each resource unit in a serial order. Serial ordering ensures that all guest and resource handlers reflect the final state of the last action issued by the lease abstraction, independent of any intermediate states, incomplete operations, or transient failures. Thus, on recovery, actors reissue pending handler actions to force resources held by each lease into a particular state (*e.g.*, on or off). Since handlers are deterministic and serial, then it is sufficient for actions to be logically serialized and idempotent at the handler level.

The property may require persistent state in the handler (*e.g.*, as in package managers, which typically adhere to the toggle principle); at minimum, it requires a persistent and consistent name space for any objects created by the handler (*e.g.*, cloned storage luns or resource partitions). Handler suppression of redundant/duplicate operations triggered by an authority's lease abstraction also avoids disrupting guests that may not have detected an authority failure that was independent of its resources. To ensure successful recovery of lease groups, as discussed in Chapter 4, the lease abstraction respects precedence orderings in reissuing the handler methods on recovery.

Handlers are responsible for their own state management. Some handlers may self-recover or self-repair after a failure without the lease abstraction having to probe and restart them. If this occurs then the toggle principle dictates that the handler must ignore the probing. However, we view handler self-recovery as an optimization where the authority state, not the handler, is authoritative. A handler does not need to notify an authority when actions complete (*e.g.*, a virtual machines comes up), although, as an optimization a handler may store and use arbitrary caches to speed recovery. We summarize the key points of the design below.

- Guest and resource handlers have two methods: turn on (*join* or *setup*) and turn off (*leave* or *teardown*). These methods are inverses. From the perspective of the lease abstraction, the resource has two configuration states: up or down, on or off, depending on which method completed last. Any intermediate states are hidden from the lease abstraction.

- The lease abstraction invokes handler methods in some serial order. The final resource state reflects the last action in the sequence, even if the actions execute asynchronously. Note that a request may arrive while an action is in progress: it may cancel the action in progress or wait for it to complete, but it must preserve the serial ordering.

- Correct execution of an action must not depend on the initial state of the resource. For

example, teardown should complete even if the setup was incomplete when the teardown request arrived. In particular, handler methods are idempotent. If the handler retains state, it should record requests and suppress duplicates. The handler must preserve idempotence even across restarts; this is trivial if all of its state is transient.

Allocated resources (*e.g.*, virtual machines) must have unique names to ensure idempotence. We discuss naming further in Chapter 5. An authority knows, and controls, these names. Actors save the state required to invoke each handler and the outcome of previously executed handler invocations. Given these principles, recovery is relatively simple from each actor's perspective. If the actor fails, it recovers the list of leases and their states, and reissues any pending actions (*i.e.*, a "redo" recovery model). The actor then periodically checks the status of leased resources by probing them. The probe handlers may use resource-specific failure detectors to derive the state of the resource. If the lease abstraction notices that a resource has failed it may notify a peer actor.

Impact of a Disconnection

There is a beneficial side effect of the physical separation of an actor from its resources: an actor failure does not immediately affect the guest or resources under its control. In the case of a service manager failure, its resource leases will expire unless the service manager renews them. The guest under the service managers control will slowly and ungracefully lose resources until none are left since the service manager cannot execute the guest handler at lease termination.

In the case of an authority, a failure will prevent any change in the resource allocations at the site. Resources allocated to service managers at the time of the failure will remain allocated and authorities will not satisfy broker allocations made to service managers after the failure. Service managers will be unable to alter their resource allotments at the time of the failure. Broker failures will prevent service managers from requesting new resource leases or extending existing leases from a failed broker. As with a service manager failure, the guest under each service manager's control will slowly, but gracefully, lose resources until none are left since the guest *leave* handler will execute as leases expire.

Table 7.1 lists the effect of each actor's failure from a guest's point of view. Clearly, actor failures can result in problems due to service interruption, especially when leases are short compared with

Actor Failure	*Effect*
Service Manager	The guest loses resources ungracefully since the service manager failure prevents execution of the *leave* handlers. The service manager failure also prevents requests for new leases or requests to extend existing leases. The site authority unilaterally tears down resources as leases end.
Broker	The guest gracefully loses resources since the broker cannot satisfy service manager requests for new leases or requests to extend existing leases. The service manager is able to execute the *leave* handler as leases expire.
Site Authority	The service manager's resource allocation at the time of the failure does not change. The service manager may request new tickets or extend existing leases from the broker, but the site cannot satisfy new requests.

Table 7.1: A table outlining the the effect of a single actor failure on a guest in a small network of actors that includes a single service manager, broker, and site authority.

the length of actor downtime. However, for infrastructures that allocate virtual and/or physical machines we envision service disruption time (*e.g.*, seconds to minutes) to be significantly shorter than lease lengths (*e.g.*, hours to days to weeks). Additionally, policies for addressing the end of leases can mitigate the severity of disconnections due to failure. For instance, an authority may choose to not teardown a resource at the end of a lease if there is no resource constraint, allowing a service manager or a broker, time to recover and resume normal operation.

A broker failure may also prevent a service manager from renewing a ticket, in which case, an authority must grant the service manager time to wait for the broker to recover. If a broker fails and a service manager *would have been* granted a ticket extension then it is certain that the authority will not be resource constrained. A failed authority will not teardown any resources, allowing a service manager to continue to renew a lease with a broker until the failed authority resumes operation and can accept the ticket extensions.

The distributed nature of the system mitigates the impact of any single actor server failure. A service manager may lease resources from multiple brokers that hold resources from multiple authorities. In the face of an actor server failure or a resource failure a service manager may simply acquire resources from another broker or authority. A key part of delegating resource control to guests is the capability to adapt to failures of actors, edge resources, and the network.

7.4 Implementation

We have implemented the lease state machine and persistence architecture in our prototype. Below we describe details of the implementation relating to recovering lease and policy state and the implementation of idempotency for different resource handlers we use.

7.4.1 Lease State Machine

The lease abstraction must accommodate long-running asynchronous operations for each lease. 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. A key design choice is to structure the lease abstraction on each actor as a non-blocking event-based state machine, rather than representing the state of pending lease operations on the stacks of threads (*e.g.*, blocked in RPC calls). Each *pending* state represents any pending action until a completion event triggers a state transition. Each of the three actor roles has a separate lease state machine that interacts with other actor state machines using RPC.

The broker and authority state machines are independent—they only interact when the authority initially delegates resource rights to the broker. Keeping the state machines simple is an explicit design goal since the actor state machines form the core of the architecture, and must apply to multiple resources (at an authority) and guests (at a service manager). For example, the state machine does not include states for overlapping transitions caused by timing events that fire before the completion of an asynchronous action; in these cases, we push the complexity to handle overlapping or canceled actions to the guest and resource handlers.

The concurrency architecture promotes a clean separation of the lease abstraction from guest-specific and resource-specific configuration. The guest and resource handlers—*setup, modify, and teardown* and *join, modify, and leave*—as well as status probes—do not hold locks on the state machines or update lease state directly. This constraint leaves them free to manage their own concurrency (*e.g.*, by using blocking threads internally). For example, the COD resource handlers start a thread to execute a designated target in an Ant script, as described in the next chapter. 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.

Timer events trigger some lease state transitions, 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 must 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: a broker may reject service manager requests for leases at a given start time 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, and the authority may unilaterally initiate *teardown* whether the guest is ready or not.

7.4.2 Persistence and Recovery

Each actor uses a LDAP or MySQL database to persist lease state. Each in-memory data object that requires persistence implements a `serializable` interface that includes three methods: `save`, `reset`, and `revisit`. The `save` method serializes the hard state of an object. Hard state is meta-data that an actor cannot reconstruct from other stored meta-data. The `reset` method takes a serialized object, unserializes it, and sets the appropriate object fields. Actors use the revisit method to regenerate soft state from hard state. As described in the previous chapter, lease objects commit lease state to a persistent store on every state transition. Lease's use the `save` method to serialize their state to a local property list. The `reset` method obtains these stored local lease properties from the persistent store on recovery. To enable reconstitution of objects from property lists, we associate each field of an object with a property name, stored as a string.

The actor recovery process starts by fetching objects from the persistent store and recursively executing the `reset` method on each object. Resetting an actor recursively resets the hard data of objects that comprise each actor including policy implementations, slices, and leases. The process initially invokes the `reset` method for each set of slice properties read from the database, and recursively invokes the `reset` method for each lease in each slice. The slice and lease `reset` methods invoke the `reset` method for any serialized object stored within a slice or lease. Once all objects have been reset the recovery process executes the `revisit` method on all data structures recursively to rebuild any soft state from the reset hard state. After the reconstruction of all slice and lease objects, the recovery process must revisit each policy implementation to ensure that the policies internal calendar state is rebuilt. Once this process completes the actor clock begins to tick. Idempotent state transitions ensure that any state transitions in-progress at the time of failure do not fail when repeated.

Recent work uses a similar data model for transforming the Condor batch scheduler into a more manageable data-centric implementation [143]. The work combines EJBs with a database-backend to provide container-managed persistence and a single point for querying and maintaining a distributed set of Condor machines. We chose to implement our own persistence model because of our explicit support for idempotent actions in the recovery model. Container-managed persistence using EJBs rebuilds hard state, but does not ensure distributed object (*e.g.*, lease) synchronization. Actor implementations differ from container-managed persistence of distributed web services since they do not rely on distributed transaction protocols (*e.g.*, variations of a two-phase commit) as standardized by WS-AtomicTransactions. Web services focus on distributed applications that must present a coherent view of shared state to a third-party (*e.g.*, transferring money between two bank accounts must be atomic). In contrast, a collection of actors does not present a coherent view of shared state to a third-party: each actor is independent and manages its own state.

7.4.3 Idempotent Handlers

We implement idempotent methods for all resource handlers (*e.g.*, the setup, modify, and teardown methods) and their associated resource drivers which instantiates and controls Xen virtual machines, iSCSI disks, LVM disks, file-backed disks, and NFS file volumes, as described in Chapter 8. In each of these cases we leverage state stored within the resource to ensure idempotency. When an authority

creates a Xen virtual machine the hypervisor stores state about the machine, including its name, in memory. Before instantiating any virtual machine we ensure that the name does not already exist: the existence of the name indicates a duplicate request for a state change. Likewise, for iSCSI, LVM, file-backed, and NFS volumes the name of the volume is stored in memory by the resource itself. Using state stored within the resource allows the resource handler and drivers to remain stateless, which ensures that recovery implementation is a simple process.

A small number of actions pose problems with this approach to idempotency and require handlers to store a small amount of state. One example is altering Xen CPU shares: this action is currently not atomic in Xen. Xen allows each virtual machine to have its CPU shares set independently of others, and there is no global set of CPU shares. To ensure the correct setting of shares we enforce that the sum of all shares on all virtual machines (including `domain-0`) sums to an arbitrary number, such as 100. We donate all unused shares to `domain-0` and reserve 20 shares for `domain-0` to execute any resource driver actions. However, the 100 share invariant is broken for a brief period of time when shares change since the handler must deduct shares from `domain-0` and then add them to a newly create virtual machine resource domain.

Shifting shares from one virtual machine to another is not an atomic operation in Xen. To make this operation idempotent, we augment Xen to write the transferred shares to disk before releasing them from `domain-0`. We then update the on-disk state when the shares have been released from `domain-0`, and remove the state when the virtual machine resource driver completes the process of adding the shares to the new domain. This problem points to the need for atomic actions to transfer resources for virtualization technologies between domains. In cases where resources do not implement atomic transfer actions the resource handler must preserve small amounts of state to log its progress in case of failure.

In addition to resource handlers, the guest handlers handlers must be idempotent to ensure that a guest recovers from actor failure or disconnection. We implement guest-level idempotency in a similar fashion as resource driver idempotency. State stored within the guest is sufficient: *join* and *leave* handlers query the guest during invocation to determine their state. For example, the GridEnginge batch scheduler stores the machines it has added to its batch pool internally in a set of flat files. The GridEngine *join* handler may use the internal state to decide whether or not to

issue duplicate GridEngine configuration directive.

7.5 Summary

In this chapter, we detail the use of leases for dealing with actor failures in a networked operating system. We focus on ensuring sustained operation by addressing actor failures and actor disconnections from peer actors, resources, and guests. We leverage the lifetime management property of leases to ensure sustained operation in the face of permanently disconnected actors, and combine commits of local state and idempotent cross-network requests to ensure a reconnected actor reenters the system gracefully. Finally, we discuss the impact of actor failures from a guests point-of-view and outline the implementation of the persistence and recovery architecture in our prototype.

Chapter 8

Flexibility and Performance

"Science is the systematic classification of experience."

George Henry Lewes

Removing all forms of policy from infrastructure resource management offers two primary advantages relative to other systems: flexibility and performance. Flexibility implies that guests may control a range of different resource types. Performance implies that the complexity and scalability of the extensible approach is acceptable. In this chapter, we detail the implementation of the lease abstraction within Shirako. We evaluate the architecture's flexibility and performance by discussing the implementation of a range of different resource types beneath the resource handler interface and conducting scalability experiments that show the performance is suitable for managing thousands of hardware components.

8.1 Goals

The contributions and insights of this thesis stem from years of experience designing, implementing, deploying, and using a real system. The team designed, developed, and deployed two COD prototypes [39, 120] before transitioning to the more extensible approach [1]. Since the transition to the extensible approach the major architectural components have been rewritten multiple times in order to separate the lease abstraction from the guests, resources, and policies that use it.

Resource handlers have been enhanced to improve their resilience and error-reporting, and to update them to support new versions of Xen and other evolving resources technologies, that continually experience independent API changes. We continuously incorporate different virtual machine disk images to support different guest environments. The extensible approach is a key to evolving the implementation along with physical and virtual hardware management technologies.

At the time of this writing, Shirako has been deployed and supporting group research activities for nearly a year. In particular, the Jaws guest we develop using Shirako, discussed in the next chapter,

[1] The specific contributors and their contributions are acknowledged in Section 1.5.

supports research into automated active learning of different application performance profiles [151]. We have also set up a deployment at a third-party research lab in Silicon Valley: the extensible approach was key to this external deployment because the research lab uses different tools and machine configurations than our research group. To support this deployment, we custom built guest and resource handlers using virtual machine disk images from rPath, an industry supplier of virtual appliances.

Experience from this integrative process indicates one of the drawbacks to the extensible approach: integration can be time-consuming and the system is only as good as the resources and guests that integrate with it to multiplex resources. While not defining policy frees a networked operating system to perform many different functions, it binds the system to the existing guest and resource technologies able to leverage it. Shirako incorporates a prototype of the narrow architecture presented in this thesis. The implementation and experiments we describe below represent a single snapshot of Shirako's implementation [2].

While the implementation below the guest and resource handler interfaces change to support different types of guests, resources, and configurations, the implementation of the lease abstraction and state machine presented in the previous chapters has been stable with continued development, representing a metric of success in evaluating leases as the fundamental programming abstraction in an operating system architecture for networked resource management.

8.2 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 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 by communicating 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. This

[2] For example, Shirako supports MySQL as well as LDAP for persistent storage. We only report LDAP numbers in our scalability experiments, since support for MySQL is recent. Note that this is a conservative choice, since LDAP performance is worse than MySQL.

Lease abstraction	5903
Lease state machines	1525
Utility classes	1836
Policy modules (mappers)	6695
Calendar support for mappers	1179

Table 8.1: Lines of Java code for Shirako/COD.

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 8.4. 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.

The implementation of the lease abstraction is not complex and is understandable by one person. Guest developers and site administrators use the Java libraries that define the lease abstraction to develop guest-specific service managers and resource-specific authorities. Table 8.1 shows the number of lines of Java code (semicolon lines) in the major system components of a current snapshot of the Shirako prototype. Note that external policy modules and calendar support represent nearly half of the code size; the implementation of the lease abstraction and state machine is compact.

Shirako provides a generic implementation of the resource and guest handler interfaces using Ant [161]. Ant is a Java-based tool that makes it possible to express a sequence of operations using an extensible declarative language. 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. Shirako also includes a few dozen Ant scripts, averaging about 40 lines each, and other supporting scripts. These scripts configure the various resources and guests that we have experimented with, including those described in Section 8.4. Finally, the system includes a basic Web interface for Shirako actors; it uses Velocity scripting code that invokes Java methods directly.

