Usher: An Extensible Framework for Managing Clusters of Virtual Machines

Marvin McNett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker
– University of California, San Diego

ABSTRACT

Usher is a virtual machine management system designed to impose few constraints upon the computing environment under its management. Usher enables administrators to choose how their virtual machine environment will be configured and the policies under which they will be managed. The modular design of Usher allows for alternate implementations for authentication, authorization, infrastructure handling, logging, and virtual machine scheduling. The design philosophy of Usher is to provide an interface whereby users and administrators can request virtual machine operations while delegating administrative tasks for these operations to modular plugins. Usher's implementation allows for arbitrary action to be taken for nearly any event in the system. Since July 2006, Usher has been used to manage virtual clusters at two locations under very different settings, demonstrating the flexibility of Usher to meet different virtual machine management requirements.

Introduction

Usher is a cluster management system designed to substantially reduce the administrative burden of managing cluster resources while simultaneously improving the ability of users to request, control, and customize their resources and computing environment. System administrators of cluster computing environments face a number of imposing challenges. Different users within a cluster can have a wide range of computing demands, spanning general best-effort computing needs, batch scheduling systems, and complete control of dedicated resources. These resource demands vary substantially over time in response to changes in workload, user base, and failures. Furthermore, users often need to customize their operating system and application environments, substantially increasing configuration and maintenance tasks. Finally, clusters rarely operate in isolated administrative environments, and must be integrated into existing authentication, storage, network, and host address and name service infrastructure.

Usher balances these imposing requirements using a combination of abstraction and architecture. Usher provides a simple abstraction of a logical cluster of virtual machines, or virtual cluster. Usher users can create any number of virtual clusters (VCs) of arbitrary size, while Usher multiplexes individual virtual machines (VMs) on available physical machine hardware. By decoupling logical machine resources from physical machines, users can create and use machines according to their needs rather than according to assigned physical resources.

Architecturally, Usher is designed to impose few constraints upon the computing environment under its management. No two sites have identical hardware and software configurations, user and application requirements, or service infrastructures. To facilitate its use in a wide range of environments, Usher combines a core set of interfaces that implement basic mechanisms, clients for using these mechanisms, and a framework for expressing and customizing administrative policies in extensible modules, or plugins.

The Usher core implements basic virtual cluster and machine management mechanisms, such as creating, destroying, and migrating VMs. Usher clients use this core to manipulate virtual clusters. These clients serve as interfaces to the system for users as well as for use by higher-level cluster software. For example, an Usher client called *ush* provides an interactive command shell for users to interact with the system. We have also implemented an adapter for a high-level execution management system [6], which operates as an Usher client, that creates and manipulates virtual clusters on its own behalf.

Usher supports customizable modules for two important purposes. First, these modules enable Usher to interact with broader site infrastructure, such as authentication, storage, and host address and naming services. Usher implements default behavior for common situations, e.g., newly created VMs in Usher can use a site's DHCP service to obtain addresses and domain names. Additionally, sites can customize Usher to implement more specialized policies; at UCSD, an Usher VM identity module allocates IP address ranges to VMs within the same virtual cluster.

Second, pluggable modules enable system administrators to express site-specific policies for the placement, scheduling, and use of VMs. As a result, Usher allows administrators to decide how to configure their virtual machine environments and determine the appropriate management policies. For instance, to support a general-purpose computing environment, administrators can

install an available Usher scheduling and placement plugin that performs round-robin placement of VMs across physical machines and simple rebalancing in response to the addition or removal of virtual and physical machines. With this plugin, users can dynamically add or remove VMs from VCs at any time without having to specify service level agreements (SLAs) [9, 17, 22], write configuration files [10], or obtain leases on resources [12, 14]. With live migration of VMs, Usher can dynamically and transparently adjust the mapping of virtual to physical machines to adapt to changes in load among active VMs or the working set of active VMs, exploit affinities among VMs (e.g., to enhance physical page sharing [20]), or add and remove hardware with little or no interruption.

Usher enables other powerful policies to be expressed, such as power management (reduce the number of active physical machines hosting virtual clusters), distribution (constrain virtual machines within a virtual cluster to run on separate nodes), and resource guarantees. Another installation of Usher uses its cluster to support scientific batch jobs running within virtual clusters, guarantees resources to those jobs when they run, and implements a load-balancing policy that migrates VMs in response to load spikes [13].

Usher is a fully functional system. It has been installed in cluster computing environments at UCSD and the Russian Research Center in Kurchatov, Russia. At UCSD, Usher has been in production use since January 2007. It has managed up to 142 virtual machines in 26 virtual clusters across 25 physical machines. The Usher implementation is sufficiently reliable that we are now migrating the remainder of our user base from dedicated physical machines to virtual clusters, and Usher will soon manage all 130 physical nodes in our cluster. In the rest of this paper we describe the design and implementation of Usher, as well as our experiences using it.

Related Work

Since the emergence of widespread cluster computing over a decade ago [8, 16], many cluster configuration and management systems have been developed to achieve a range of goals. These goals naturally influence individual approaches to cluster management. Early configuration and management systems, such as Galaxy [19], focus on expressive and scalable mechanisms for defining clusters for specific types of service, and physically partition cluster nodes among those types.

More recent systems target specific domains, such as Internet services, computational grids, and experimental testbeds, that have strict workload or resource allocation requirements. These systems support services that express explicit resource requirements, typically in some form of service level agreement (SLA). Services provide their requirements as input to the system, and the system allocates its resources among services while satisfying the constraints of the SLA requirements.

For example, Océano provides a computing utility for e-commerce [9]. Services formally state their workload performance requirements (e.g., response time), and Océano dynamically allocates physical servers in response to changing workload conditions to satisfy such requirements. Rocks and Rolls provide scalable and customizable configuration for computational grids [11, 18], and Cluster-on-Demand (COD) performs resource allocation for computing utilities and computational grid services [12]. COD implements a virtual cluster abstraction, where a virtual cluster is a disjoint set of physical servers specifically configured to the requirements of a particular service, such as a local site component of a larger wide-area computational grid. Services specify and request resources to a site manager and COD leases those resources to them. Finally, Emulab provides a shared network testbed in which users specify experiments [21]. An experiment specifies network topologies and characteristics as well as node software configurations, and Emulab dedicates, isolates, and configures testbed resources for the duration of the experiment.

The recent rise in virtual machine monitor (VMM) popularity has naturally led to systems for configuring and managing virtual machines. For computational grid systems, for example, Shirako extends Cluster-on-Demand by incorporating virtual machines to further improve system resource multiplexing while satisfying explicit service requirements [14], and VIOLIN supports both intra-and inter-domain migration to satisfy specified resource utilization limits [17]. Sandpiper develops policies for detecting and reacting to hotspots in virtual cluster systems while satisfying application SLAs [22], including determining when and where to migrate virtual machines, although again under the constraints of meeting the stringent SLA requirements of a data center.

On the other hand, Usher provides a framework that allows system administrators to express site-specific policies depending upon their needs and goals. By default, the Usher core provides, in essence, a general-purpose, best-effort computing environment. It imposes no restrictions on the number and kind of virtual clusters and machines, and performs simple load balancing across physical machines. We believe this usage model is important because it is widely applicable and natural to use. Requiring users to explicitly specify their resource requirements for their needs, for example, can be awkward and challenging since users often do not know when or for how long they will need resources. Further, allocating and reserving resources can limit resource utilization; guaranteed resources that go idle cannot be used for other purposes. However, sites can specify more elaborate policies in Usher for controlling the placement, scheduling, and migration of VMs if desired. Such policies can range

from batch schedulers to allocation of dedicated physical resources.

In terms of configuration, Usher shares many of the motivations that inspired the Manage Large Networks (MLN) tool [10]. The goal of MLN is to enable administrators and users to take advantage of virtualization while easing administrator burden. Administrators can use MLN to configure and manage virtual machines and clusters (*distributed projects*), and it supports multiple virtualization platforms (Xen and User-Mode Linux). MLN, however, requires administrators to express a number of static configuration decisions through configuration files (e.g., physical host binding, number of virtual hosts), and supports only coarse granularity dynamic reallocation (manually by the administrator). Usher configuration is interactive and dynamic, enables users to create and manage their virtual clusters without administrative intervention, and enables a site to globally manage all VMs according to cluster-wide policies.

XenEnterprise [5] from XenSource and VirtualCenter [4] from VMware are commercial products for managing virtual machines on cluster hardware from the respective companies. XenEnterprise provides a graphical administration console, Virtual Data Center, for creating, managing, and monitoring Xen virtual machines. VirtualCenter monitors and manages VMware virtual machines on a cluster as a data center, supporting VM restart when nodes fail and dynamic load balancing through live VM migration. Both list interfaces for external control, although it is not clear whether administrators can implement arbitrary plugins and policies for integrating the systems into existing infrastructure, or controlling VMs in response to arbitrary events in the system. In this regard, VMWare's Infrastructure Management SDK provides functionality similar to that provided by the Usher client API. However, this SDK does not provide the tight integration with VMWare's centralized management system that plugins do for the Usher system. Also, of course, these are all tied to managing a single VM product, whereas Usher is designed to interface with any virtualization platform that exports a standard administrative interface.

System Architecture

This section describes the architecture of Usher. We start by briefly summarizing the goals guiding our design, and then present a high-level overview of the system. We then describe the purpose and operation of each of the various system components, and how they interact with each other to accomplish their tasks. We end with a discussion of how the Usher system accommodates software and hardware failures.

Design Goals

As mentioned, no two sites have identical hardware and software configurations, user and application requirements, or service infrastructures. As a result, we designed Usher as a flexible platform for constructing virtual machine management installations customized to the needs of a particular site.

To accomplish this goal, we had two design objectives for Usher. First, Usher maintains a clean separation between policy and mechanism. The Usher core provides a minimal set of mechanisms essential for virtual machine management. For instance, the Usher core has mechanisms for placing and migrating virtual machines, while administrators can install site-specific policy modules that govern where and when VMs are placed.

Second, Usher is designed for extensibility. The Usher core provides three ways to extend functionality, as illustrated in Figure 1. First, Usher provides a set of *hooks* to integrate with existing infrastructure. For instance, while Usher provides a reference implementation for use with the Xen VMM, it is straightforward to write stubs for other virtualization platforms. Second, developers can use a *Plugin API* to enhance Usher functionality. For example, plugins can provide database functionality for persistently storing system state using a file-backed database, or provide authentication backed by local UNIX passwords. Third, Usher provides a *Client API* for integrating with user interfaces and third-party tools, such as the Usher command-line shell and the Plush execution management system (discussed in the Applications subsection of the Implementation section).

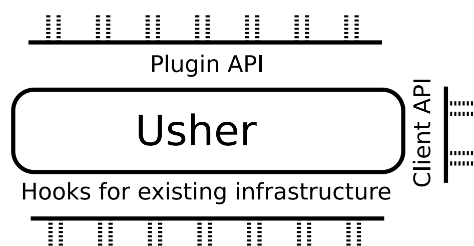

Figure 1: Usher interfaces.

Usher Overview

A running Usher system consists of three main components: local node managers (LNMs), a centralized controller, and clients. A client consists of an application that utilizes the Usher client library to send virtual machine management requests to the controller. We have written a few applications that import the Usher client library for managing virtual machines (a shell and XML-RPC server) with more under development (a web frontend and command line suite).

Figure 2 depicts the core components of an Usher installation. One LNM runs on each physical node and interacts directly with the VMM to perform management operations such as creating, deleting, and migrating VMs on behalf of the controller. The local node managers also collect resource usage data from the VMMs and monitor local events. LNMs report resource usage updates and events back to the controller for use by plugins and clients.

The controller is the central component of the Usher system. It receives authenticated requests from clients and issues authorized commands to the LNMs. It also communicates with the LNMs to collect usage data and manage virtual machines running on each physical node. The controller provides event notification to clients and plugins registered to receive notification for a particular event (e.g., a VM has started, been destroyed, or changed state). Plugin modules can perform a wide range of tasks, such as maintaining persistent system-wide state information, performing DDNS updates, or doing external environment preparation and cleanup.

The client library provides an API for applications to communicate with the Usher controller. Essentially, clients submit requests to the controller when they need to manipulate their VMs or request additional VMs. The controller can grant or deny these requests as its operational policy dictates. One purpose of clients is to serve as the user interface to the system, and users use clients to manage their VMs and monitor system state. More generally, arbitrary applications can use the client library to register callbacks for events of interest in the Usher system.