Shirako service managers and authorities associate an Ant XML file with each resource type. The Ant XML file implements the resource and guest handler interface. Specific Ant targets encode the sequence of operations necessary for each of the handler methods including *join*, *modify*, and *leave* for a service manager and *setup*, *modify*, and *teardown* for an authority. Ant allows actors to

share commonly used configuration operations between multiple resources, and makes it possible to assemble a handler from a collection of building blocks without requiring recompilation. 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.

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. The guest event handlers may connect to machines using key-based logins through *jsch*, a Java secure channel interface (SSH2). To store persistent state, actors optionally use *jldap* to interface to external LDAP repositories or the JDBC MySQL driver to interface with MySQL. The handlers for a COD authority also employ several open-source components for network management based on LDAP directory servers (RFC 2307 schema standard), as described in Section 4.4.

We evaluate a snapshot of the Shirako 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 resource handlers use 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 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 experiments use LDAP, a single LDAP server serves all actors on the same LAN segment. Note that these choices are conservative in that the management overhead concentrates on a single server.

8.3 Flexibility

One measure of neutrality is the number of different resources and guests that authorities and service managers support, respectively. We outline the different types of resources we use beneath the resource handler interface in our deployments. In the next chapter, we outline case studies of

specific guests that use these resources.

Guest and resource handlers interact with *resource drivers* to instantiate and control resources. Resource drivers encapsulate the hardware management services of a particular type of resource. For example, the resource driver for Xen exports methods to create a virtual machine, migrate a virtual machine, and destroy a virtual machine, amongst others. The input to each resource driver function is a property list defining specific runtime parameters needed to satisfy a handler invocation. Policy implementations are responsible for passing the required information to each resource handler—each policy must know the set of properties required to correctly execute the resource handlers and its corresponding resource drivers. The result of each resource driver method invocation is also a property list, which flows back to the actor, which persistently stores the result by attaching it to the corresponding lease. The property list conveys information about the outcome of a guest and resource handler invocation (*e.g.*, exit code and error message). An error may trigger a failure policy on the service manager or authority that resets a resource or re-assigns the resource to an equivalent machine.

As described in Section 7.1.2, to keep the state machine simple we allow lease state transitions to invoke guest and resource handlers, which issue resource driver actions, at anytime without requiring that all previous handler invocations are complete. For example, it must be possible to issue the *teardown* event handler to destroy a given resource regardless of what other resource handler invocations may be in progress. At the same time, the lease abstraction may invoke the same handler multiple times as a result of an actor or network failure, as discussed in Chapter 7. To account for these requirements, all guest and resource driver methods must be idempotent and thread-safe. Since each guest and resource handlers constitute a sequence of driver invocations, each handler is also idempotent.

The authority either sequences the invocation of multiple resource drivers inside the same resource handler or exposes partial control of resource driver interface, and the hardware management services it provides, to the service manager. The leasing architecture does not preclude the approach described in Chapter 5 that decouples virtual machines from slivers, and controls each sliver using a separate lease with its own resource handler.

The resource handlers and resource drivers we use in our deployment require the authority to

Resource	*Example Drivers*
Virtual Machines	Xen Hypervisor with CPU and memory slivering
Physical Machines	Disk re-imaging with PXE/DHCP network boots
Bandwidth limits	Linux `tc` utility
Disk Volume Management	Logical Volume Manager (LVM)
Block devices	iSCSI roots using a NetApp filer
Remote file systems	NFS roots using Sun's ZFS file system

Table 8.2: A table outlining the different resource drivers currently supported by Shirako. Aspects of the specific technology are hidden from the lease abstraction beneath the resource handler interface.

access hardware management services to control machine configuration. However, these resource handlers and resource drivers need few changes to decouple virtual machines and slivers and delegate control of hardware management services to service managers. The only required change is to allow a service manager to securely invoke the resource driver methods within the guest handler, instead of authority, as well as new broker policies that name slivers and virtual machines independently.

For example, authorities must define additional resource types for CPU share, memory allotment, and bandwidth limits, and minimal resource handlers that configure these types, and export these types to a new broker policy that individually issues tickets for them. Since our resource handlers adjust slivers using the *modify* event handler, this event handler is a basis for sliver resource handlers. Below we discuss resource handlers and resource drivers from our deployment of COD.

We also show how the architecture incorporates resources from other sites, with different resource management models. In particular, we develop resource handlers for Amazon's Elastic Compute Cloud [74] (EC2) to enhance EC2 and demonstrate Shirako's architectural separation of policy from mechanism. Authorities may resell or donate EC2 resources by defining a new resource type and handler and exporting EC2 resources to a broker, which may use any policy to allocate these resources to service managers. Service managers may also use lease grouping primitives with EC2 resources in the same way that they use them for resources instantiated at any authority.

8.3.1 Physical Machines, Virtual Machines, and Slivers

The authority prototype includes implementations of resource drivers for physical machines, Xen virtual machines, as well as multiple storage devices, as shown in Table 8.2. Storage technologies that the authority supports include flash-cloning iSCSI-accessible block devices on a NetApp FAS3020 filer and creating, snapshotting, and cloning local LVM block devices, file-backed block devices, and

Function	*Description*
boot(ResourceId vm)	Boot a new virtual machine.
shutdown(ResourceId vm)	Shutdown a virtual machine.
setCpu(ResourceId vm)	Set a virtual machine's CPU share.
setMemory(ResourceId vm)	Set a virtual machine's memory allotment.
setBandwidth(ResourceId vm)	Limit a virtual machine's outgoing bandwidth.
reboot(ResourceId vm)	Reboot a virtual machine.
reset(ResourceId vm)	Reset a virtual machine and its disk image to the initial state.
save(ResourceId vm)	Save the the running state of a virtual machine.
restore(ResourceId vm)	Restore the running state of a virtual machine.

Table 8.3: A table outlining the resource driver interface for a virtual machine. The authority controls the binding of slivers to virtual machines, which are in italics. An authority may export control of the hardware management services in bold since their execution does not concern resource multiplexing or interfere with competing guests.

NFS root-mounted ZFS file systems. Virtual machines may also be broken into leases for individual slivers of CPU, memory, bandwidth, and storage. The virtual machine resource driver defines the collection of functions required for the authority to interact with the Xen virtual machine monitor (Table 8.3) to configure virtual machines.

A COD authority leases machines. A COD authority's resource handler encodes the configuration and type of machine. Configuration properties allow an authority to expose some control over each machine's configuration to the service manager. A COD authority controls and manipulates administrative services that bind host names to IP addresses (*e.g.*, using BIND), user logins to machines (*e.g.*, using PAM), user access permissions to the file system (*e.g.*, using NSS), and remote pools of storage (*e.g.*, using NFS). These configuration properties flow to the resource handler, which decide how to interpret and use them to direct each machine's configuration. Since the COD authority operates these services on behalf of the service manager and its guest, the guest does not have to deal with the complexity of managing administrative services, itself.

Figure 8.1 shows an example of a simplified resource handler for a Xen virtual machine in our COD deployment. In this example, the creation of a virtual machine involves several distinct operations: add an entry to LDAP to configure a DNS name, create an iSCSI root disk, create local scratch space, boot the virtual machine, and configure the Xen resource scheduler to impose limits on the resources consumed by the virtual machine. Each of these operations requires interaction with a resource driver.

Both virtual and physical machine (in Figure 8.2 resource handlers require similar inputs to

```
<project name="Xen Virtual Machine" basedir=".">
 <description>
  This handler contains directives
  for allocating Xen virtual machines
  using cloned iSCSI root disks.
 </description>

 <target name="setup"
        description="Create a Xen VM with a DNS name, a cloned
                     root disk, scratch disk space, and reserved
                     resource sliver.">
  <dns.ldap.add/>
  <iscsi.clone.disk image=${image.name}/>
  <lvm.create.disk size=${size}/>
  <vm.create
       root="${iscsi.disk}
       disk="${lvm.disk}"
       kernel="${linux.kernel}"/>
  <vm.setshares
      cpu="${cpu.share}"
      memory="${memory.mb}"
      bandwidth="${bandwidth.mb}"/>
 </target>

 <target name="modify">
         description="Migrate a VM or change its sliver size.">
  <vm.migrate
      machineName="${unit.dns.hostName}"
      destination="${new.host.net.ip}"
      live="${boolean.live}"/>
  <vm.changeshares
      cpu="${new.cpu.share}"
      memory="${new.memory.mb}"
      bandwidth="${new.bandwidth.mb}"/>
  </target>

 <target name="teardown">
         description="Destroy a Xen VM's DNS name and root disk.">
  <dns.ldap.remove/>
  <vm.remove lvm="${lvm.disk}"/>
  <iscsi.remove.disk/>
 </target>
</project>
```

Figure 8.1: Handler for configuring a virtual machine. Note that the handler is a simplification that does not show the details of each Ant target.

```
<project name="physical" basedir=".">
 <description>
  This handler contains directives
  for allocating physical machines.
 </description>

 <target name="setup"
        description="Re-image and reboot a physical machine.">
  <dhcp.ldap.add/>
  <exec executable="${reimage.disk} ${host.ip} ${image.name}"/>
  <exec executable="${reboot} ${host.ip}"/>
  <exec executable="${probe} ${unit.ip}"/>
 </target>

 <traget name="modify"/>

 <target name="teardown"
        description="Re-image and network boot a physical machine.">
  <dhcp.ldap.remove/>
  <exec executable="${reboot} ${unit.ip}"/>
  <exec executable="${probe} ${host.ip}"/>
 </target>
</project>
```

Figure 8.2: Handler for configuring a physical machine. Note that the handler is a simplification that does not show the details of each Ant target.

their respective handlers. Both require an IP address, specification of a kernel, a root disk image (whether local or remote), a hostname, and remote disks and/or file systems. Additionally, virtual machines support slivering the resources of a physical host and migration. Note that the virtual machine resource handler may use the *modify* handler to change the sliver size or migrate the virtual machine on lease boundaries, while the physical machine handler does not have the capability to perform these tasks.

The fundamental difference between the two resource handlers resides in the technology to initiate the booting of their respective kernels: the presence of a control domain with a command line or programmatic API simplifies the process to initiate boot of a virtual machine. The physical boot process may become simpler in the future as blade chassis with management backplanes that provide rich hardware management interfaces become more predominant.

The configuration attributes and policies for fulfilling requests for physical and virtual are also similar. The resource, configuration, and unit properties in Table 4.4 are common to both physical and virtual machines. A basic assignment policy in both cases matches a request to a physical host: in the case of a physical machine the policy represents the “host” as the physical machine in the trampoline state, while in the case of a virtual machine the policy represents the host as the control domain used to create virtual machines.

An authority may use the same assignment policy for physical and virtual machines if it limits each physical machine to hosting a single virtual machine, as discussed in Section 5.3.1. However, part of the power of virtual machines is the ability to co-locate machines on the same physical host, and use performance isolating schedulers to ensure that the actions of one virtual machine does not affect the behavior of another. An authority may allocate each virtual machine a sliver of the resources of the host physical machine using the methods from Chapter 5 to specify sliver size and/or name physical machines, slivers, and virtual machines. Our current prototype specifies sliver size and names physical machines to provide sliver sizing capabilities.

8.3.2 Amazon's Elastic Compute Cloud

The architecture is flexible enough to allow authorities control resources offered by third-party infrastructure providers. In this case, the authority uses interfaces exported by the infrastructure provider to configure hardware resources. Since authorities are independent of the resources they control their design permits the reselling of resources from third-party providers. Reselling resources is beneficial if third-party providers do not provide policies or programming abstractions that arbitrate resources under constraint or enable guest development. As a test of the architecture we develop resource handlers for an authority to instantiate virtual machines from Amazon's Elastic Compute Cloud [74] (EC2).

EC2 is a service for users to upload virtual machine disk images and boot them on Amazon's infrastructure. The service provides web service (*e.g.*, SOAP) interfaces that allow third-party software, such as a Shirako authority, to programmatically create virtual machines as they need them. Thus, EC2 implements functionality similar to resource drivers for virtual machines—a network accessible programmatic API to control virtual machine instantiation and management. However, EC2 does not lease resources, which limits the allocation policies it supports. Not supporting leases impacts EC2's operation: as of this writing, EC2 requires users to contact an EC2 administrator if they wish to create more than 20 virtual machines. They also do not support advance machine reservations.

Instead of using leases, EC2 provides customers with an assurance that it will not terminate a running virtual machine—if the infrastructure becomes overloaded EC2 simply rejects new customer

requests until available resources exist. EC2 does not provide primitives for distributed guests, such as lease groups, to enable development of guests that adapt and control their virtual machines. While guests may use the EC2 interfaces to request machines, they do not have the necessary primitives for grouping resources and sequencing configuration operations.

EC2 is similar to the original incarnation of COD [39] that managed a single type of resource—a machine. Shirako's integration requires a resource handler that accesses the EC2 web services interface to issue requests to run instances of EC2 virtual machines. The authority creates and owns these machines. By default, the machines boot with a public key registered by the COD authority with the EC2 service. The *setup* handler uses this key to install a public key specified by the requesting service manager so the service manager can access the machine using `ssh`.

Internally, the authority represents EC2 virtual machines as simply another resource type it may allocate to service managers and brokers. The authority requires a simple assignment policy since the EC2 service hides the physical assignment of the EC2 virtual machines from its customers (in this case the authority)—the authority uses a simple assignment policy that treats the virtual machines as being hosted by a single machine represented by the EC2 service. The authority also does not need to allocate IP addresses since the EC2 service allocates EC2-specific IP addresses. In all other respects, the operation of the authority remains the same.

Virtual machines are an execution environment that binds to slivers. EC2 does not support adaptation of a virtual machine's sliver. EC2 also has no direct support for brokers to enable Amazon to delegate resource rights in order to coordinate the allocation of resources from multiple sites. The integration with Shirako allows EC2 to take advantage of these brokered policies for leasing resources. For example, using leases, a Shirako broker may implement a policy that limits the number of machine-hours allowed for EC2 machines. Since Amazon charges $0.10 per hour of machine time the policy may set an upper bound on the amount of money a COD authority expends managing EC2 machines.

Adherence to the neutrality principle made Shirako's integration with EC2 simple: we only needed to write a simple resource handler that implements the *setup*, *modify*, and *teardown* event handlers. We were then able to leverage Shirako's existing virtual machine adaptation, provisioning, and assignment policies.

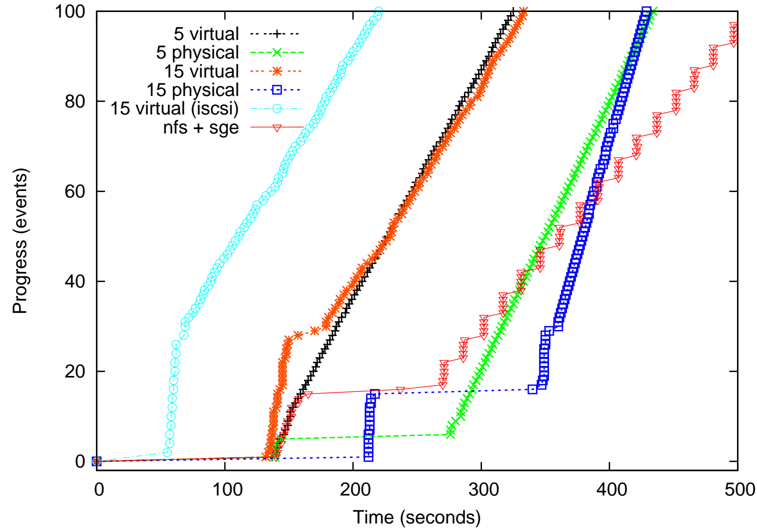

Figure 8.3: The progress of *setup* and *join* events and CardioWave execution on leased machines. The slope of each line gives the rate of progress. Xen clusters (left) activate faster and more reliably, but run slower than leased physical nodes (right). The step line shows an GridEngine batch scheduler instantiated and subjected to a synthetic load. The fastest boot times are for virtual machines with flash-cloned iSCSI roots (far left).