Typically, a few services also support a running Usher system. Depending upon the functionality desired and the infrastructure provided by a particular site, these services might include a combination of the following: a database server for maintaining state information or logging, a NAS server to serve VM filesystems, an authentication server to provide authentication for Usher and VMs created by Usher, a DHCP server to manage IP addresses, and a DNS server for name resolution of all Usher created VMs. Note that an administrator may configure Usher to use any set of support services desired, not necessarily the preceding list.

Usher Components

As noted earlier, Usher consists of three main components, local node managers on each node, a central controller, and Usher clients.

Local Node Managers

The local node managers (LNMs) operate closest to the hardware. As shown in Figure 2, LNMs run as servers on each physical node in the Usher system. The LNMs have three major duties: i) to provide a remote API to the controller for managing local VMs, ii) to collect and periodically upload local resource usage data to the controller, and iii) to report local events to the controller.

Each LNM presents a remote API to the controller for manipulating VMs on its node. Upon invoking an API method, the LNM translates the operation into the equivalent operation of the VM management API exposed by the VMM running on the node. Note that all LNM API methods are asynchronous so that the controller does not block waiting for the VMM operation to complete. We emphasize that this architecture abstracts VMM-specific implementations – the controller is oblivious to the specific VMMs running on the physical nodes as long as the LNM provides the remote API implementation. As a result, although our implementation currently uses the Xen VMM, Usher can target other virtualization platforms. Further, Usher is capable of managing VMs running any operating system supported by the VMMs under its management.

As the Usher system runs, VM and VMM resource usage fluctuates considerably. The local node manager on each node monitors these fluctuations and reports them back to the controller. It reports resource usage of CPU utilization, network receive and transmit loads, disk I/O activity, and memory usage in 1, 5, and 15-minute averages.

In addition to changes in resource usage, VM state changes sometimes occur unexpectedly. VMs can crash or even unexpectedly appear or disappear from the system. Detecting these and other related events requires both careful monitoring by the local node managers as well as VMM support for internal event notification. Administrators can set a tunable parameter for how often the LNM scans for missing VMs or unexpected VMs. The LNM will register callbacks with the VMM platform for other events, such as VM crashes; if the VMM does not support such callbacks, LNM will periodically scan to detect these events.

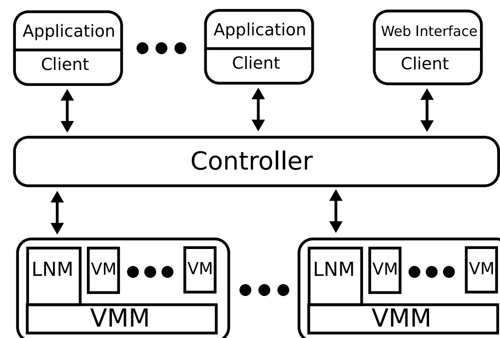

Figure 2: Usher components.

Usher Controller

The controller is the center of the Usher system. It can either be bootstapped into a VM running in the system, or run on a separate server. The controller provides the following:

- User authentication
- VM operation request API
- Global state maintenance
- Consolidation of LNM monitoring data
- Event notification

User authentication: Usher uses SSL-encrypted user authentication. All users of the Usher system must authenticate before making requests of the system. Administrators are free to use any of the included authentication modules for use with various authentication

backends (e.g., LDAP), or implement their own. An administrator can register multiple authentication modules, and Usher will query each in turn. This support is useful, for instance, to provide local password authentication if LDAP or NIS authentication fails. After receiving a user's credentials, the controller checks them against the active authentication module chain. If one succeeds before reaching the end of the chain, the user is authenticated. Otherwise, authentication fails and the user must retry.

VM operation request API: A key component of the controller is the remote API for Usher clients. This API is the gateway into the system for VM management requests (via RPC) from connecting clients. Typically, the controller invokes an authorization plugin to verify that the authenticated user can perform the operation before proceeding. The controller may also invoke other plugins to do preprocessing such as checking resource availability and making placement decisions at this point. Usher calls any plugin modules registered to receive notifications for a particular request once the controller receives such a request.

Usher delegates authorization to plugin modules so that administrators are free to implement any policy or policies they wish and stack and swap modules as the system runs. In addition, an administrator can configure the monitoring infrastructure to automatically swap or add policies as the system runs based upon current system load, time of day, etc. In its simplest form, an authorization policy module openly allows users to create and manipulate their VMs as they desire or view the global state of the system. More restrictive policies may limit the number of VMs a user can start, prohibit or limit migration, or restrict what information the system returns upon user query.

Once a request has successfully traversed the authorization and preprocessing steps, the controller executes it by invoking asynchronous RPCs to each LNM involved. As described above, it is up to any plugin policy modules to authorize and check resource availability prior to this point. Depending upon the running policy, the authorization and preprocessing steps may alter a user request before the controller executes it. For example, the policy may be to simply "do the best I can" to honor a request when it arrives. If a user requests more VMs than allowed, this policy will simply start as many VMs as are allowed for this user, and report back to the client what action was taken. Finally, if insufficient resources are available to satisfy an authorized and preprocessed request, the controller will attempt to fulfill the request until resources are exhausted.

Global state maintenance: The controller maintains a few lists which constitute the global state of the system. These lists link objects encapsulating state information for running VMs, running VMMs, and instantiated virtual clusters (VCs). A virtual cluster in Usher can contain an arbitrary set of VMs, and administrators are free to define VCs in any way suitable to their computing environment.

In addition to the above lists, the controller maintains three other lists of VMs: *lost*, *missing*, and *unmanaged* VMs. The subtle distinction between lost and missing is that lost VMs are a result of an LNM or VMM failure (the controller is unable to make this distinction), whereas a missing VM is a result of an unexpected VM disappearance as reported by the LNM where the VM was last seen running. A missing VM can be the result of an unexpected VMM error (e.g., we have encountered this case upon a VMM error on migration). Unmanaged VMs are typically a result of an administrator manually starting a VM on a VMM being managed by Usher; Usher is aware of the VM, but is not itself managing it. The list of unmanaged VMs aids resource usage reporting so that Usher has a complete picture of all VMs running on its nodes.