8.4 Performance

8.4.1 Illustration

The previous sections describe the different resources that an authority may lease to service managers. In this section, we demonstrate how this flexibility is useful. For these experiments, we use a simple policy that views each physical Xen server as a pool of seven fixed-size slivers, (with 128 MB of memory each), or computons, for binding to virtual machines, as described in Section 5.3.1.

We first examine the latency and overhead differences to lease physical and virtual machines for a sample guest, the CardioWave parallel MPI heart simulator [135]. A Cardiowave-aware service manager requests two leases: one for a coordinator machine to launch the MPI job and another for a variable-sized block of worker machines to run the job. It groups and sequences the lease joins as described in Section 4.3.2 so that all workers activate before the coordinator. An MPI application has no notion of a master as in a batch scheduler. However, some machine must launch the application, and in our service manager we request a coordinator machine to serve this purpose. The *join* handler launches CardioWave programmatically when all machines are fully active.

Figure 8.3 charts the progress of lease activation and the CardioWave run for virtual clusters of 5

and 15 machines, using both physical and Xen virtual machines, all with 512MB of available memory. The guest earns progress points for each completed machine 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 machine and initiates boots; (2) a step up as machines boot and join, (3) a second flatter portion indicating some straggling machines, and (4) a linear segment that tracks the rate at which the guest completes useful work on the machines once they are running.

The authority prepares each machine 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 machine setup delays result from contention to load the images from a shared NFS server, demonstrating the value of smarter image distribution (*e.g.*, [89]). The left-most line in Figure 8.3 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 virtual machine configuration time to approximately 35 seconds. Network booting of physical machines 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 machines, but its performance degrades beyond ten machines. Five Xen machines are 14% slower than the physical machines, and with 15 machine they are 37% slower. For a long CardioWave run, the added Xen virtual machine overhead outweighs the higher setup cost to lease physical machines. This exercise represents the benefits of resource flexibility to a guest: a service manager may request physical machines for a long Cardiowave run and virtual machines for a short Cardiowave run.

A more typical usage of a service manager in this setting would be to instantiate batch schedulers on machines [39], and let them schedule Cardiowave and other jobs without rebooting the machines. An extensible authority does not preclude this option. Figure 8.3 includes a line showing the time to instantiate a leased set of five Xen machines and an NFS file server, launch a standard GridEngine batch scheduler on them, and subject it to a synthetic task load. This example uses lease groups to sequence configuration as described in Section 4.3.2. 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.

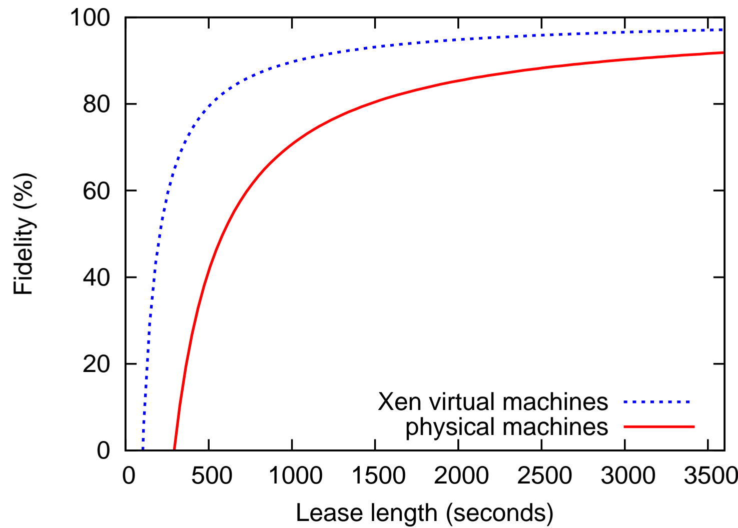

Figure 8.4: Fidelity is the percentage of the lease term usable by the guest, excluding setup costs. Xen virtual machines are faster to *setup* than physical machines, yielding better fidelity.

Figure 8.4 uses the guest and resource handler 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 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.

The experiment above demonstrates two different service managers that execute Cardiowave runs: one that directly leases machines for individual runs and one that leases machines for a batch scheduler which queues a schedules multiple Cardiowave runs. Shirako supports multiple approaches to executing tasks that exhibit different tradeoffs. Installing a batch scheduler allows a user to inject multiple Cardiowave runs and let the batch scheduler schedule them on the available resources. In this case, the service manager simply examines the batch scheduler's load and adjusts the number of machines in the batch pool based on load. However, batch schedulers do not exercise the control exported by an authority to manage each job individually. In the next Chapter, we experiment with a third alternative that combines these two approaches using the extensible leasing architecture: Jaws is a minimal service manager that accepts task submissions and instantiates one or more virtual machines for each task, such as Cardiowave.

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

Table 8.4: Parameter definitions for Section 8.4

8.4.2 Scalability

We use emulation experiments to demonstrate how the implementation of the lease abstraction scales at saturation. The scaling experiments depend on the number of leases processed by Shirako and is independent of resources being leased (*e.g.*, slivers, virtual machines, physical machines) and the naming schemes detailed in Chapter 5. For simplicity, these experiments use the site-assigned computons naming scheme from Section 5.3.1. Table 8.4 lists the parameters used in our experiment: for a given number of machines N at a single site, one service manager injects lease requests to a broker for N machines (without lease extensions) evenly split across l leases (for $N/l = n$ machines 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 machines. 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.

As explained in Section 7.1.2, 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.

In this experiment, all actors run on a single x335 server and communicate with local method

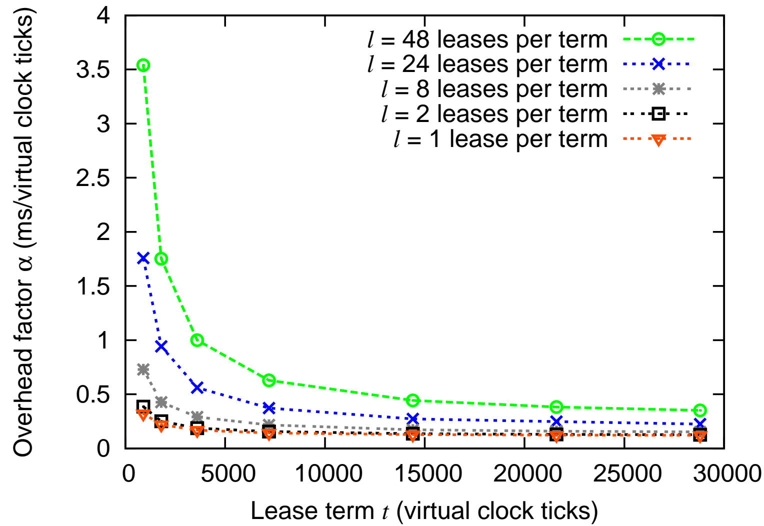

Figure 8.5: The implementation overhead for an example scenario for a single emulated site with 240 machines. As lease term increases, the overhead factor α decreases as the actors spend more of their time polling lease status rather than more expensive setup/teardown operations. Overhead increases with the number of leases (l) requested per term.

calls and an in-memory database (no LDAP). Figure 8.5 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 machines, 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 8.5 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 machine (*i.e.*, to write a new OS image on local disk and boot it). As an example from Figure 8.5, 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 an authority, these overheads are at least an order of magnitude less than the *setup/teardown* cost of machines 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

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 8.5: 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'.

Database	α	stdev α	t'	r'
Memory	.1743	.0001	627	.3824
LDAP	5.556	.1302	20,003	.0120

Table 8.6: Impact of overhead from LDAP access. 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'.

on more fine-grained resource allocation, such as CPU scheduling, where reassignments occur at millisecond time scales.

Table 8.5 shows the effect of varying the number of machines N on the overhead factor α. For each row of the table, the service manager requests one lease (l=1) for N machines (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 the number of machines, but as before, t' remains an order of magnitude less than the setup/teardown costs of a machine.

We repeat the same experiment with the service manager, broker, and authority writing their state to a shared LDAP directory server. Table 8.6 shows the impact of the higher overhead on t' and r', for N=240. Using α, we calculate the maximum number of machine flips per millisecond $r'=N/(T\alpha)$ at saturation. The LDAP overheads dominate all other lease management costs: with $N = 240$ nodes, an x335 can process 380 machine flips per second, but LDAP communication overheads reduce peak flip throughput to 12 machines per second. Even so, neither value presents a limiting factor for today's sites which contain thousands of machines. Using LDAP at saturation requires a minimum lease term t' of 20 seconds, which approaches the setup/teardown latencies (Section 8.4.1).

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

configuration actions and log them.

8.5 Summary

In this chapter, we present an overview of a prototype implementation of a networked operating system. We evaluate the neutrality principle along two axis: flexibility and performance. We examine the benefits of supporting different resource types in a real deployment and the architecture's integration with Amazon's Elastic Compute Cloud. We supplement our evaluation with experiences deploying and using a prototype within our research group and at an external site.

Chapter 9

Guest Case Studies

"Research is to see what everybody else has seen

and to think what nobody else has thought."

Albert Szent-Gyorgi

In this chapter, we evaluate our architecture by integrating multiple types of guests and coordinating resource sharing between them. The guests include the GridEngine batch scheduler [75], the PlanetLab network testbed [23], the Plush distributed application manager [9], and a job management system for virtual computing called Jaws. Related work uses the architecture to develop adaptive service managers for other guests as well, including the Rubis web application [185] and Globus grids [138].

Developing a service manager for each guest exercises each aspect of the leasing architecture including developing guest, resource, and lease handlers and custom cross-actor properties that drive handler invocation from Section 4.3.1, determining guest dependencies that require lease groups from Section 4.3.2, writing customized guest monitoring systems to gather data to react to load, and developing adaptation policies that use monitoring data to drive resource requests. For each guest we first give an overview of the guest's capabilities and then describe the implementation and integration of each prototype guest in Shirako with references to the architectural elements from Chapter 4 and Chapter 5. We then discuss lessons learned from the integration of each guest.

9.1 GridEngine

GridEngine [75] is widely used to run compute-intensive batch tasks on collections of machines. It is similar to other local batch job schedulers for cluster and grid sites [67, 113, 183]. A GridEngine service manager is able to alter GridEngine's resource allotment, using visible allocation, modification, and revocation, by using existing administrative functions to introspect on GridEngine. Importantly, the service manager does not need to modify GridEngine to adapt its resource allot-

ment. Service manager resource requests are driven by an adaptation policy that determines how to request resources in response to task submissions.

The adaptation policy defines a feedback loop: the service manager monitors the size of the task queue over time and uses the queue size to formulate resource requests. The service manager transmits requests for resource tickets to a broker and redeems tickets granted by the broker at an authority. Once an authority finishes instantiating and configuring the machines it notifies the service manager, which invokes *join* for each machine instantiated by the authority in order to incorporate the new machines into the batch scheduler's collective environment. Once the service manager incorporates the machines into the guest batch scheduler's environment, the guest batch scheduler is free to control the machines by scheduling and executing tasks on them according to its internal scheduling policy.

The description of the adaptation policy above belies some of the complexities of implementing the service manager's feedback loop. For example, batch schedulers use a single machine to act as a *master*, which schedules tasks across a set of machines that act as *workers*. The relationship between master and worker necessitates the use of lease groups to bind worker machines to the master. The process of monitoring a batch scheduler requires periodically polling its task queue length using some form of remote execution engine, such as `ssh`. A long queue of tasks indicates that the batch scheduler needs more machines to increase the rate of task completion. As described in Chapter 8, service managers implement a clock that ticks at a specified rate—service managers may monitor guests at clock intervals.

A service manager must also inspect 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. Service managers may use the guest lease handler's `onExtendTicket` (see Section 4.3.2) method to inspect the task queue before each lease's expiration to determine whether or not to extend a lease, and, if extending the lease, how to flex or modify it.

9.1.1 Integration

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. To maintain a uniform environment across the active set, as required by GridEngine, each worker must define a set of user identities eligible to use the batch scheduler, 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.

We developed a GridEngine service manager using Shirako. The service manager leases resources from brokers and authorities and instantiates and adapts the number of machines allotted to GridEngine. The core of the GridEngine service manager is a set of guest handlers that alter GridEngine's resource allotment and an adaptation policy module that periodically polls GridEngine's task queue and formulates resource requests. The implementation of GridEngine itself is unchanged. We first discuss the guest handlers for the GridEngine service manager and then discuss the adaptation policy of our prototype.

Guest Handlers

The GridEngine guest handlers encapsulate administrative complexities for incorporating machines into GridEngine's environment. The GridEngine service manager uses 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.

A GridEngine service manager uses lease groups to sequence the instantiation of the guest handlers for the master and each worker. Lease groups also permit the lease lifetimes of the master and each worker to be different; the lease for the GridEngine 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. The GridEngine service manager initially formulates a lease request for one machine to act as the GridEngine master, which we call the master lease. To group each worker lease with the master

lease, the service manager sets the master lease as the predecessor of each worker lease, as described in Section 4.3.2. 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.

To add a master, *join* installs the GridEngine software to the master machine. In our COD prototype, the *join* handler mounts a well-known file volume served by an existing NFS server and copies the GridEngine software to the master. The *join* entry point for the master lease then executes the *sge_master* daemon. The *leave* handler for the master lease stops the *sge_master* daemon and removes the installed GridEngine software. Recall that both the GridEngine master and each worker require a shared file system. In our prototype, *join* for the master starts an NFS server and exports a file volume to each worker machine. 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. In our prototype, either the master machine acts as the NFS server or the master and each worker mount a file volume exported by a pre-existing NFS server.

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.privIPaddr`. 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.

Adaptation Policy

Our prototype implements a simple adaptation policy: request a new worker for every X pending tasks and relinquishes any idle worker at the end of its lease. The policy illustrates dynamic adaptation of a guest within our architecture. For this policy 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. As a result, our prototype uses computons and does not monitor the resources of each worker to request adjustments in sliver size that match a task's resource usage. Defining more sophisticated policies that account for spikes in task submissions, machine instantiation overheads, and sliver sizing is left for future work.

The GridEngine service manager monitors the task queue at clock tick granularities (one or more) to check the size of the task queue. The monitoring engine accesses the GridEngine *qstat* command using a secure `ssh` channel to obtain a list of queues and the tasks scheduled to them. If there are queued tasks that have not yet started, the service manager requests a new worker for every X queued tasks. 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. Brokers may choose to not extend leases according to their own arbitration policies, in which case, the service manager has no recourse but to fall below the k threshold. In our deployment 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 service manager uses the `onExtendTicket` function of the lease handler from Table 4.3 to implement a victim selection policy: the function inspects the task queue for each worker in the lease to determine which workers are executing tasks and which workers are idle. If idle workers exist, the service manager flexes the lease (see Section 4.3.2) in its ticket extension request to match the number of workers currently executing tasks. The service manager appends the name of idle workers to the ticket using a configuration property (*i.e.*, `lease.victims`) before redeeming the

ticket at the authority. The authority uses the names to select victim machines to teardown on a shrinking lease. In our prototype, the service manager uses the IP address of the worker to name victim machines.

One interesting aspect of a purely reactive GridEngine adaptation policy is that it never flexes a lease to grow the number of workers on a lease extension boundary. We chose to make the policy highly responsive to task submissions by allowing it to generate new resource requests as soon as the service manager detects a new task. 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 scalability experiments from Section 8.4 show that 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.

9.1.2 Demonstration

We present simple experiments with GridEngine using an initial Shirako prototype under trace-driven task load. Our initial prototype of the leasing architecture combined the arbitration policy of the broker with the assignment policy of the authority—importantly, as the prototype has evolved and changed over time the functions of the guest handler—*join* and *leave*—for GridEngine have remained the same, validating the separation of the guests and resources from the lease abstraction. The results demonstrate dynamic provisioning behavior with the policy described in the previous subsection. We ran our experiments with multiple similarly configured GridEngine batch schedulers on a testbed of 80 rackmount IBM xSeries-335 uniprocessor servers within the "Devil" Cluster at Duke Computer Science.