Having the controller maintain system state removes the need for it to query all LNMs in the system for every VM management operation and state query. However, the controller does have to synchronize with the rest of system and we discuss synchronization further in the Component Interaction subsection below.

Consolidation of LNM monitoring data: Proper state maintenance relies upon system monitoring. The controller is responsible for consolidating monitoring data sent by the local node managers into a format accessible by the rest of the system. Clients use this data to describe the state of the system to users, and plugins use this data to make policy decisions. For example, plugin modules may use this data to restrict user resource requests based on the current system load, or make VM scheduling decisions to determine where VMs should run.

Event notification: Usher often needs to alert clients and plugin modules when various events in the system occur. Events typically fall into one of three categories:

- VM operation requests
- VM state changes
- Errors and unexpected events

Clients automatically receive notices of state changes of their virtual machines. Clients are free to take any action desired upon notification, and can safely ignore them. Plugin modules, however, must explicitly register with the controller to receive event notifications. Plugins can register for any type of event in the system. For example, a plugin may wish to receive notice of VM operation requests for preprocessing, or error and VM state change events for reporting and cleanup.

Clients and the Client API

Applications use the Usher client API to interact with the Usher controller. This API provides methods for requesting or manipulating VMs and performing state queries. We refer to any application importing this API as a client.

The client API provides the mechanism for clients to securely authenticate and connect to the Usher controller. Once connected, an application may call any of the methods provided by the API. All methods are asynchronous, event-based calls to the controller (see the Implementation section below). As mentioned above, connected clients also receive notifications from the controller for state changes to any of their VMs. Client applications can have their own callbacks invoked for these notifications.

Component Interaction

Having described each of the Usher components individually, we now describe how they interact in more detail. We first discuss how the controller and LNMs interact, and then describe how the controller and clients interact. Note that clients never directly communicate with LNMs; in effect, the controller ''proxies'' all interactions between clients and LNMs.

Controller and LNM Interaction

When an LNM starts or receives a controller recovery notice, it connects to the controller specified in its configuration file. The controller authenticates all connections from LNMs, and encrypts the connection for privacy. Upon connection to the controller, the LNM passes a capability to the controller for access to its VM management API.

Using the capability returned by the LNM, the controller first requests information about the hardware configuration and a list of currently running virtual machines on the new node. The controller adds this information to its lists of running VMs and VMMs in the system. It then uses the capability to assume management of the VMs running on the LNM's node.

The controller also returns a capability back to the LNM. The LNM uses this capability for both event notification and periodic reporting of resource usage back to the controller.

When the controller discovers that a new node already has running VMs (e.g., because the node's LNM failed and restarted), it first determines if it should assume management of any of these newly discovered VMs. The controller makes this determination based solely upon the name of the VM. If the VM name ends with the domain name specified in the controller's configuration file, then the controller assumes it should manage this VM. Any VMs which it should not manage are placed on the *unmanaged* list discussed above. For any VMs which the controller should manage, the controller creates a VM object instance and places this object on its running VMs list. These instances are sent to the LNMs where the VMs are running and cached there. Whenever an LNM sees that a cached VM object is inconsistent with the corresponding VM running there (e.g., the state of the VM changed), it alerts the controller of this event. The controller then updates the cached object on the LNM. In this way, the update serves as an acknowledgment and the LNM knows that the controller received notice of the event.

Similarly, the controller sends VM object instances for newly created VMs to an LNM before the VM is actually started there. Upon successful return from a start command, the controller updates the VMs cached object state on the LNM. Subsequently, the LNM assumes the responsibility for monitoring and reporting any unexpected state changes back to the controller.

Controller and Client Interaction

Clients to the Usher system communicate with the controller. Before a client can make any requests, it must authenticate with the controller. If authentication succeeds, the controller returns a capability to the client for invoking its remote API methods. Clients use this API to manipulate VMs.

Similar to the local node managers, clients receive cached object instances corresponding to their VMs from the controller upon connection. If desired, clients can filter this list of VMs based upon virtual cluster grouping to limit network traffic. The purpose of the cached objects at the client is twofold. First, they provide a convenient mechanism by which clients can receive notification of events affecting their VMs, since the controller sends updates to each cached VM object when the actual VM is modified. Second, the cached VM objects provide state information to the clients when they request VM operations. With this organization, clients do not have to query the controller about the global state of the system before actually submitting a valid request. For example, a client should not request migration of a non-existent VM, or try to destroy a VM which it does not own. The client library is designed to check for these kinds of conditions before submitting a request. Note that the controller is capable of handling errant requests; this scheme simply offloads request filtering to the client.

The controller is the authority on the global state of the system. When the controller performs an action, it does so based on what it believes is the current global state. The cached state at the client reflects the controller's global view. For this reason, even if the controller is in error, its state is typically used by clients for making resource requests. The controller must be capable of recovering from errors due to inconsistencies between its own view of the global state of the system and the actual global state. These inconsistencies are typically transient (e.g., a late event notification from an LNM), in which case the controller may log an error and return an error message to the client.

Failures

As the Usher system runs, it is possible for the controller or any of the local node managers to become unavailable. This situation could be the result of hardware failure, operating system failure, or the server itself

failing. Usher has been designed to handle these failures gracefully.

In the event of a controller failure, the LNMs will start a listening server for a recovery announcement sent by the controller. When the controller restarts, it sends a recovery message to all previously known LNMs. When the LNMs receive this announcement, they reconnect to the controller. As mentioned in the Controller and LNM Interaction section above, when an LNM connects to the controller, it passes information about its physical parameters and locally running VMs. With this information from all connecting LNMs, the controller recreates the global state of the system. With this design, Usher only requires persistent storage of the list of previously known LNMs rather than the entire state of the system to restore system state upon controller crash or failure.

Since the controller does not keep persistent information about which clients were known to be connected before a failure, it cannot notify clients when it restarts. Instead, clients connected to a controller which fails will attempt to reconnect with timeouts following an exponential backoff. Once reconnected, clients flush their list of cached VMs and receive a new list from the controller.

The controller detects local node manager failures upon disconnect or TCP timeout. When this situation occurs, the controller changes the state of all VMs known to be running on the node with the failed LNM to *lost*. It makes no out of band attempts to determine if lost VMs are still running or if VMMs on which LNMs have failed are still running. The controller simply logs an error, and relies upon the Usher administrator or a recovery plugin to investigate the cause of the error.

Implementation

In this section we describe the implementation of Usher, including the interfaces that each component supports and the plugins and applications currently implemented for use with the system.

Component	LoC
LNM (w/ Xen hooks)	907
Controller	1703
Client API	750
Utilities	633
Ush	1099

Table 1: Code size of individual components.

The main Usher components are written in Python [2]. In addition, Usher makes use of the Twisted network programming framework [3]. Twisted provides convenient mechanisms for implementing event based servers, asynchronous remote procedure calls, and remote object synchronization. Table 1 shows source code line counts for the main Usher components, for total of 3993 lines of code. Also included is the line count for the *ush* application (over 400 of which is simply online documentation).

Local Node Managers

Local Node Managers export the remote API shown in Table 2 to the controller. This API is made available to the controller via a capability passed to the controller when an LNM connects to it.

Method Name	Description
get_details(vm name)	get VM state information
get_status(vm name)	get VM resource usage statistics
receive(vm instance)	receive new cached VM object
start(vm name)	start cached VM
op_on(operation, vm name)	operate on existing VM
migrate(vm name, lnm name)	migrate VM to LNM
get_node_info()	get node physical characteristics
get_node_status()	get node dynamic and resource usage info

Table 2: Local node manager remote API.

This API includes methods to query for VM state information and VM resource usage details using the get_details and get_status methods, respectively. State information includes run state, memory allocation, IP and MAC addresses, the node on which VM is running, VM owner, etc. Resource usage includes 1, 5, and 15-minute utilizations of the various hardware resources.

The receive method creates a cached copy of a VM object on an LNM. An LNM receives the cached copy when it connects to the controller. It compares the state of the VM object with the actual state of the virtual machine. If the states differ the LNM notifies the controller, which updates the LNM's cached copy of the VM as an acknowledgment that it received the state change notice.

In addition, the cached copy of a VM at its LNM contains methods for manipulating the VM it represents. When a VM manipulation method exposed by the LNM's API is invoked (one of start, op_on, or migrate), the method calls the corresponding method of the cached VM object to perform the operation. This structure provides a convenient way to organize VM operations. To manipulate a VM, a developer simply calls the appropriate method of the cached VM object. Note that the controller must still update the state of its VM object as an acknowledgment that the controller knows the operation was successful.

Most operations on an existing VM are encapsulated in the op_on function, and have similar signatures. Table 3 shows the list of valid operations to the op_on method.

Operation	Description
pause	pause VM execution, keeping memory image resident
resume	resume execution of a paused VM
shutdown	nicely halt a VM
reboot	shutdown and restart VM
hibernate	save VM's memory image to persistent storage
restore	restore hibernated VM to run state
destroy	hard shutdown a VM
cycle	destroy and restart a VM

Table 3: Operations supported by the op_on method.

All VM operations invoke a corresponding operation in the VMM's administration API. Though Usher currently only manages Xen VMs, it is designed to be VMM-agnostic. An installation must provide an implementation of Usher's VMM interface to support new virtual machine managers.

The LNM's remote API exposes a few methods that do not operate on VMs. The get_node_info method returns hardware characteristics of the physical machine. The controller calls this method when an LNM connects. The get_node_status method is similar to the get_status method. Additionally, it reports the number of VMs running on the VMM and the amount of free memory on the node.

Usher Controller

The remote API exported by the controller to connecting clients closely resembles the interface exported by LNMs to the controller. Table 4 lists the methods exported by the controller to Usher clients. This API is made available to clients via a capability passed upon successful authentication with the controller.

Note that most of these methods operate on lists of VMs, rather than single VMs expected by the LNM API methods. Since Usher was designed to manage clusters, the common case is to invoke these methods on lists of VMs rather than on a single VM at a time. This convention saves significant call overhead when dealing with large lists of VMs.

The start and migrate methods both take a list of LNMs. For start, the list specifies the LNMs on which the VMs should be started. An empty list indicates that the VMs can be started anywhere. Recall that this parameter is simply a suggestion to the controller. Policies installed in the controller dictate whether or not the controller will honor the suggestion. Likewise, the LNM list passed to the migrate method is simply a suggestion to the controller as to where to migrate the VMs. The controller can choose to ignore this suggestion or ignore the migrate request altogether based upon the policies installed.

The operations supported by the op_on method in the controller API are the same as those to the op_on method of the remote LNM API (Table 3).

Method Name	Description
list(vm list, status)	list state and resource usage information for VMs
list_lnms(lnm list, status)	list LNMs and resource usage information for VMMs
start(vm list, lnm list)	start list of VMs on LNMs
op_on(operation, vm list)	operate on existing VMs
migrate(vm list, lnm list)	migrate VMs to LNMs

Table 4: Controller remote API for use by Usher clients.

Client API

The client API closely mirrors that of the controller. An important difference between these two APIs, though, is that the client API signatures contain many additional parameters to aid in working with large sets of VMs. These additional parameters allow users to operate on arbitrary sets of VMs and virtual clusters in a single method call. The API supports specifying VM sets as regular expressions, explicit lists, or ranges (when VM names contain numbers). The client API also allows users to specify source and destination LNMs using regular expressions or explicit lists.

Another difference between the client and controller APIs is that the client API expands the op_on method into methods for each type of operation. Explicitly enumerating the operations as individual methods avoids confusing application writers unfamiliar with the op_on method. These methods simply wrap the call to the op_on method, which is still available for those wishing to call it directly.

Finally, the client API contains connect and reconnect methods. These methods contact and authenticate with the controller via SSL. They also start the client's event loop to handle cached object updates and results from asynchronous remote method calls. The reconnect method is merely a convenience method to avoid having to pass credentials to the API if a reconnect is required after having been successfully connected. This method can be used by a reconnecting application upon an unexpected disconnect.

Configuration Files

All Usher components are capable of reading configuration data from text files. All valid configuration parameters, their type, and default values are

specified in the code for each component. When each component starts, it first parses its configuration files (if found). The configuration system tries to read in values from the following locations (in order): a default location in the host filesystem, a default location in the user's home directory, and finally a file indicated by an environment variable. This search ordering enables users to override default values easily. Values read in later configuration files replace values specified in a previously read file.