Traces. To stress the prototype under high levels of contention and resource constraint, we construct test trace segments drawn from a nineteen-month trace of production batch queue activity on the Devil Cluster. The full trace starts in September 2001 and continues until mid-April 2003. Each trace entry contains a submit timestamp, task start time, task completion time, and the user identity and executable file name for the task. We divide the GridEngine users into three user groups, select a three-month window of activity for each group, and combine the trace segments to form the test trace. The three user groups are:

- **Systems.** Researchers in networking and peer-to-peer systems submitted large numbers of short tasks, usually no more than a few minutes long. Activity from this group was highly bursty. The Systems trace segment runs from 17:01:52, 2002/02/01 until 00:00:00, 2002/05/01.
- **Architecture.** This team submitted a large number of computer architecture simulations, each consuming many hours of CPU time. Task arrival rate for this group was relatively stable. The Architecture trace section runs from 23:43:50, 2001/09/14 until 00:00:00, 2001/12/01.
- **BioGeometry.** These tasks evaluate new algorithms to predict protein folding and docking. Submitted tasks ran for days or weeks. This group submitted tasks with steadily increasing frequency. The BioGeometry trace segment runs from 18:34:47, 2002/10/03 until 00:00:00, 2003/01/01.

In the first test we ran the system on a testbed of seventy-one servers for twelve hours to examine the behavior of the provisioning policies. The test instantiates three GridEngine batch schedulers in separate virtual clusters, then replays each trace segment from above to each batch queue in real time. All trace records execute a task that spins in a busy loop for a specified time. To accelerate the experiment, we sped up the submission and completion of tasks by a factor of 160. This speedup allows a full three-month trace to complete in under twelve hours. While speeding up the trace introduces significant error in the flip times of machines, the general trends of machine allocations are not affected.

Each GridEngine service manager ticks every seven seconds to negotiate for resources according to the adaption policy in the previous subsection. The adaptation policy requests one machine for every 15 tasks still in the queue and relinquishes a machine after being idle for 60 seconds. We use a fixed priority ordering for allocating resources to the batch pools that guarantees each pool a minimum of two machines. In this experiment leases are less than the epoch allowing the reallocation of all resources at every epoch boundary. In our experiment, the Architecture group has the highest priority, the BioGeometry group has middle priority, and the Systems group has lowest priority.

Figure 9.1 and Figure 9.2 show the number of active machines and queued tasks, respectively, for all three groups over a selected eight-day period. The graphs show time in days, where each day

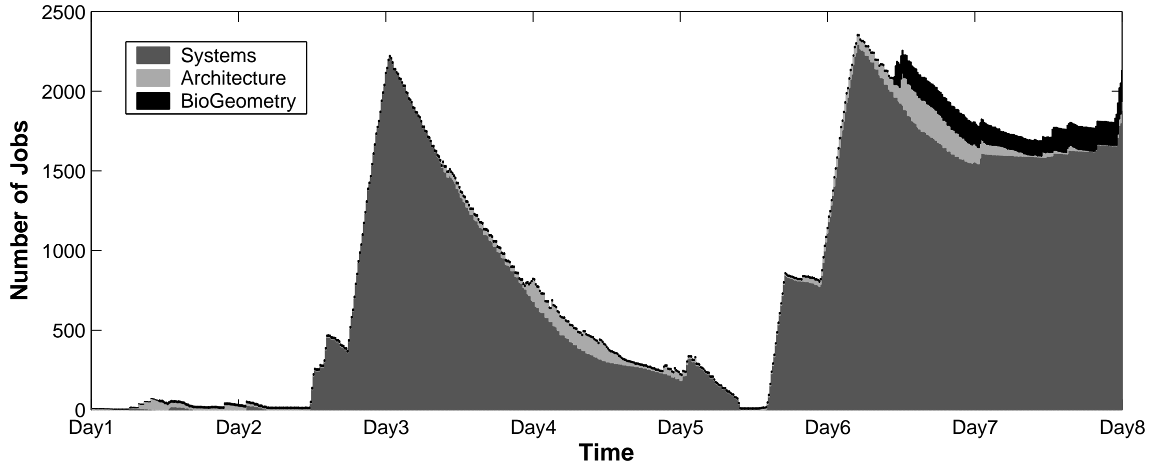

Figure 9.1: Number of GridEngine tasks in each batch queue over time during a trace-driven execution. Note from the y-axis that the batch scheduler is experiencing intense constraint from the task load (2400 tasks at peak load) relative to the 71 machines that are available.

represents approximately nine minutes in real time. We examine the stages of resource contention and negotiation visible during this time period.

Initially the cluster is underutilized for approximately two days. A brief second stage involves a large spike in demand from the Systems group, as the number of queued tasks increases over 2000—accommodating this burst requires pulling machines from the idle pool and allocating them to the Systems virtual cluster. While the Systems virtual cluster is growing, the Architecture group submits over 100 tasks to its GridEngine pool. Due to the load spike from Systems, there are no free machines available. Since the Architecture group has the highest priority, the site broker reclaims machines from the other virtual clusters, primarily the low-priority Systems virtual cluster, and transfers them to Architecture.

Figure 9.3 focuses on the Architecture group activity from the same experiment to clearly illustrate the relationship between the length of the task queue and the number of machines in the virtual cluster. As the queue length grows, the GridEngine service manager obtains more machines from COD to deliver a faster turnaround time on Architecture tasks. GridEngine distributes the tasks to the new machines, restoring equilibrium and causing the queues to shrink. As machines become idle, the service manager relinquishes them back to the global resource pool. If the size of the queue is below the request threshold, X — for example, midway through day two to midway through day three — the service manager leaves the virtual cluster at roughly constant size.

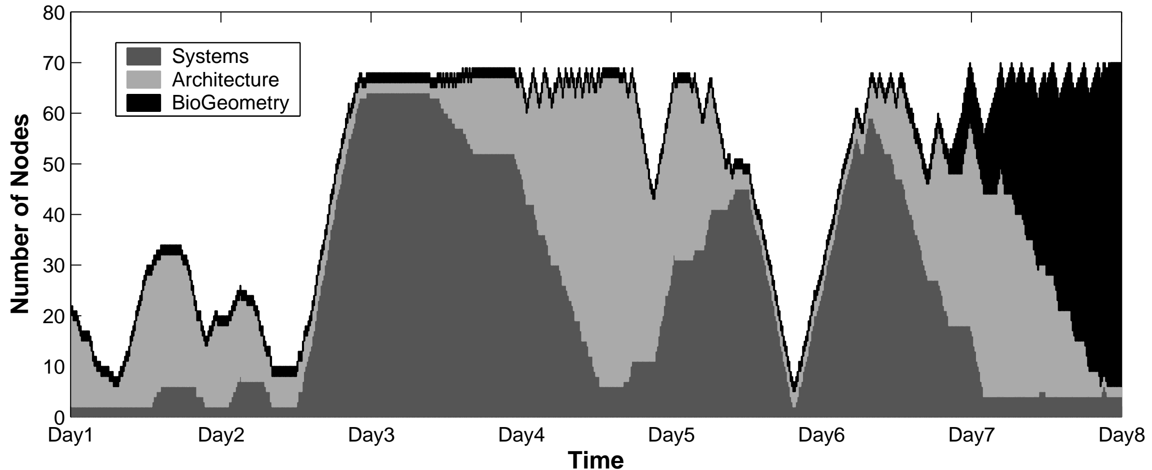

Figure 9.2: Number of machines in each of three virtual clusters over time during a trace-driven execution. Machines transfer to higher priority research groups as their task load increases. Strict priority arbitration results in machine reallocations to the highest priority research group whenever they have queued tasks.

Starting on day six, the Systems virtual cluster receives a burst of over 2000 tasks, and requests machines from the idle pool. It keeps these machines until the higher-priority Architecture and BioGeometry receive new task bursts and start competing for machines. Since Architecture is higher priority than BioGeometry, it acquires more machines and retires its tasks faster, eventually relinquishing its tasks to BioGeometry.

While this experiment uses a simple adaptation policy, it illustrates dynamic policy-based provisioning with differentiated service for GridEngine batch pools within our architecture, without any special support for advanced resource management in the batch scheduler itself.

9.1.3 Lessons

Using the GridEngine service manager, we can instantiate batch schedulers on demand, and dynamically resize the resource's of each batch scheduler according to task load. The authority isolates users of different batch schedulers, and brokers may apply arbitrary policies to allocate resources to the pools under constraint. Our prototype broker assigns a priority and a minimum guaranteed reservation to each batch pool. A low-priority batch pool is similar to the Condor resource-scavenging model [113]; that is, the broker policy allocates machines to the batch pool only when they are idle. The approach ensures a consistent environment across the batch pool at the price of a higher ma-

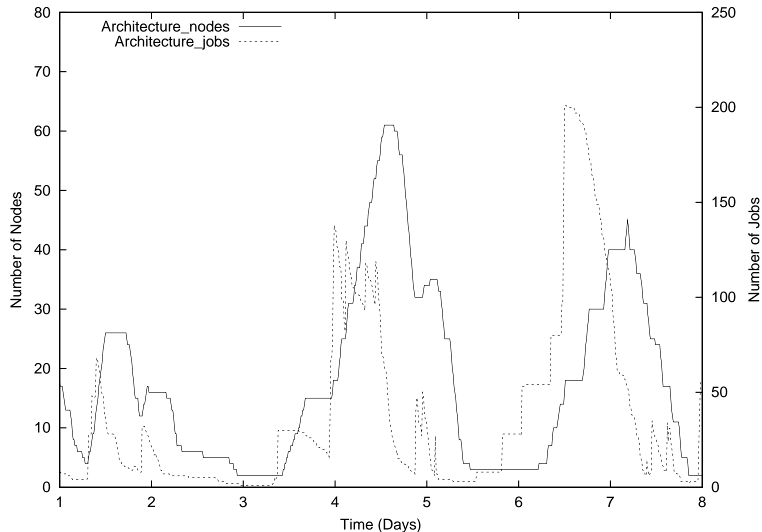

Figure 9.3: Combined size of the Architecture running and pending task queues, and virtual cluster size over an eight-day period.

chine allocation cost to reinstall. The model also protects users with fixed leases against interference from competing batch schedulers.

Many batch schedulers support multiple queues, and some schedule tasks according to priority. Shirako's approach provides these and other resource management features (*e.g.*, reservations and leases) at the system level, without relying on support within the batch scheduler middleware. The approach also allows more flexible policies for allocating resources between the batch scheduler and other competing environments (*e.g.*, grids, web applications). As we demonstrate in this chapter, the features extend to other environments.

As noted earlier, we may extend the model directly to Globus grids, which coordinate task scheduling across multiple batch schedulers, including GridEngine, at multiple sites across a wide-area network. In the grid, local site managers pass information about their available resources to a global grid manager, which makes informed global scheduling decisions about where to route tasks [70]. Shirako enables multiple distinct grids to run in isolated partitions of a shared physical cluster [138].

GridEngine illustrates a simple example of dynamic adaptation and the power and generality of Shirako's policy-neutral architecture. In this case, implementing a GridEngine service manager was straightforward and required no modifications to GridEngine itself to support a rich set of resource

management capabilities. Since GridEngine is middleware it runs above existing node operating systems and does not require more powerful functions to manage the hardware as long as the node operating system is compatible with the GridEngine daemons (*e.g.*, a UNIX variant).

9.2 Plush

Plush [8, 9] manages the deployment of distributed applications by defining phases that govern the application life-cycle: describing software installation and process workflows, discovering resources (SWORD [124]), acquiring resources, transferring software and starting processes, monitoring the application, and providing notifications of application completion and failure. These functions define abstractions for managing distributed applications—each phase defines functionality necessary for execution. Interaction with Plush occurs either manually through a shell environment using a command line or programmatically using an XML-RPC interface. Plush is similar to other high-level shell environments that provide a consistent view of networked resources by abstracting the process of application execution [76]. It was originally designed to execute guests on PlanetLab (see Section 9.3).

We integrate Plush with Shirako to evaluate a guest that implements its own abstractions for distributed applications. In contrast to GridEngine and PlanetLab, Plush is an example of a guest that implements abstractions without multiplexing resources. Plush defines a common interface for executing distributed applications in the wide area where it must account for network and host failures. As a result, Plush masks the allocation and revocation of resources from the applications that run above it. A Plush service manager leverages this characteristic: the service manager exposes resource allocation and revocation to Plush, which then masks it from the higher-level application. Without Plush, distributed applications that use Shirako to acquire resources would require support for visible allocation and revocation. However, a Plush service manager enables existing applications, which already leverage Plush's abstractions, to execute in our architecture without modification.

Method	*Description*
`Load`	The command loads an application description into Plush. The function takes an XML specification as a paramater. `Load` has an equivalent shell command.
`Run`	Initiates the execution of the application once the required resources are available. `Run` has an equivalent shell command.
`AddResource`	The *join* method for the Plush service manager's guest handler executes this method to add a resource to the Plush master's pool.
`RemoveResource`	The *leave* method for the Plush service manager's guest handler executes this method to remove a resource from the Plush master's pool.
`CreateResource`	Invokes a method exported by the Plush service manager. The Plush master uses the method to inject resource requests specified in the application description. `CreateResource` has an equivalent shell command.
`NotifyExit`	Notifies the Plush service manager of the completion of an application.
`NotifyFailure`	Notifies the Plush service manager of the failure of an application.

Table 9.1: A description of functions exposed by the Plush master via XML-RPC. External entities, such as a service manager, call `Load`, `CreateResource`, `Run`, `AddResource`, and `RemoveResource` to drive application execution. The Plush application description registers two callbacks with the Plush service manager, `NotifyExit` and `NotifyFailure`, which the Plush master invokes when an applications completes or fails, respectively.

9.2.1 Integration

Plush consists of a master process that coordinates application deployment and a set of agent processes that execute on each remote machine. As stated above, Plush divides the life-cycle of an application into distinct phases: application description, resource discovery, resource acquisition, application deployment, and monitoring. Plush requests resources through a Plush service manager as part of its resource acquisition phase. To avoid confusion, we use the term guest in this section to refer to the Plush agents and the term application to refer to the software environment that the Plush agents execute.

We wrote a Plush service manager that communicates with the Plush master, written in `C++`, using an XML-RPC interface to coordinate the acquisition of resources and deployment for distributed applications. Plush users may launch and control distributed applications from a terminal interface using a set of supported commands that drive the application's execution. Important commands include the following: the `load` command loads an application description (Figure 9.4), the `create resource` command issues requests to the Plush service manager to create resources, and the `run` command initiates execution of the application if its resources are available.

A service manager may invoke these commands programmatically using an XML-RPC interface exported by the Plush master. Table 9.1 describes the XML-RPC interface that defines the communication between the Plush master and the Plush service manager. In Section 9.4, we describe

```
<?xml version="1.0" encoding="utf-8"?>
<plush>
 <project name="test_project">
   <component name="test_cluster">
     <rspec>
      <num_hosts>1</num_hosts>
      <shirako>
       <num_hosts>1</num_hosts>
       <type>4</type>
       <cpu>100</cpu>
       <memory>256</memory>
       <bandwidth>500</bandwidth>
       <storage>99</storage>
       <lease_length>40</lease_length>
      </shirako>
     </rspec>
    <resources>
      <resource type="ssh" group="shirako"/>
    </resources>
   </component>
   <experiment name="simple">
     <execution>
       <component_block name="cb1">
         <component name="test_cluster" />
         <process_block name="pb1">
           <process name="date">
            <path>date</path>
           </process>
         </process_block>
       </component_block>
     </execution>
   </experiment>
 </project>
</plush>
```

Figure 9.4: An example Plush application description that includes a request for resources from a Shirako broker/authority.

how Jaws uses this interface and Plush to implement a batch scheduler that requests a virtual machine per task. In this section, the users of the Plush shell invoke the `Load` and `CreateResource` commands using the shell.

A sample (simplified) Plush application description appears in Figure 9.4. The top portion of the application description defines the resources needed by the application. The description of the resources includes a resource type, a set of resource attributes, and a lease term for the resource request—in this case, the resource is a virtual machine sliver. The request also specifies the IP address of the Plush service manager. The Plush service manager exports the `CreateResource` via XML-RPC; the Plush master invokes the method to notify the Plush service manager of resource requests. The separation of the Plush master from the Plush service manager is an implementation

detail (*i.e.*, they were developed independently in two different languages). Conceptually, the Plush master is part of the implementation of the Plush service manager.