Plugins

Plugins are separate add-on modules which can be registered to receive notification of nearly any event in the system. Plugins live in a special directory (aptly named ''plugins'') of the Usher source tree. Usher also looks in a configurable location for third-party/user plugins. Any plugins found are automatically sourced and added to a list of *available* plugins. To register a plugin, the controller provides an additional API call register_plugin(plugin name, event, configuration file). Each plugin is required to provide a method named entry_point to be called when an event fires for which it is registered. It is possible to add a single plugin to multiple event handler chains. Note that the register_plugin method can be called from anywhere in the Usher code.

By default, plugins for each event are simply called in the order in which they are registered. Therefore careful consideration must be given to ordering while registering plugins. A plugin's configuration object can optionally take an *order* parameter that governs the order in which plugins are called on the event's callback list. The plugin API also provides a converse unregister_plugin call to change event handling at runtime.

Plugins can be as simple or complex as necessary. Since the controller invokes plugin callback chains asynchronously, complex plugins should not interfere with the responsiveness of the Usher system (i.e., the main controller event loop will not block waiting for a plugin to finish its task).

Policies in an Usher installation are implemented as plugins. As an example, an administrator may have strict policies regarding startup and migration of virtual machines. To enforce these policies, a plugin (or plugins) is written to authorize start and migrate requests. This plugin gets registered for the start_request and migrate_request events, either manually using the controllers register_plugin command, or by specifying these registrations in the controller's configuration file. Once registered, subsequent start and migrate requests are passed to the plugin (in the form of a Request object) for authorization. At this point, the plugin can approve, approve with modification, or simply reject the request. Once this is done, the request is passed on to any other plugins registered on the start_request or migrate_request event lists with a higher order attribute.

Besides authorization policies, one can imagine policies for VM operation and placement. For example, initial VM placement, VM scheduling (i.e., dynamic migration based on load or optimizing a utility function), or reservations. A policy plugin for initial placement would be registered for the start_request event (probably with a higher order attribute than the startup authorization policy discussed above so that it is called later in the plugin callback chain). Some simple policies such a plugin might support are round-robin and least-loaded. Scheduling and reservation plugins could be registered with a timer to be fired periodically to evaluate the state of the system and make decisions about where VMs should be migrated and which VMs might have an expired reservation, respectively.

As a concrete example of plugin usage in Usher, we now discuss plugins implemented for use by the UCSD Usher installation, and outline the sequence of events for a scenario of starting a set of VMs. Detailed discussion about these plugins is deferred to the UCSD SysNet subsection of the Usher Deployments section.

The UCSD installation uses the following plugins: an SQL database plugin for logging, mirroring global system state, and IP address management; an LDAP plugin for user authentication for both Usher and VMs created by Usher; a filesystem plugin for preparing writable VM filesystems; a DNS plugin for modifying DNS entries for VMs managed by Usher; and a default placement plugin to determine where VMs should be started. We are developing additional modules for VM scheduling as part of ongoing research.

All plugins for the UCSD installation are written in Python. Table 5 contains line counts for these plugins. Overall, the UCSD plugins total 1406 lines of code.

Plugin	LoC
Database	260
LDAP	870
Filesystem	54
DNS	90
Placement	132

Table 5: Code size of UCSD plugins.

When a request to start a list of VMs arrives, the controller calls the modules registered for the ''start request'' event. The placement and database modules are in the callback list for this event. The placement module first determines which VMs to start based on available resources and user limits, then determines where each of the allowed VMs will start. The database module receives the modified request list, logs the request, then reserves IP addresses for each of the new VMs.

The controller generates a separate VM start command for each VM in the start list. Prior to invoking the start command, the controller triggers a "register VM" event. The database, DNS, and file system plugin modules are registered for this event. The database module adds the VM to a "VMs" table to mirror the fact that this is now a VM included in the controller's view of the global state. The DNS plugin simply sends a DDNS update to add an entry for this VM in our DNS server. The filesystem module prepares mount points on an NFS server for use by each VM.

Finally, upon return from each start command, a "start complete" event fires. The database module registers to receive this event. The database module checks the result of the command, logs this to the database, then either marks the corresponding IP address as used (upon success) or available (upon failure). Note that the database module does not change the state of the VM in the VMs table until receiving a "state changed" event for this VM (which originates at the LNM).

Sites can install or customize plugins as necessary. The Usher system supports taking arbitrary input and passing it to plugins on request events. For example, a request to start a VM may include information about which OS to boot, which filesystem to mount, etc. Plugin authors can use this mechanism to completely customize VMs at startup.

Applications

We have written two applications using the client API, a shell named *Ush* and an XML-RPC server named *plusher*, and are developing two other applications, a Web interface and a command-line suite. The Web interface will provide a convenient point and click interface accessible from any computer with a web browser. The command line suite will facilitate writing shell scripts to manage virtual machines.

Ush Client

The Usher shell *Ush* provides an interactive command-line interface to the API exported by the Usher controller. *Ush* provides persistent command line history and comes with extensive online help for each command. If provided, *Ush* can read connection details and other startup parameters from a configuration file. *Ush* is currently the most mature and preferred interface for interacting with the Usher system.

As an example of using the Usher system, we describe a sample *Ush* session from the UCSD Usher installation, along with a step-by-step description of actions taken by the core components to perform each request. In this example, user "mmcnett" requests ten VMs. Figure 3 contains a snapshot of *Ush* upon completion of the start command.

First a user connects to the Usher controller by running the "connect" command. In connecting, the controller receives the user's credentials and checks them against the LDAP database. Once authentication succeeds, the controller returns a capability for its remote API and all of user mmcnett's VMs. The somewhat unusual output "<Command 0 result pending...>" reflects the fact that all client calls to the controller are asynchronous. When "connect" returns, *Ush* responds with the "Command 0 result:" message followed by the actual result "Connected".

Upon connecting *Ush* saves the capability and cached VM instances sent by the controller. Once connected, the user runs the "list" command to view his currently running VMs. Since the client already has cached instances of user mmcnett's VMs, the list command does not invoke any remote procedures. Consequently, *Ush* responds immediately indicating that user mmcnett already has two VMs running.

The user then requests the start of ten VMs in the "sneetch" cluster. In this case, the -n argument specifies the name of a cluster, and the -c argument specifies how many VMs to start in this cluster. When the controller receives this request, it first calls on the authorization and database modules to authorize the request and reserve IP addresses for the VMs to be started. Next, the controller calls the initial placement plugin to map where the authorized VMs should be started. The controller calls the start method of the remote LNM API at each new VM's LNM. The LNMs call the corresponding method of the VMM administration API to start each VM. Upon successful return of all of these remote method calls, the controller responds to the client that the ten VMs were started in two seconds and provides information about where each VM was started. After completing their boot sequence, user mmcnett can ssh into any of his new VMs by name.

```
mmcnett@5:~ (on 5.sysnet.usher.ucsdsys.net)
Usher Shell 0.2
Type '?' or 'help' for help
Use: help <command> for command specific help
ush> connect
Password:
<Command 0 result pending...>
Command 0 result:
Connected
mmcnett:ush> list
2 VMs:
VM                                     state     VMM
==============================================================================
horton.mmcnett.usher.ucsdsys.net       run       vmm47.usher.ucsdsys.net
usher.mmcnett.usher.ucsdsys.net        run       vmm52.usher.ucsdsys.net
mmcnett:ush> start -n sneetch -c 10
<Command 1 result pending...>
Command 1 result:
Controller started 10 VMs in 2 seconds:
1.sneetch.mmcnett.usher.ucsdsys.net started on vmm43.usher.ucsdsys.net
10.sneetch.mmcnett.usher.ucsdsys.net started on vmm44.usher.ucsdsys.net
2.sneetch.mmcnett.usher.ucsdsys.net started on vmm76.usher.ucsdsys.net
3.sneetch.mmcnett.usher.ucsdsys.net started on vmm74.usher.ucsdsys.net
4.sneetch.mmcnett.usher.ucsdsys.net started on vmm48.usher.ucsdsys.net
5.sneetch.mmcnett.usher.ucsdsys.net started on vmm60.usher.ucsdsys.net
6.sneetch.mmcnett.usher.ucsdsys.net started on vmm71.usher.ucsdsys.net
7.sneetch.mmcnett.usher.ucsdsys.net started on vmm46.usher.ucsdsys.net
8.sneetch.mmcnett.usher.ucsdsys.net started on vmm78.usher.ucsdsys.net
9.sneetch.mmcnett.usher.ucsdsys.net started on vmm72.usher.ucsdsys.net
mmcnett:ush>
```

Figure 3: Ush

Plusher

Plush [7] is an extensible execution management system for large-scale distributed systems, and *plusher* is an XML-RPC server that integrates Plush with Usher. Plush users describe batch experiments or computations in a domain-specific language. Plush uses this

input to map resource requirements to physical resources, bind a set of matching physical resources to the experiment, set up the execution environment, and finally execute, monitor and control the experiment.

Since Usher is essentially a service provider for the virtual machine ‘‘resource’’, it was natural to integrate it with Plush. This integration enables users to request virtual machines (instead of physical machines) for running their experiments using a familiar interface.

Developing *plusher* was straightforward. Plush already exports a simple control interface through XML-RPC to integrate with resource providers. Plush requires providers to implement a small number of up-calls and down-calls. Up-calls allow resource providers to notify Plush of asynchronous events. For example, using down-calls Plush requests resources asynchronously so that it does not have to wait for resource allocation to complete before continuing. When the provider finishes allocating resources, it notifies Plush using an up-call.

To integrate Plush and Usher in *plusher*, we only needed to implement stubs for this XML-RPC interface in Usher. The XML-RPC stub uses the Client API to talk to the Usher controller. The XML-RPC stub acts as a proxy for authentication – it relays the authentication information (provided by users to Plush) to the controller before proceeding. When the requested virtual machines have been created, *plusher* returns a list of IP addresses to Plush. If the request fails, it returns an appropriate error message.

Usher Deployments

Next we describe two deployments of Usher that are in production use at different sites. The first deployment is for the UCSD CSE Systems and Networking research group, and the second deployment is at the Russian Research Center, Kurchatov Institute (RRC-KI). The two sites have very different usage models and computing environments. In describing these deployments, our goal is to illustrate the flexibility of Usher to meet different virtual machine management requirements and to concretely demonstrate how sites can extend Usher to achieve complex management goals. Usher does not force one to setup or manage their infrastructure as done by either of these two installations.

UCSD SysNet

The UCSD CSE Systems and Networking (SysNet) research group has been using Usher experimentally since June 2006 and for production since January 2007. The group consists of nine faculty, 50 graduate students, and a handful of research staff and undergraduate student researchers. The group has a strong focus on experimental networking and distributed systems research, and most projects require large numbers of machines in their research. As a result, the demand for machines far exceeds the supply of physical machines, and juggling physical machine allocations never satisfies all parties. However, for most of their lifetimes, virtual machines can satisfy the needs of nearly all projects: resource utilization is bursty with very low averages (1% or less), an ideal situation for multiplexing; virtualization overhead is an acceptable trade-off to the benefits Usher provides; and users have complete control over their clusters of virtual machines, and can fully customize their machine environments. Usher can also isolate machines, or even remove them from virtualization use, for particular circumstances (e.g., obtaining final experimental results for a paper deadline) and simply place them back under Usher management when the deadline passes.

At the time of this writing, we have staged 25 physical machines from our hardware cluster into Usher. On those machines, Usher has multiplexed up to 142 virtual machines in 26 virtual clusters, with an average of 63 VMs active at any given time. Our Usher controller runs on a Dell PowerEdge 1750 with a 2.8 GHz processor and 2 GB of physical memory. This system easily handles our workload. Although load is mostly dictated by plugin complexity, using the plugins discussed below, the Usher controller consumes less than 1 percent CPU on average (managing 75 virtual machines) with a memory footprint of approximately 20 MB. The Usher implementation is sufficiently reliable that we are now migrating the remainder of our user base from dedicated physical machines to virtual clusters, and Usher will soon manage all 130 physical nodes in our cluster.