The Plush master exports XML-RPC methods to add and remove resources to and from its available pool. A Plush service manager includes guest handlers that invoke these methods to notify the Plush master of the addition (on *join*) or removal (on *leave*) of a resource, respectively. The Plush lease handler uses the `onActiveLease` method from Table 4.3 to invoke the Plush master's *run* method once all of the requested resources have been added. The Plush master sends a notification via XML-RPC of application completion or failure to the Plush service manager. Since our architecture uses leases to govern the lifetime of resources the service manager continually extends leases until a completion or failure notification occurs.

9.2.2 Lessons

Plush was originally designed to execute guests on PlanetLab, discussed in the next section. Integrating Plush as a guest allows experimentation with many existing distributed applications already supported by Plush. For example, we used the Plush service manager to request machines and instantiate Bullet applications [102]. The key point underlying the integration of Plush is that it provides a common interface for adaptive applications that is compatible with both our architecture and with PlanetLab. In the case of PlanetLab, Plush masks resource allocation and revocation from the applications running above it to hide network and host failures. In our architecture, Plush masks the visible allocation and revocation of resources to support existing applications that use Plush without modifying them.

9.3 Planetlab

PlanetLab is a platform for developing networked guests. As with our architecture, its focus is on guests that execute on resources owned by multiple spheres of authority. At the time of this writing PlanetLab consists of over 700 machines from 379 distinct clusters located on 6 continents. The PlanetLab Consortium acts as a brokering service that establishes relationships with sites spread across the world. To join the consortium, sites permanently relinquish control of at least two machines by registering them with PlanetLab Central (PLC) and installing a PlanetLab-specific

operating system and disk image. Users at member sites may then access these machines, subject to PlanetLab policies, to test and deploy new networked guests on a global scale.

Our concern is not with the programming models and features of PlanetLab, but with the core architectural choices for managing hardware resources and trust. PlanetLab's design multiplexes the resources of a collection of machines, which it controls and images centrally. Much of the common API is provided by a Linux kernel flavor mandated by PlanetLab. The programming abstractions are node operating system abstractions: local file name space, processes, and a UNIX/`ssh` security model with keys controlled by PLC. Much of the research focus has been on extending the operating system to virtualize these programming abstractions to isolate multiple virtual servers running on the same physical server [23]. PlanetLab Central permits users to log on to a web site and select machines to add to their PlanetLab slice—a distributed set of VServer virtual machines.

9.3.1 Integration

We develop a PlanetLab service manager that requests machines, on behalf of PlanetLab Central, from a COD authority and dynamically inserts them into PlanetLab. The integration serves to show that the architecture is powerful enough to host an adaptive PlanetLab that provides resource management functionality at a lower layer. The PlanetLab service manager uses MyPLC, a downloadable "PlanetLab-in-a-box" for creating new PlanetLab instances. We completed the integration to run PlanetLab kernels on Xen virtual machines. This required minor modifications to the PlanetLab boot process along with a patch to enable *kexec* in Xen-capable kernels.

PlanetLab employs lease groups from Section 4.3.2 to ensure that the MyPLC central server is instantiated before additional machines join the testbed. The *join* handler for each PlanetLab host communicates with the MyPLC web server using an XML-RPC API to programmatically add and remove machines from the MyPLC instance. The same system is applicable to the public PlanetLab.

PlanetLab is an example of a guest that requires hardware management services to control the booting of a custom disk image. Additionally, the process of adding a new machine to PlanetLab requires that a service manager obtain the network information for the new machine and notify PlanetLab of the information before initiating the boot process. Our current prototype uses a custom authority resource handler to upload network information to PlanetLab before starting the bootup process.

A PlanetLab service manager also requires logic to determine when to request new resources, how much to request, and for how long. The request policy may monitor the testbed (*i.e.*, using CoMon [127] or SWORD [124]) to detect when resources are scarce, and proactively add machines to satisfy user demand. In our prototype, we combine a static policy that requests a fixed number of machines every lease interval with a priority-based arbitration policy that gives PlanetLab low priority in order to shift idle resources to PlanetLab. PlanetLab may provide an interface through their web site for users to log their desire for more resources in order to provide hints about resource requirements to better guide request policy. Adding machines to PlanetLab may prevent periods of high resource contention without requiring PlanetLab to be unnecessarily over-provisioned. PlanetLab network services will benefit, indirectly, if they can adapt to use less-loaded machines as they become available.

9.3.2 Discussion

Architecturally, PlanetLab has made similar choices to Mach, an example of a microkernel structure (see Section 2.1.1). Since it exports node operating system abstractions, PlanetLab is able to support a wide range of hosted environments, including middleware guests such as Globus. PlanetLab has taken the first step of extensibility in progression from microkernels to exokernels with its emphasis on "unbundled management" of infrastructure services. Unbundled management defines key system interfaces to enable alternative implementations of foundational infrastructure services outside of the system's trusted core. The choice enables evolution, and a competitive market for extensions.

But like Mach, PlanetLab retains basic resource management in the core and does not expose its resource allocation choices or allow significant control over policy. It unbundles some resource management functions to subsystems, only with the consent of the central point of trust. For example, a contributing site cannot change the resource allocation policy for its own resources without the consent of the PlanetLab Consortium. It combines programming abstractions and resource management: the abstractions are easy to program with, but resource allocation is built into the implementation of those abstractions.

PlanetLab established a "best-effort" resource management model as a fundamental architectural choice. At the earliest meetings it was observed that any stronger model requires admission control, which is a barrier to use. PlanetLab research has argued that it is a good choice for PlanetLab

because it is underprovisioned, claiming that conservative resource allocation is undesirable because it wastes resources, and exposing hardware-level machine abstractions or multiple operating system instances (*e.g.*, using Xen instead of vservers) consumes too much memory. It has also been stated that the "best-effort" model mirrors the reality of the current Internet, in which edge resources and transit are unreliable. A "best-effort" model forces guests to evolve the means to adapt reactively to whatever confronts them, which is a good skill for any long-running network service to have in a dangerous world.

Guest Adaptation

The "best-effort" philosophy is also in keeping with Butler Lampson's hints to keep it simple and keep secrets [106] (*i.e.*, don't make promises to guests that you might not keep). But at the same time PlanetLab users do have expectations about stability, which has been presented as an explicit goal [130]. As one example, it is considered bad manners to withdraw a machine from PlanetLab without warning; however, PlanetLab's abstractions do not provide a means to visibly deliver such a warning other than to broadcast on an email list. In Section 9.2, we discuss how Plush masks allocation and revocation on PlanetLab from adaptive applications.

In addition to complicating maintenance, the lack of visible notifications discourages sites from contributing resources to PlanetLab on a temporary basis. PlanetLab has added programmatic APIs to contribute and withdraw machines, but warned against their casual use. More importantly, while "best effort" is a reasonable policy choice and should not be excluded, it is too limited as a basis for a general model for sharing networked resources. Some guests require predictable service quality, which must be supported "at the bottom or not at all [51]."

The architectural choices of PlanetLab limit the capabilities of the guests that use the infrastructure since guests have no ability to reserve resources or receive notifications when their resource allotments change. PlanetLab does not allow contributing sites to control when, how much, and how long resources are made available to the testbed, or provide guests a primitive to configure resources for new uses by instantiating a guest-specific software stack. For example, PlanetLab does not export enough control over the software stack necessary to implement guests, such as Jaws (see Section 9.4), that instantiate and control virtual machines.

PlanetLab Adaptation

Our design encourages further participation in PlanetLab by enabling sites to dynamically donate and withdraw their machines as local conditions dictate, reflecting the view that large-scale, sustainable federation will only occur when sites are not forced to permanently relinquish their machines, and are not bound to a specific software stack.

PlanetLab does exhibit the key elements of a system that can serve as the "narrow waist" of a future Internet-scale operating system. For example, debates over different policies that define "best-effort" resource shares and isolated resource shares obscure the lack of a fundamental design principle: the architectural choice PlanetLab makes to mask resource allocation, modification, and revocation. This architectural choice is an obstacle to some guests that PlanetLab aims to support, such as network experiments that must know their resource allotment to ensure repeatable results [24].

Debates over the specific type of virtualization or resource isolation technology (*e.g.*, Vservers vs. Xen) are orthogonal to the issue of defining the principles for an Internet-scale operating system architecture that is compatible with all types of virtualization and isolation technologies. The lease abstraction and design principles presented in this thesis are compatible with any type of virtualization or isolation technology because they only define the single aspect common to all forms of resource sharing: guests share resources and do not use them forever. As a result, the architecture applies to any guest or resource, including PlanetLab.

A recent paper describing lessons from the PlanetLab experience details nine problems with the current PlanetLab architecture. They include centralizing trust, centralizing resource control, decentralizing management, treating bandwidth as a free resource, providing only best-effort service, restricting the execution environment to Linux, not providing common operating system services (*e.g.*, common file system), continual API change, and an inherent focus on the machine room [14].

A narrow leasing architecture addresses each of these concerns by limiting the architecture to only the set of elements common to all resource sharing platforms. The architecture decentralizes trust, resource control, and management among a network of participating actors. Sites are free to negotiate the terms of their resource contributions, such as available bandwidth and sliver isolation. The leasing architecture is general enough to delegate control of hardware to service managers that

configure their own software environment, Linux or otherwise.

9.3.3 Lessons

The key point of integrating PlanetLab is that we do not have to change PlanetLab to make it an adaptive guest in our architecture. However, an adaptive PlanetLab is unable to notify its applications of resource loss or gain since it has no method for delivering notifications. Plush provides a way to mask these allocation and revocation actions from PlanetLab applications, although our architecture is general enough to allow PlanetLab guests to execute using Plush and our architecture directly, as described in the previous section. The integration with PlanetLab (as well as Plush and GridEngine) supports our hypothesis that focusing narrowly on leasing resources allows our architecture to support a wide range of guests.

9.4 Jaws

We develop Jaws to evaluate the architecture's power to serve as a platform for new guests that leverage the control the architecture offers. Jaws combines virtual computing and resource provisioning with the GridEngine model of task scheduling and execution.

As described in Section 8.4.1, a CardioWave-aware service manager leases machines to execute a single parallel MPI application. Section 8.4.1 also demonstrates a service manager for a guest batch scheduler, discussed further in Section 9.1, that adjusts its resource allotment based on the batch scheduler's load. The advantage of developing a service manager per task is that the service manager is able to tailor its resource adaptation and control policies to the individual needs of the task. In contrast, a traditional batch scheduler, such as GridEngine, does not provide these hooks. However, developing a service manager per task does not take advantage of the commonalities between task scheduling, invocation, and monitoring. Below we discuss Jaws, an approach that combines the advantages of developing a service manager for each task and a batch scheduler that executes a stream of task requests.

Using Shirako's architecture we develop a guest that combines the benefits of both approaches: Jaws is a new type of batch scheduler built to integrate with a Shirako service manager. Jaws exposes a `submit` interface similar to that of typical batch schedulers like GridEngine [75]. However, Jaws

differs from typical batch scheduler middleware in that it requests the resources it uses to execute tasks from brokers and executes each task within an isolated virtual container. We design the Jaws to use a Shirako service manager to lease resources from brokers and authorities for task execution.

Since batch schedulers take a middleware approach to resource sharing they rely on node operating systems to expose control functions. While some batch schedulers offer functions, such as slivering and migration, if the operating system provides them, support for other hardware management services is often lacking. While most existing batch schedulers have yet to take advantage of recent virtual machine support for suspend/resume, migration, and slivering, recent batch schedulers advocate support for virtualization [65, 99, 112, 146] to expose more resource control functions to the scheduler. However, virtualization-aware batch schedulers only apply virtualization and resource control to a system that executes tasks. As a result, these systems are not policy-neutral: they define a programming model based on the scheduling and execution of tasks. In contrast, a Jaws service manager provides a similar programming model using a policy-neutral architecture that also accommodates other programming models, such as PlanetLab.

9.4.1 Design and Prototype

Although it manages task execution, Jaws is not a task scheduler: all functions to schedule and arbitrate shared resources migrate to authorities and brokers. Jaws has access to all resource control functions exposed by the authority on a per-task basis. Decoupling task management, task scheduling, and resource scheduling simplifies the implementation of Jaws since the broker and authority address the underlying trust, authorization, and resource scheduling/arbitration issues.

Jaws executes each task within the customized virtual cluster instantiated by the authority. As opposed to the CardioWave-aware service manager from Section 8.4.1, Jaws interacts with a service manager to coordinate resource requests for multiple tasks by requesting resources from a broker on each task's behalf. When virtual machines for a task are made available to the service manager by the authority, it notifies Jaws, which executes each task.

The core functionality of Jaws is task execution and issuing resource requests to brokers. A Jaws service manager submits requests for resources to a broker for each submitted task: the task description includes a preferred lease length. The service manager requests leases to start

immediately. However, each resource request is `deferrable`, meaning that the broker may queue the request and delay fulfilling it if there are no resources available. As brokers fulfill requests, they issue tickets to the Jaws service manager, which redeems the ticket for a virtual cluster at an authority. Jaws relies on the broker's scheduling of resource leases: in the current implementation, the Jaws service manager associates each task with a block of leased resources and does not attempt to schedule multiple tasks on the same set of resources. Related work explores the relationship between resource scheduling and task scheduling [79]. Jaws associates tasks with a single lease: if a task completes before the lease expires then Jaws does not attempt to schedule queued tasks on the existing lease. We discuss the impact of this in Section 9.4.2.

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. Jaws then notifies the service manager of the request and tags the request with the port of the Plush master, its IP address, and the location of the Plush description for the task. In the prototype, Jaws and its service manager run inside the same JVM and use local procedure calls to pass this information. However, in general, a service manager is free to implement any interface to communicate with third-party services, such as Jaws. For Jaws, the service manager interface includes the `queueRequest` function, which Jaws uses to notify the service manager of a resource request.

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. Each *join* for the Jaws-aware service manager adds a machine to the Plush master instance managing the task's execution using the Plush master's XML-RPC interface (see Section 9.2). When the Jaws service manager finishes executing *join* for each resource a lease event handler fires (`onActiveLease` from Table 4.3) and triggers the Plush master to begin task execution (the `Run` command from Section 9.2). On receiving a callback that signals task completion or failure from the Plush master, Jaws notifies its service manager to not renew lease for a task's resources; in the prototype, the notification occurs through a shared variable that triggers the renewal. Jaws uses the `onExtendTicket` method of the guest lease handler to determine whether or not to renew a lease—Jaws only renews leases if the task has not completed.

Guest	*Description*
GridEngine	A prototypical batch scheduler that schedules and executes tasks.
Cardiowave	A parallel MPI application that simulates the electromagnetic pulses of the heart.
Plush	A distributed application manager for applications in volatile, wide-area networks.
PlanetLab	A network testbed for developing new global-scale applications.
Jaws	A new type of batch scheduler built to leverage Shirako's resource control mechanisms.

Table 9.2: A summary of the different guests that we have integrated into Shirako as service managers. Integration enables a level of adaptation and resource control not available to each platform, itself. In addition to these guests, others have used the architecture to integrate the Rubis multi-tier web application [185] and Globus grids [138].

9.4.2 Lessons

A drawback of a batch scheduler that leases resources is the potential for resource waste due to over-estimating task duration and task resource requirements [109]. Recent work shows promise for providing better estimation of task duration and resource profiles [151]. In our prototype, the Jaws service manager requests, and continually renews, short leases to ensure minimal waste after a task's completion; additionally, we offload the setting of the resource profile (*e.g.*, the sliver of CPU, memory, and I/O bandwidth) to the user submitting the task. Jaws is deployed and has been actively used by researchers to conduct large-scale experiments that study the effect of different resource profiles and software configurations on tasks ranging from web application to databases to scientific applications [150, 151].

Jaws is an important example of the power of the neutrality principle in separating programming abstractions from resource multiplexing. As a result of the separation we were able to deploy a minimal batch scheduler that only managed the execution of a task, and not the resources each task executes on, showing that the architecture is general enough to meet the evolving needs of batch schedulers in a direct, but general, way that does not constrain other guests, resources, or usage scenarios.

9.5 Summary

Table 9.2 shows the different guests that we have integrated with Shirako; the guests represent a cross-section of the multitude of different software environments used today to manage resources for networked collections of machines, including batch scheduling systems, multi-tier web applications, network testbeds, and distributed execution environments. The leasing architecture enhances each

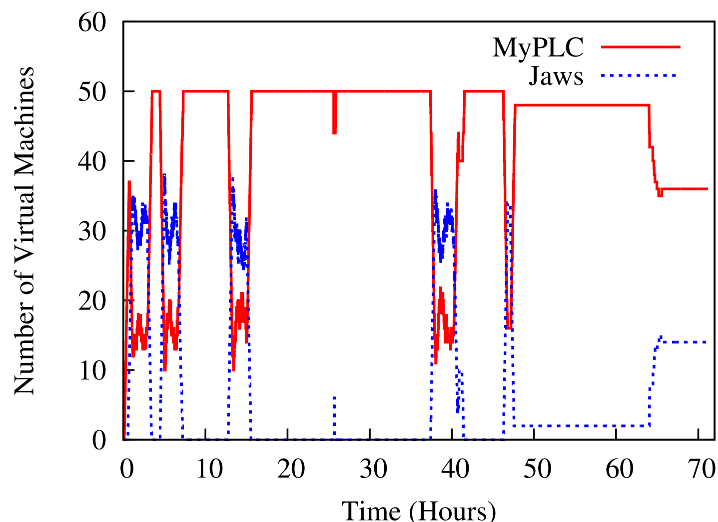

Figure 9.5: MyPLC and Jaws obtaining Xen virtual machines from the Shirako prototype. The prototype manages sharing of hardware resources between MyPLC and Jaws with a simple policy: MyPLC gets any resources not used by Jaws.

of these guests by allowing them to adapt more precisely to load and provides a foundation for new guests, such as Jaws.

Importantly, our experiences from the integration of each guest shaped the design of our architecture over time. We used each guest as a test to determine the requirements of the leasing architecture, and evaluate and stress the architecture's different elements. To illustrate the architecture's role in resource sharing we crafted a simple demonstration using instances of MyPLC and Jaws on a shared set of machines. PlanetLab is not sufficient as a foundation for Jaws: resources can neither be isolated nor scheduled. As shown in related work on active learning, Jaws requires strong resource isolation to learn accurate performance models [150, 151].

Figure 9.5 depicts the number of allocated to a MyPLC-aware service manager and a Jaws-aware service manager in the days leading up to a deadline. Jaws users submitted a few bursts of tasks to collect data for an impending deadline, and received priority to reclaim machines from MyPLC as their leases expire, according to local policy. The top line of Figure 9.5 shows the number of machines allocated to MyPLC over time. Note that the graph shows only the number of machines, not the resources bound to them—for this experiment we used 50 physical machines and were able to allow only one virtual machine per physical machine due to Jaws and PlanetLab's heavy use of local disk space. The key points from this demonstration are that (1) PlanetLab is not a sufficient platform

for all environments—especially those requiring strict resource isolation—and (2) we were able to, relatively easily, port PlanetLab's platform to use our architecture to share hardware resources with a different platform.

Chapter 10

Final Thoughts

"Insanity: doing the same thing over and over again and expecting different results."

Albert Einstein

This thesis presents an operating system architecture for networked server infrastructure that is policy-neutral. The architecture uses a lease abstraction as the basis for managing aggregations of resources under the control of multiple spheres of authority. We extract the design principles for such a networked operating system and relate them to previous work on node operating systems. We validate our hypothesis by prototyping and deploying the system and using it to manage a range of different resources and guests. We conclude by summarizing our contributions and discussing future work.

10.1 Contributions

The hypothesis of this thesis is that an operating system architecture for networked server infrastructure that focuses narrowly on leasing control of hardware to guests provides a single programming abstraction that is a foundation for multi-lateral resource negotiation, arbitration, and fault tolerance.

- Chapter 2 introduces a set of design principles for networked operating systems and relates these design principles to Exokernel, which argues for a clean separation of programming abstraction and resource multiplexing in node operating systems. We describe how our approach to managing networked server infrastructures is similar to the design of Exokernel [61], which allows node operating systems to extend their functionality by reducing the functions of the operating system to resource multiplexing. We show how Exokernel's motivation and design principles are similar to those of a networked operating system. We also summarize the differences between node operating systems and networked operating systems.

- Chapter 3 details different system designs for aggregate resource management that have been proposed. Our architecture differs from previous systems for managing resource aggregations in its focus on exposing, rather than hiding, the capabilities of the underlying hardware. The most popular systems in 2007 for managing shared networked resources fall into a class of middleware systems that hide the underlying complexity of managing hardware by restricting the way in which users and applications access resources—for example, Grid middleware simplifies resource management in practice by allowing users to submit jobs that execute for some time and then terminate.

- Chapter 4 describes the architecture of a prototype networked operating system that combines the idea of a reconfigurable data center from Cluster-on-Demand [39] and others [5, 15, 17, 97, 99, 116, 144, 177] with the SHARP framework [73] for secure resource peering. We design, implement, and deploy the architecture using a single programming abstraction—the lease—and identify three design principles necessary to be policy-neutral: guest/resource independence, visible allocation, modification, and revocation, and an abort protocol. We then describe the integration of Cluster-on-Demand that leverages these principles to multiplex machines in a data center.

- Chapter 5 identifies shortcomings in the original conception of SHARP and Cluster-on-Demand that make them unsuitable for multiplexing aggregations of slivers bound to virtualized hardware. We extend the SHARP and Cluster-on-Demand model using two principles: exposing names and secure bindings. Exposing names augments SHARP tickets to include logical names for hosts, virtualized hardware, and slivers that allow service managers and authorities to reference each entity separately. Secure bindings permit an authority or service manager to access the hardware management services of each virtual hardware device. We describe control mechanisms that exposing names and secure bindings enable for guests.

- Chapter 6 demonstrates that leases are a foundational primitive for addressing arbitration in a networked operating system that supports both market-based and proportional-share arbitration policies. Leasing currency defines a configurable tradeoff between proportional-share scheduling and a market economy. We demonstrate the properties of self-recharging virtual currency using simulations of two market-based task services.

- Chapter 7 outlines the architecture's model for addressing failures, which combines the use of leases for long-term resource management with state recovery mechanisms. The model provides system robustness to transient faults and failures in a loosely coupled distributed system that coordinates resource allocation.

- Chapter 8 describes the implementation of a prototype and evaluates its flexibility and performance. We asses the prototype's flexibility by multiplexing storage devices, physical machines, and virtual machines in a data center, and evaluate its scalability using emulation experiments that stress the prototype under saturation. We find that the design principle of resource independence does not preclude the type of resource multiplexed by the prototype and that performance is suitable for managing thousands of machines.

- Chapter 9 further evaluates the architecture by discussing case studies of integrating multiple types of guests including the PlanetLab network testbed [23], the Plush distributed application manager [9], and the GridEngine batch scheduler [75]. Each guest requires a different set of capabilities for controlling resources and policies for adapting its resources to changes in load. The integrations leverage the design principles of guest independence, visible allocation, modification, and revocation, and an abort protocol. We also leverage the architecture to construct Jaws, a light-weight batch scheduler that instantiates one or more virtual machines per task.

In summary, we validate our hypothesis by examining previous work in a different domain, distilling a set of design principles that accounts for previous work and incorporates the characteristics of a distributed environment, describing the impact on existing systems that deviate from these design principles, constructing a prototype of the architecture and using it to implement Cluster-on-Demand, extending the prototype and its principles to control aggregations of virtualized resources, and showing that leases are a basis for arbitration and fault tolerance. We evaluate the system by studying the impact of the architecture on multiple types of guests and resources that we integrate with it.

10.2 Future Directions

The GENI initiative represents a growing consensus in the network research community that the Internet needs drastic changes to continue to evolve to support new innovations from academic and industry research [131]. The initiative is a product of previous lessons learned trying to inject new functionality into the existing Internet: both Active Networks [174] and overlay networks [91] inject new functionality by allowing external code to execute on internal network routers or layering new network functionality on top of the existing Internet, respectively. GENI combines elements of both approaches by proposing an Internet architecture founded on virtualization and slivering as the primary means to isolate different guests and networks in the wide-area. Slivering of both the edge and internal resources that comprise the Internet affords a flexible design that accommodates existing Internet packet routing functionality as well as future innovations by isolating them in virtual slices of a networked infrastructure [132].

The impact of applying core principles of node operating system design to a networked operating system, as proposed by GENI, is left for the future. However, we can observe the impact of not applying these principles on the operation of GENI's precursor, PlanetLab. As one example, PlanetLab does not use leases for resource management; however, it has found it necessary to use leases for the lifetime management of slices, which expire if unused for more than 8 weeks. Lifetime management of slices is necessary for PlanetLab because each individual machine associates a minimal amount of state with each slice.

A networked operating system must wrestle with many of the same questions that node operating system designers wrestled with: namely, what are the right programming abstractions for networked computing? Years of operating systems research produced a common set of basic programming abstractions implemented by nearly all node operating systems including files, processes, threads, pipes, sockets, and virtual memory. Even so, the implementation of these most basic, widely used, and fundamental abstractions came under scrutiny in research on extensible operating systems [13, 27, 61].

In the end, the research did not have a significant impact on commodity node operating systems because machine resources became cheap enough to partition at a coarse grain using technologies such as virtual machines. Instead, node operating systems developed to a point where users were

comfortable with the programming abstractions they provided (possibly because they *had* to become comfortable with them because there were no other options). The model for scalable distributed computing is still being determined. Previous work on different types of distributed operating systems did not gain wide acceptance [4, 6, 11, 21, 30, 41, 53, 55, 66, 76, 94, 104, 113, 123, 137, 140, 160, 169, 171] because they did not cleanly separated programming abstractions from mechanisms for multiplexing resources. For instance, the systems cited above combine different elements of remote execution, migration, checkpointing, task scheduling and control, parallel job execution, and idle resource detection. While useful, the implementation of each of these mechanisms assumes the guests that use the resources will actually use the mechanism.

The architecture in this thesis seeks to provide an extensible platform for higher-level platforms and services to use resources in any way they choose. We choose to adopt principles similar to those of Exokernel [61] by focusing on implementing servers that securely multiplex resources but do not implement programming abstractions. The architecture eliminates management and arbitration policy from the authorities that control resources. The architectural decision to delegate resource management to guests frees the architecture to serve as the foundation for any type of guest, as long as the guest accommodates visible resource allocation and revocation.

We focus on multiplexing networked collections of physical and virtual machines in data centers. Virtual machines use local scheduling policies to sliver machines resources and allocate them at a coarser grain than Exokernel. This thesis does not define the local resource scheduling policies for slivers. High fidelity slivering of physical machine resources, including multi-core processors, is ongoing research that is orthogonal to our architecture. The architecture is compatible with any slivering technology that partitions the resources of physical hardware and presents a machine-level abstraction to guests.

The principle of secure low-level resource multiplexing is also applicable to other environments. In particular, applying these principles to network virtualization and isolation is an open problem; virtual clusters do not currently isolate each virtual cluster's network traffic. Virtual clusters with resources that span multiple physical locations also cannot control their own network namespace, and, given proper virtualization at the routers, request bandwidth-provisioned network paths to enable multiple, self-contained networks that can better control how their traffic interacts with the

broader Internet. Applying the same principles to more resource constrained environments, such as mobile devices and sensors, is left for future work.

This thesis focuses on an architecture for hardware resources that guests may reuse. We view other characterizations of a resource, such as energy [186], as policy constraints placed on the use of a renewable hardware resource. For example, an authority for a data center that must adhere to a strict daily power budget must have a policy that leases power-hungry hardware to guests in order to meet the daily budget [139]. The policy implications of energy constraints are also paramount for battery-powered mobile devices and sensors: the design principles of our architecture are independent, but compatible, with these policy considerations. However, further investigation of the policy considerations for energy is left for future work.

Designing new guests and policies that leverage the architecture's foundational capabilities for resource multiplexing in new domains is continuing future work. As stated above, simplifying programming abstractions reduces the burden of developing new guests; however, programming abstractions are especially necessary in networked environments that exhibit volatile behavior. Node operating systems present a variety of useful programming abstractions to the programmer including processes, threads, files, pipes, and virtual memory. Designing useful programming abstractions for guests executing on machines spread across networks, such as the Internet, that are as useful to programmers as node programming abstractions is an important next step in the evolution of distributed computing.

Bibliography

[1] http://www.opsware.com, Accessed October 2007.

[2] http://www.wrhambrecht.com/ind/auctions/openbook/index.html, Accessed February 2008.

[3] Who Wants to Buy a Computon? In *The Economist*, March 12, 2005.

[4] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In Proceedings of the *USENIX Summer Technical Conference*, pages 93-113, Atlanta, Georgia, June 1986.

[5] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, Jose Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu and Xiaomin Zhu. From Virtualized Resources to Virtual Computing Grids: The in-VIGO System. In *Future Generation Computing Systems*, 21(1):896-909, January 2005.

[6] Rakesh Agrawal and Ahmed K. Ezzat. Location Independent Remote Execution in NEST. In *IEEE Transactions on Software Engineering.*, 13(8):905-912, August 1987.

[7] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In Proceedings of the *Symposium on Operating Systems Principles*, pages 74-89, Bolton Landing, New York, October 2003.

[8] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren and Amin Vahdat. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In Proceedings of the *Large Installation System Administration Conference*, Dallas, Texas, November 2007.

[9] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren and Amin Vahdat. PlanetLab Application Management Using Plush. In *SIGOPS Operating Systems Review*, 40(1):33-40, January 2006.

[10] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren and Amin Vahdat. Loose Synchronization for Large-Scale Networked Systems. In Proceedings of the *USENIX Annual Technical Conference*, Boston, Massachusetts, June 2006.

[11] Guy T. Almes, Andrew P. Black, Edward D. Lazowska and Jerre D. Noe. The Eden System: A Technical Review. In *IEEE Transactions on Software Engineering.*, 11(1):43-59, January 1985.

[12] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek and Robert Morris. Resilient Overlay Networks. In Proceedings of the *Symposium on Operating Systems Principles*, pages 131-145, Banff, Canada, October 2001.

[13] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In Proceedings of the *Symposium on Operating Systems Principles*, pages 95-109, Pacific Grove, California, October 1991.

[14] Thomas Anderson and Timothy Roscoe. Lessons from PlanetLab. In Proceedings of the *Workshop on Real, Large Distributed Systems*, Seattle, Washington, November 2006.

[15] Karen Appleby, Sameh Fakhouri, Liana Fong, Germán Goldszmidt, Michael Kalantar, Srirama Krishnakumar, Donald P. Pazel, John Pershing and Benny Rochwerger. Oceano-SLA Based Management of a Computing Utility. In Proceedings of the *International Symposium on Integrated Network Management*, pages 855-858, Seattle, Washington, May 2001.

[16] Alvin AuYoung, Laura Grit, Janet Wiener and John Wilkes. Service Contracts and Aggregate Utility Functions. In Proceedings of the *IEEE Symposium on High Performance Distributed Computing*, pages 119-131, Paris, France, June 2006.

[17] Sam Averitt, Michael Bugaev, Aaron Peeler, Henry Shaffer, Eric Sills, Sarah Stein, Josh Thompson and Mladen Vouk. Virtual Computing Laboratory (VCL). In Proceedings of the *International Conference on the Virtual Computing Initiative*, pages 1-6, Research Triangle Park, North Carolina, May 2007.

[18] Ivan D. Baev, Waleed M. Meleis and Alexandre E. Eichenberger. Algorithms for Total Weighted Completion Time Scheduling. In *Algorithmica*, 33(1):34–51, May 2002.

[19] Magdalena Balazinska, Hari Balakrishnan and Michael Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In Proceedings of the *Symposium on Networked System Design and Implementation*, pages 197-210, San Francisco, California, March 2004.

[20] Gaurav Banga, Peter Druschel and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 45-58, New Orleans, Louisiana, February 1999.

[21] Antônio Marinho Pilla Barcellos, João Frederico Lacava Schramm, Valdir Rossi Belmonte Filho and Cláudio F. R. Geyer. The HetNOS Network Operating System: A Tool for Writing Distributed Applications. In *SIGOPS Operating Systems Review*, 28(4):34-47, October 1994.

[22] Paul T. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield. Xen and the Art of Virtualization. In Proceedings of the *Symposium on Operating Systems Principles*, pages 164-177, Bolton Landing, New York, October 2003.

[23] Andy C. Bavier, Mic Bowman, Brent N. Chun, David E. Culler, Scott Karlin, Steve Muir, Larry L. Peterson, Timothy Roscoe, Tammo Spalink and Mike Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In Proceedings of the *Symposium on Networked System Design and Implementation*, pages 253-266, San Francisco, California, March 2004.

[24] Andy C. Bavier, Nick Feamster, Mark Huang, Larry L. Peterson and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In Proceedings of the *SIGCOMM Conference*, pages 3-14, Piza, Italy, September 2006.

[25] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the *USENIX Annual Technical Conference, FREENIX Track*, pages 41-46, Anaheim, California, April 2005.

[26] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau and Miron Livny. Explicit Control in the Batch-Aware Distributed File System. In Proceedings of the *Symposium on Networked System Design and Implementation*, pages 365-378, San Francisco, California, March 2004.

[27] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers and Susan J. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In Proceedings of the *Symposium on Operating Systems Principles*, pages 267-284, Copper Mountain, Colorado, December 1995.

[28] Andrew Birrell, Greg Nelson, Susan S. Owicki and Edward Wobber. Network Objects. In Proceedings of the *Symposium on Operating Systems Principles*, pages 217-230, Asheville, North Carolina, October 1993.

[29] Rebecca Braynard, Dejan Kostić, Adolfo Rodriguez, Jeff Chase and Amin Vahdat. Opus: An Overlay Peer Utility Service. In Proceedings of the *Open Architectures and Network Programming*, pages 167-178, New York, New York, June 2002.

[30] David R. Brownbridge, Lindsay F. Marshall and Brian Randell. The Newcastle Connection Or UNIXes of the World Unite. In *Classic Operating Systems: From Batch Processing to Distributed Systems*, pages 528-549, Springer-Verlag New York, Inc. New York, New York, 2001.

[31] Navin Budhiraja, Keith Marzullo, Fred B. Schneider and Sam Toueg. The Primary-Backup Approach. In *Distributed Systems*, pages 199-216, ACM Press/Addison-Wesley Publishing Co. New York, New York, 1993.

[32] Jennifer Burge, Partha Ranganathan and Janet Wiener. Cost-Aware Scheduling for Heterogeneous Enterprise Machines (CASH'EM). HP Laboratories, Technical Report HPL-2007-63, May 2007.

[33] Alan Burns, Divya Prasad, Andrea Bondavalli, Felicita Di Giandomenico, Krithi Ramamritham, John Stankovic and Lorenzo Strigini. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. In *Journal of Systems Architecture*, 46(4):305-325, January 2000.

[34] Rajkumar Buyya, David Abramson and Jonathan Giddy. An Economy Driven Resource Management Architecture for Global Computational Power Grids. In Proceedings of the *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, North Dakota, 2000.

[35] Zhongtang Cai, Vibhore Kumar and Karsten Schwan. IQ-Paths: Self-Regulating Data Streams Across Network Overlays. In Proceedings of the *IEEE Symposium on High Performance Distributed Computing*, Paris, France, June 2006.

[36] Emmanuel Cecchet, Julie Marguerite and Willy Zwaenepoel. Performance and Scalability of EJB Applications. In Proceedings of the *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246-261, Seattle, Washington, November 2002.

[37] Clovis Chapman, Paul Wilson, Todd Tannenbaum, Matthew Farrellee, Miron Livny, John Brodholt and Wolfgang Emmerich. Condor Services for the Global Grid: Interoperability Between Condor and OGSA. In Proceedings of the *2004 UK e-Science All Hands Meeting*, pages 870-877, Nottingham, United Kingdom, August 2004.

[38] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centres. In Proceedings of the *Symposium on Operating Systems Principles*, pages 103-116, Banff, Canada, October 2001.

[39] Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore and Sara E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In Proceedings of the *IEEE Symposium on High Performance Distributed Computing*, pages 90-100, Seattle, Washington, June 2003.

[40] Ken Chen and Paul Muhlethaler. A Scheduling Algorithm for Tasks Described By Time Value Function. In *Real-Time Systems*, 10(3):293-312, 1996.

[41] David R. Cheriton and Willy Zwaenepoel. The Distributed V Kernel and Its Performance for Diskless Workstations. In Proceedings of the *Symposium on Operating Systems Principles*, pages 129-140, Bretton Woods, New Hampshire, October 1983.

[42] Brent Chun. Market-Based Cluster Resource Management. Ph.D. Thesis, University of California at Berkeley, November 2001.

[43] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren and Amin Vahdat. Mirage: A Microeconomic Resource Allocation System for SensorNet Testbeds. In Proceedings of the *Workshop on Embedded Networked Sensors*, pages 19-28, Sydney, Australia, May 2005.

[44] Brent N. Chun and David E. Culler. User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers. In Proceedings of the *IEEE International Symposium on Cluster Computing and the Grid*, pages 30-38, Berlin, Germany, May 2002.

[45] Brent N. Chun, Yun Fu and Amin Vahdat. Bootstrapping a Distributed Computational Economy with Peer-to-Peer Bartering. In Proceedings of the *Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, California, June 2003.

[46] Ed G. Coffman, Jr., Mike R. Garey and David S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In *Approximation algorithms for NP-hard problems*, pages 46-93, 1996.

[47] Bram Cohen. Incentives Build Robustness in BitTorrent. In Proceedings of the *Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, California, June 2003.

[48] Landon P. Cox and Peter Chen. Pocket Hypervisors: Opportunities and Challenges. In Proceedings of the *Workshop on Mobile Computing Systems and Applications*, Tuscon, February 2007.

[49] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini and Willy Zwaenepoel. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In Proceedings of the *Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, October 2007.

[50] Robert J. Creasy. The Origin of the VM/370 Time-Sharing System. In *IBM Journal of Research and Development*, 25(5):483-490, September 1981.

[51] Jon Crowcroft, Steven Hand, Richard Mortier, Timothy Roscoe and Andrew Warfield. QoS's Downfall: At the Bottom, Or Not At All! In Proceedings of the *Workshop on Revisiting IP QoS: What Have We Learned, Why Do We Care?*, pages 109-114, Karlsruhe, Germany, August 2003.

[52] Jon Crowcroft, Steven Hand, Richard Mortier, Timothy Roscoe and Andrew Warfield. Plutarch: An Argument for Network Pluralism. In *SIGCOMM Computer Communications Review*, 33(4):258 - 266, October 2003.

[53] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad and Umakishore Ramachandran. The Clouds Distributed Operating System. In *Computer*, 24(11):34-44, November 1991.

[54] Jeff Dike. User-Mode Linux. In Proceedings of the *Linux Showcase and Conference*, Oakland, California, November 2001.

[55] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. In *Software-Practice and Experience*, 21(8):757-785, August 1991.

[56] Peter Druschel, Vivek S. Pai and Willy Zwaenepoel. Extensible Kernels Are Leading OS Research Astray. In Proceedings of the *Workshop on Hot Topics in Operating Systems*, pages 38-42, Cape Cod, Massachusetts, May 1997.

[57] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 211-224, Boston, Massachusetts, December 2002.

[58] Eric Eide, Leigh Stoller and Jay Lepreau. An Experimentation Workbench for Replayable Networking Research. In Proceedings of the *Symposium on Networked System Design and Implementation*, Cambridge, Massachusetts, April 2007.

[59] Dawson R. Engler. The Exokernel Operating System Architecture. Ph.D. Thesis, Massachusetts Institute of Technology, October 1998.

[60] Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating Systems Abstractions. In Proceedings of the *Workshop on Hot Topics in Operating Systems*, pages 78-85, Orcas Island, Washington, 1995.

[61] Dawson R. Engler, M. Frans Kaashoek and James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Proceedings of the *Symposium on Operating Systems Principles*, pages 251-266, Copper Mountain, Colorado, December 1995.

[62] Nick Feamster, Lixin Gao and Jennifer Rexford. How to Lease the Internet in Your Spare Time. In *SIGCOMM Computer Communications Review*, 37(1):61-64, January 2007.

[63] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In Proceedings of the *Symposium on Operating Systems Principles*, pages 201-212, Copper Mountain, Colorado, December 1995.

[64] Michal Feldman, Kevin Lai and Li Zhang. A Price-Anticipating Resource Allocation Mechanism for Distributed Shared Clusters. In Proceedings of the *ACM Conference on Electronic Commerce*, pages 127-136, Vancouver, Canada, June 2005.

[65] Renato J. O. Figueiredo, Peter A. Dinda and José A. B. Fortes. A Case for Grid Computing on Virtual Machine. In Proceedings of the *International Conference on Distributed Computing Systems*, pages 550-559, Providence, Rhode Island, May 2003.

[66] Raphael Finkel, Michael L. Scott, Yeshayahu Artsy and Hung-Yang Chang. Experience with Charlotte: Simplicity and Function in a Distributed Operating System. In *IEEE Transactions on Software Engineering.*, 15(6):676-685, June 1989.

[67] Ian Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In Proceedings of the *International Conference on Network and Parallel Computing*, pages 2-13, Beijing, China, December 2005.

[68] Ian Foster, Karl Czajkowski, Donald F. Ferguson, Jeffrey Frey, Steve Graham, Tom Maguire, David Snelling and Steven Tuecke. Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF. In *IEEE*, 93(3):604-612, March 2005.

[69] Ian Foster and Carl Kesselman. The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2004.

[70] Ian Foster, Carl Kesselman, Jeffrey M. Nick and Steven Tuecke. Grid Services for Distributed System Integration. In *Computer*, 35(6):37-46, June 2002.

[71] Michael J. Freedman, Eric Freudenthal and David Mazières. Democratizing Content Publication with Coral. In Proceedings of the *Symposium on Networked System Design and Implementation*, pages 239-252, San Francisco, California, March 2004.

[72] Yun Fu. Resource Allocation for Global-Scale Network Services. Ph.D. Thesis, Duke University, December 2004.

[73] Yun Fu, Jeffrey S. Chase, Brent N. Chun, Stephen Schwab and Amin Vahdat. SHARP: An Architecture for Secure Resource Peering. In Proceedings of the *Symposium on Operating Systems Principles*, pages 133-148, Bolton Landing, New York, October 2003.

[74] Simson Garfinkel. Commodity Grid Computing with Amazon's S3 and EC2. In *;login: The USENIX Magazine*, 32(1):7-13, February 2007.

[75] Wolfgang Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In Proceedings of the *IEEE International Symposium on Cluster Computing and the Grid*, pages 35-36, Chicago, Illinois, April 2004.

[76] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat and Thomas E. Anderson. GLUnix: A Global Layer UNIX for a Network of Workstations. In *Software-Practice and Experience*, 28(9):929-961, July 1998.

[77] Frank E. Gillett and Galen Schreck. Server Virtualization Goes Mainstream. Forrester Research Inc. Technical Report 2-22-06, February 2006.

[78] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In Proceedings of the *Symposium on Operating Systems Principles*, pages 202-210, Litchfield Park, Arizona, December 1989.

[79] Laura Grit. Extensible Resource Management for Networked Virtual Computing. Ph.D. Thesis, Duke University, December 2007.

[80] Laura Grit, Jeff Chase David Irwin and Aydan Yumerefendi. Adaptive Virtual Machine Hosting with Brokers. Duke University, Technical Report CS-2006-12, August 2006.

[81] Laura Grit, David Irwin, Aydan Yumerefendi and Jeff Chase. Virtual Machine Hosting for Networked Clusters: Building the Foundations for "Autonomic" Orchestration. In Proceedings of the *Workshop on Virtualization Technology in Distributed Computing*, Tampa, Florida, November 2006.

[82] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In Proceedings of the *International Middleware Conference*, pages 342-362, Melbourne, Australia, November 2006.

[83] Rohit Gupta and Varun Sekhri. CompuP2P: An Architecture for Internet Computing Using Peer-to-Peer Networks. In *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1306-1320, November 2006.

[84] Joseph Hall, Jason D. Hartline, Anna R. Karlin, Jared Saia and John Wilkes. On Algorithms for Efficient Data Migration. In Proceedings of the *Symposium on Discrete Algorithms*, pages 620-629, Washington, D.C. January 2001.

[85] Steven Hand, Tim Harris, Evangelos Kotsovinos and Ian Pratt. Controlling the XenoServer Open Platform. In Proceedings of the *IEEE Conference on Open Architectures and Network Programming*, pages 3-11, April 2003.

[86] Timothy L. Harris. Extensible Virtual Machines. Ph.D. Thesis, University of Cambridge, December 2001.

[87] Kieran Harty and David R. Cheriton. Application-Controlled Physical Memory Using External Page Cache Management. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems*, pages 187-197, Boston, Massachusetts, October 1992.

[88] Pat Helland. Life Beyond Distributed Transactions: An Apostate's Opinion. In Proceedings of the *Conference on Innovative Data Systems Research*, pages 132-141, Asilomar, California, January 2007.

[89] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci and Chad Barb. Fast, Scalable Disk Imaging with Frisbee. In Proceedings of the *USENIX Annual Technical Conference*, pages 283-296, San Antonio, Texas, June 2003.

[90] David E. Irwin, Laura E. Grit and Jeffrey S. Chase. Balancing Risk and Reward in a Market-Based Task Service. In Proceedings of the *IEEE Symposium on High Performance Distributed Computing*, pages 160-169, Honolulu, Hawaii, June 2004.

[91] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek and James O'Toole, Jr. Overcast: Reliable Multicasting with an Overlay Network. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 197-212, San Diego, California, October 2000.

[92] Xuxian Jiang and Dongyan Xu. SODA: A Service-on-Demand Architecture for Application Service Hosting in Utility Platforms. In Proceedings of the *IEEE Symposium on High Performance Distributed Computing*, pages 174-183, Seattle, Washington, June 2003.

[93] Ashlesha Joshi, Samuel T. King, George W. Dunlap and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In Proceedings of the *Symposium on Operating Systems Principles*, pages 91-104, Brighton, United Kingdom, October 2005.

[94] Jiubin Ju, Gaochao Xu and Jie Tao. Parallel Computing Using Idle Workstations. In *SIGOPS Operating Systems Review*, 27(3):87-96, July 1993.

[95] Chet Juszczak. Improving the Performance and Correctness of an NFS Server. In Proceedings of the *USENIX Winter Technical Conference*, pages 53-63, San Diego, California, February 1989.

[96] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In Proceedings of the *Symposium on Operating Systems Principles*, pages 52-65, Saint Malo, France, October 1997.

[97] Mahesh Kallahalla, Mustafa Uysal, Ram Swaminathan, David E. Lowell, Mike Wray, Tom Christian, Nigel Edwards, Chris I. Dalton and Frederic Gittler. SoftUDC: A Software-Based Data Center for Utility Computing. In *Computer*, 37(11):38-46, November 2004.

[98] A. Karve, Tracy Kimbrel, Giovanni Pacifici, Mike Spreitzer, Malgorzata Steinder, Maxim Sviridenko and Asser N. Tantawi. Dynamic Placement for Clustered Web Applications. In Proceedings of the *International World Wide Web Conference*, pages 595-604, Edinburgh, Scotland, May 2006.

[99] Kate Keahey, Ian Foster, Timothy Freeman, Xuehai Zhang and Daniel Galron. Virtual Workspaces in the Grid. In Proceedings of the *International Euro-Par Conference on Parallel Processing*, pages 421-431, Lisbon, Portugal, September 2005.

[100] Terence Kelly. Utility-Directed Allocation. In Proceedings of the *Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[101] Samuel T. King, George W. Dunlap and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In Proceedings of the *USENIX Annual Technical Conference*, pages 1-11, Anaheim, California, April 2005.

[102] Dejan Kostic, Adolfo Rodriguez, Jeannie R. Albrecht and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In Proceedings of the *Symposium on Operating Systems Principles*, pages 282-297, Bolton Landing, New York, October 2003.

[103] Evangelos Kotsovinos, Tim Moreton, Ian Pratt, Russ Ross, Keir Fraser, Steven Hand and Tim Harris. Global-Scale Service Deployment in the XenoServer Platform. In Proceedings of the *Workshop on Real, Large Distributed Systems*, San Francisco, California, December 2004.

[104] Nancy P. Kronenberg, Henry M. Levy and William D. Strecker. VAXclusters: A Closely-Coupled Distributed System. In *ACM Transactions on Computer Systems*, 4(2):130-146, May 1986.

[105] Kevin Lai, Lars Rasmusson, Eytan Adar, Stephen Sorkin, Li Zhang and Bernardo A. Huberman. Tycoon: An Implemention of a Distributed Market-Based Resource Allocation System. HP Laboratories, Technical Report DC/0412038, December 2004.

[106] Butler W. Lampson. Hints for Computer System Design. In Proceedings of the *Symposium on Operating Systems Principles*, pages 33-48, Bretton Woods, New Hampshire, October 1983.

[107] Butler W. Lampson. How to Build a Highly Available System Using Consensus. In Proceedings of the *International Workshop on Distributed Algorithms*, pages 1-17, Bologna, Italy, October 1996.

[108] Butler W. Lampson and Robert F. Sproull. An Open Operating System for a Single-User Machine. In Proceedings of the *Symposium on Operating Systems Principles*, pages 98-105, Pacific Grove, California, December 1979.

[109] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy and Allan Snavely. Are User Runtime Estimates Inherently Inaccurate? In Proceedings of the *International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 253-263, New York, New York, June 2004.

[110] Philip Levis and David Culler. MatÉ: A Tiny Virtual Machine for Sensor Networks. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems*, pages 85-95, October 2002.

[111] Jochen Liedtke. On Micro-Kernel Construction. In Proceedings of the *Symposium on Operating Systems Principles*, pages 237-250, Copper Mountain, Colorado, December 1995.

[112] Bin Lin and Peter A. Dinda. VSched: Mixing Batch and Interactive Virtual Machines Using Periodic Real-Time Scheduling. In Proceedings of the *ACM/IEEE Conference on Supercomputing*, Seattle, Washington, November 2005.

[113] Michael J. Litzkow, Miron Livny and Matt W. Mutka. Condor - a Hunter of Idle Workstations. In Proceedings of the *International Conference on Distributed Computing Systems*, pages 104-111, San Jose, California, June 1988.

[114] Rick Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In Proceedings of the *USENIX Winter Technical Conference*, pages 261-278, San Francisco, California, January 1994.

[115] John Markoff and Saul Hansell. Hiding in Plain Sight, Google Seeks More Power. In *New York Times*, June 14, 2006.

[116] Marvin McNett, Diwaker Gupta, Amin Vahdat and Geoffrey M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In Proceedings of the *Large Installation System Administration Conference*, Dallas, Texas, November 2007.

[117] Aravind Menon, Alan L. Cox and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In Proceedings of the *USENIX Annual Technical Conference*, pages 199-212, Boston, Massachusetts, June 2006.

[118] Jennifer D. Mitchell-Jackson. Energy Needs in an Internet Economy: A Closer Look At Data Centers. Master's Thesis, University of California, Berkeley, May 2001.

[119] Justin Moore, Jeff Chase, Parthasarathy Ranganathan and Ratnesh Sharma. Making Scheduling "Cool": Temperature-Aware Resource Assignment in Data Centers. In Proceedings of the *USENIX Annual Technical Conference*, Anaheim, California, April 2005.

[120] Justin Moore, David Irwin, Laura Grit, Sara Sprenkle and Jeff Chase. Managing Mixed-Use Clusters with Cluster-on-Demand. Duke University, Technical Report CS-2002-11, November 2002.

[121] Ripal Nathuji and Karsten Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In Proceedings of the *Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, October 2007.

[122] Peter Naur and Brian Randall. Software Engineering. In *Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, January 1969.

[123] David A. Nichols. Using Idle Workstations in a Shared Computing Environment. In Proceedings of the *Symposium on Operating Systems Principles*, pages 5-12, Austin, Texas, November 1987.

[124] David Oppenheimer, Jeannie Albrecht, David Patterson and Amin Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In Proceedings of the *IEEE Symposium on High Performance Distributed Computing*, pages 113-124, 2005.

[125] David Oppenheimer, Brent Chun, David Patterson, Alex C. Snoeren and Amin Vahdat. Service Placement in a Shared Wide-Area Platform. In Proceedings of the *USENIX Annual Technical Conference*, Boston, Massachusetts, June 2006.

[126] Philip M. Papadopoulos, Mason J. Katz and Greg Bruno. NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters. In Proceedings of the *IEEE International Conference on Cluster Computing*, pages 258-267, Newport Beach, California, October 2001.

[127] KyoungSoo Park and Vivek S. Pai. CoMon: A Mostly-Scalable Monitoring System for PlanetLab. In *SIGOPS Operating Systems Review*, 40(1):65-74, January 2006.

[128] David C. Parkes. Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency. Ph.D. Thesis, University of Pennslyvania, May 2001.

[129] Larry Peterson, Tom Anderson, David Culler and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology Into the Internet. In Proceedings of the *Workshop on Hot Topics in Networks*, Princeton, New Jersey, November 2002.

[130] Larry Peterson, Andy Bavier, Marc E. Fiuczynski and Steve Muir. Experiences Building PlanetLab. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 351-366, Seattle, Washington, November 2006.

[131] Larry Peterson and John Wroclawski Editors. Overview of the GENI Architecture. GENI Design Document, Technical Report 06-11, September 2006.

[132] Larry Peterson, Scott Shenker and Jon Turner. Overcoming the Internet Impasse through Virtualization. In Proceedings of the *Workshop on Hot Topics in Networks*, San Diego, California, November 2004.

[133] Ben Pfaff, Tal Garfinkel and Mendel Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In Proceedings of the *Symposium on Networked System Design and Implementation*, San Jose, California, May 2006.

[134] Florentina I. Popovici and John Wilkes. Profitable Services in an Uncertain World. In Proceedings of the *ACM/IEEE Conference on Supercomputing*, Seattle, Washington, November 2005.

[135] John B. Pormann, John A. Board, Donald J. Rose and Craig S. Henriquez. Large-Scale Modeling of Cardiac Electrophysiology. In Proceedings of the *Computers in Cardiology*, pages 259-262, September 2002.

[136] Pradeep Pradala, Kang Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant and Kenneth Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In Proceedings of the *European Conference on Computer Systems*, pages 289-302, Lisbon, Portugal, March 2007.

[137] David L. Presotto and Barton P. Miller. Process Migration in DEMOS/MP. In Proceedings of the *Symposium on Operating Systems Principles*, pages 110-119, Bretton Woods, New Hampshire, October 1983.

[138] Lavayna Ramakrishnan, Laura Grit, Adriana Iamnitchi, David Irwin, Aydan Yumerefendi and Jeff Chase. Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control. In Proceedings of the *ACM/IEEE Conference on Supercomputing*, Tampa, Florida, November 2006.

[139] Parthasarathy Ranganathan, Phil Leech, David Irwin and Jeffrey Chase. Ensemble-Level Power Management for Dense Blade Servers. In Proceedings of the *International Symposium on Computer Architecture*, pages 66-77, Boston, Massachusetts, June 2006.

[140] Richard F. Rashid and George G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In Proceedings of the *Symposium on Operating Systems Principles*, pages 64-75, Pacific Grove, California, December 1981.

[141] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In Proceedings of the *Symposium on Networked System Design and Implementation*, San Jose, California, May 2006.

[142] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera and Jay Lepreau. The Flexlab Approach to Realistic Evaluation of Networked Systems. In Proceedings of the *Symposium on Networked System Design and Implementation*, Cambridge, Massachusetts, April 2007.

[143] Eric Robinson and David J. DeWitt. Turning Cluster Management Into Data Management; a System Overview. In Proceedings of the *Conference on Innovative Data Systems Research*, pages 120-131, Asilomar, California, January 2007.

[144] Paul Ruth, Junghwan Rhee, Dongyan Xu, Rick Kennell and Sebastien Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In Proceedings of the *International Conference on Autonomic Computing*, pages 5-14, Dublin, Ireland, June 2006.

[145] Jerome H. Saltzer, David P. Reed and David D. Clark. End-to-End Arguments in System Design. In *ACM Transactions on Computer Systems*, 2(4):277-288, November 1984.

[146] Sriya Santhanam, Pradheep Elango, Andrea Arpaci-Dusseau and Miron Livny. Deploying Virtual Machines as Sandboxes for the Grid. In Proceedings of the *Workshop on Real, Large Distributed Systems*, San Francisco, California, December 2005.

[147] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam and Mendel Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In Proceedings of the *Large Installation System Administration Conference*, pages 181-194, San Diego, California, October 2003.

[148] Constantine Sapuntzakis and Monica S. Lam. Virtual Appliances in the Collective: A Road to Hassle-Free Computing. In Proceedings of the *Workshop on Hot Topics in Operating Systems*, pages 55-60, Lihue, Hawaii, May 2003.

[149] Fred B. Schneider. Replication Management Using the State-Machine Approach. In *Distributed Systems*, pages 169-197, ACM Press/Addison-Wesley Publishing Co. New York, New York, 1993.

[150] Piyush Shivam. Proactive Experiment-Driven Learning for System Management. Ph.D. Thesis, Duke University, December 2007.

[151] Piyush Shivam, Shivnath Babu and Jeff Chase. Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications. In Proceedings of the *International Conference on Very Large Databases*, pages 535-546, Seoul, Korea, September 2006.

[152] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat and Brent N. Chun. Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems. In Proceedings of the *Workshop on Hot Topics in Operating Systems*, pages 7, Santa Fe, New Mexico, June 2005.

[153] Stephen Soltesz, Herbert Potzl, Marc Fiuczynski, Andy Bavier and Larry Peterson. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In Proceedings of the *European Conference on Computer Systems*, pages 275-288, Lisbon, Portugal, March 2007.

[154] Ion Stoica, Hussein M. Abdel-Wahab and Alex Pothen. A Microeconomic Scheduler for Parallel Computers. In Proceedings of the *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 200-218, Santa Barbara, California, April 1995.

[155] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proceedings of the *SIGCOMM Conference*, pages 149-160, San Diego, California, August 2001.

[156] Michael Stonebraker, Robert Devine, Marcel Kornacker, Witold Litwin, Avi Pfeffer, Adam Sah and Carl Staelin. An Economic Paradigm for Query Processing and Data Migration in Mariposa. In Proceedings of the *International Conference on Parallel and Distributed Information Systems*, pages 58-68, Austin, Texas, September 1994.

[157] David G. Sullivan and Margo I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In Proceedings of the *USENIX Annual Technical Conference*, pages 337-350, San Diego, California, June 2000.

[158] Ivan E. Sutherland. A Futures Market in Computer Time. In *Communications of the ACM*, 11(6):449-451, June 1968.

[159] Karthik Tamilmani, Vinay Pai and Alexander Mohr. SWIFT: A System with Incentives for Trading. In Proceedings of the *Workshop on the Economics of Peer-to-Peer Systems*, Cambridge, Massachusetts, June 2004.

[160] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp and Sape J. Mullender. Experiences with the Amoeba Distributed Operating System. In *Communications of the ACM*, 33(12):46-63, December 1990.

[161] Jesse Tilly and Eric M. Burke. Ant: The Definitive Guide. O'Reilly Media Inc. 2002.

[162] Bhuvan Urgaonkar, Arnold Rosenberg and Prashant Shenoy. Application Placement on a Cluster of Servers. In Proceedings of the *International Conference on Parallel and Distributed Computing Systems*, pages 85-90, SanFrancisco, September 2004.

[163] Bhuvan Urgaonkar and Prashant Shenoy. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. In *IEEE Transactions on Parallel and Distributed Systems*, 15(1):2-17, January 2004.

[164] Bhuvan Urgaonkar, Prashant J. Shenoy and Timothy Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In Proceedings of the *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[165] Vivek Vishnumurthy, Sangeeth Chandrakumar and Emin Gűn Sirer. KARMA: A Secure Economic Framework for Peer-to-Peer Resource Sharing. In Proceedings of the *Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, California, June 2003.

[166] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In Proceedings of the *Symposium on Operating Systems Principles*, pages 148-162, Brighton, United Kingdom, October 2005.

[167] Jim Waldo. The JINI Architecture for Network-Centric Computing. In *Communications of the ACM*, 42(7):76-82, July 1999.

[168] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 181-194, Boston, Massachusetts, December 2002.

[169] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffery O. Kephart and W. Scott Stornetta. Spawn: A Distributed Computational Economy. In *IEEE Transactions on Software Engineering*, 18(2):103-117, February 1992.

[170] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 1-11, Monterey, California, November 1994.

[171] Bruce J. Walker, Gerald J. Popek, Robert English, Charles S. Kline and Greg Thiel. The LOCUS Distributed Operating System. In Proceedings of the *Symposium on Operating Systems Principles*, pages 49-70, Bretton Woods, New Hampshire, October 1983.

[172] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek S. Pai and Larry L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In Proceedings of the *USENIX Annual Technical Conference*, pages 171-184, Boston, Massachusetts, June 2004.

[173] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach and Steven Hand. Parallax: Managing Storage for a Million Machines. In Proceedings of the *Workshop on Hot Topics in Operating Systems*, pages 4, Santa Fe, New Mexico, June 2005.

[174] David Wetherall. Active Network Vision and Reality: Lessons from a Capsule-Based System. In Proceedings of the *Symposium on Operating Systems Principles*, pages 64-79, Kiawah Island, South Carolina, December 1999.

[175] Andrew Whitaker, Richard S. Cox and Steven D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 77-90, San Francisco, California, December 2004.

[176] Andrew Whitaker, Marianne Shaw and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In Proceedings of the *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[177] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 255-270, Boston, Massachusetts, December 2002.

[178] John Wilkes, Patrick Goldsack, G. (John) Janakiraman, Lance Russell, Sharad Singhal and Andrew Thomas. eOS - the Dawn of the Resource Economy. In Proceedings of the *Workshop on Hot Topics in Operating Systems*, pages 188, Elmau, Germany, May 2001.

[179] Ann Wollrath, Roger Riggs and Jim Waldo. A Distributed Object Model for the Java System. In Proceedings of the *USENIX Conference on Object-Oriented Technologies*, pages 219-232, Toronto, Canada, June 1996.

[180] Rich Wolski, James S. Plank, John Brevik and Todd Bryan. Analyzing Market-Based Resource Allocation Strategies for the Computational Grid. In *International Journal of High Performance Computing Applications*, 15(3):258-281, August 2001.

[181] Timothy Wood, Prashant Shenoy, Arun Venkataramani and Mazin Yousif. Black-Box and Gray-Box Strategies for Virtual Machine Migration. In Proceedings of the *Symposium on Networked System Design and Implementation*, Cambridge, Massachusetts, April 2007.

[182] William A. Wulf, Roy Levin and C. Pierson. Overview of the HYDRA Operating System Development. In Proceedings of the *Symposium on Operating Systems Principles*, pages 122-131, Austin, Texas, November 1975.

[183] Ming Q. Xu. Effective Metacomputing Using LSF MultiCluster. In Proceedings of the *IEEE International Symposium on Cluster Computing and the Grid*, pages 100-105, Brisbane, Australia, May 2001.

[184] Jia Yu and Rajkumar Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. In *ACM SIGMOD Record*, 34(3):44-49, September 2005.

[185] Aydan Yumerefendi, Piyush Shivam, David Irwin, Pradeep Gunda, Laura Grit, Azbayer Demberel, Jeff Chase and Shivnath Babu. Towards an Autonomic Computing Testbed. In Proceedings of the *Workshop on Hot Topics in Autonomic Computing*, Jacksonville, Florida, June 2007.

[186] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck and Amin Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems*, pages 123-132, October 2002.

Biography

David Emory Irwin was born on July 29th, 1980 in Birmingham, Alabama. He received a Bachelor of Science in Mathematics and Computer Science *magna cum laude* from Vanderbilt University in Nashville, Tennessee on May 11th, 2001 and a Master of Science in Computer Science from Duke University in Durham, North Carolina on December 30th, 2005. He is currently a Research Fellow in the Computer Science Department at the University of Massachusetts, Amherst and resides in Williamstown, Massachusetts. He has co-authored several technical papers in peer reviewed computer systems conferences and workshops. His work broadly focuses on improving resource management for collections of networked hardware; specific topics include data center power management, market-based resource allocation, and system structures and policies for virtual machine hosting.