Usher Usage

The straightforward ability to both easily create arbitrary numbers of virtual machines as well as destroy them has proved to be very useful, and the SysNet group has used this capability in a variety of ways. As expected, this ability has greatly eased demand for physical machines within the research group. Projects simply create VMs as necessary. Usher has also been used to create clusters of virtual machines for students in a networking course; each student can create a cluster on demand to experiment with a distributed protocol implementation. The group also previously reserved a set of physical machines for general login access (as opposed to reserved use by a specific research project). With Usher, a virtual cluster of convenience VMs now serves this purpose, and an alias with round-robin DNS provides a logical machine name for reference while distributing users among the VMs upon login. Even mundane tasks, such as experimenting with software installations or configurations, can benefit as well because the cost of creating a new machine is negligible. Rather than having to undo mistakes, a user can simply destroy a VM with an aborted configuration and start from scratch with a new one.

The SysNet group currently uses a simple policy module in Usher to determine the scheduling and

placement of VMs. This module relies upon monitoring data collected by the controller to make its decisions. It uses heuristics to place new VMs on lightly loaded physical machines, and to migrate VMs when a particular VM imposes sustained high load on a physical machine. Users are reasonably self-policing; they could always create large numbers of VMs to fully consume system resources, for example, but in practice do not. Eventually, as the utilization of physical machines increases to the point where VMs substantially interfere with each other, the group will interpret it as a signal that it is time to purchase additional hardware for the cluster.

This policy works well for the group, but of course is not necessarily suitable for all situations, such as the RRC-KI deployment described below in the RRC-KI section.

Support Services

Usher at UCSD uses plugins to automatically assigns IP addresses and VLANs to VMs, creates convenient domain name groupings for VMs in a virtual cluster, installs default user accounts, and provides structured VM-local, VC-global, and system-global file system access. These plugins interact with four support servers running as part of the site infrastructure.

SQL Server: The global state of the SysNet installation is kept in an SQL backing database. The database plugin mentioned in the Plugins subsection of the Implementation section provides access to the SQL database. Though most of the stored data is logging data stored for offline analysis of system performance and behavior, the SQL database does provide one required service: IP address management. The SysNet installation does not use DHCP to manage IP address ranges. The SysNet group manages several subnets, spanning multiple VLANs. Assigning ownership of arbitrary IP address ranges of these subnets to specified Usher users would be impossible using DHCP. As a result, an Usher plugin handles IP address management across these subnets.

LDAP Server: The SysNet LDAP plugin serves two purposes. First, it provides methods for managing and authenticating Usher users. Second, it provides the convenience of creating a branch in the LDAP database for each cluster an Usher user creates. This branch enables each VM the user creates to authenticate its users through the LDAP database.

This functionality provides a convenient authentication service to virtual cluster creators. First, it allows Usher users to use their Usher credentials as their VM login credentials since they are automatically added as a user in each cluster created. Since each cluster uses a different branch in the LDAP database, we use aliasing in LDAP to provide Usher users a single set of credentials. In addition, the plugin adds each Usher user to the ''admin'' group of each cluster the user creates. VM filesystems can then be configured to grant special privileges to this group (e.g., sudo privileges). This approach is convenient when using a read-only NFS root filesystem where no default root password is set.

Second, and more importantly, this arrangement addresses the cluster authentication problem for Usher users in the SysNet group. Authentication for clusters is challenging enough for experienced administrators. Delegating this problem to users is not only time consuming for them, but could lead to insecure VMs.

Creating a separate branch for each cluster allows Usher users to create accounts and groups for their clusters without burdening the Usher administrator with this task. This capability is especially conducive to collaborative work, a common case in a research lab setting. An administrator could easily be overwhelmed with management requests in a setting where users are free to create their own clusters, yet are unable to fully manage them. This approach pushes many mundane administrative tasks out to the users who have the incentive to create accounts on their VMs.

Allowing Usher users to modify the LDAP database requires careful configuration of the LDAP server, however. An LDAP server configuration file that allows Usher users to only manage branches which they own is included with the Usher source code. In addition, the Usher plugin for the LDAP server includes scripts for installation on a user's VM filesystems to modify cluster LDAP entries (i.e., to add, modify, or delete users and groups).

DNS Server: By default, Usher names VMs using the following naming scheme:

```
<requested VM name>.<creator's username>.
    <Usher system domain name>
```

where the Usher system domain name is specified in a configuration file read by the controller at startup. The DNS plugin adds this name for both forward and reverse name resolution for each VM.

NAS Server: Live migration of virtual machines requires a filesystem accessible by the VM at both the source and destination VMM. Since migration is a requirement of the SysNet installation, SysNet VMs must have their root filesystems provided via network-attached storage. These filesystems are served read-only NFS.

Serving the root filesystem read-only has multiple benefits. First, it is straightforward to keep filesystems across all running VMs synchronized and updated using read-only NFS root filesystems. Furthermore, an experienced administrator can manage this filesystem to ensure that it is secure (e.g., default firewall rules, minimal services started by default, latest security patches, etc.).

Second, since all VMs mount this filesystem, it is important that it be as responsive as possible. Ensuring that the NFS server serving this filesystem is read-

only helps improve performance. Furthermore, an administrator can configure a read-only NFS server to cache the entire filesystem in main memory. As a result, reads go to disk only once.

One issue with using a read-only root filesystem is that some files and directories on the filesystem must be writable at system startup. We solve this problem using a ramdisk for any files and directories which must be writable. Early in the boot process, these files and directories are copied into the ramdisk, then mounted using the --bind flag to make them writable.

Since the SysNet installation serves its root filesystems read-only, another NFS server provides persistent writable storage. The filesystem plugin initializes the filesystems to be mounted prior to starting up a new VM. This plugin creates the following directories for each VM on the group's read-write NFS server:

- **/net/global:** This directory is where users install or store anything they would like to have globally accessible by all of their clusters. The contents of /net/global is the same for all VMs a user creates.
- **/net/cluster:** This directory is where users can store files they want accessible by the current cluster only. The contents of /net/cluster is the same for all VMs in the same cluster.
- **/net/local:** This directory is unique to the current VM only. The contents of /net/local is different for every VM a user creates. Users can use this directory to set up services and configuration files specific to particular VMs.

Finally, all SysNet users are given a home directory. Automount takes care of mounting these directories upon login. Alternatively, Usher users can choose an alternate URI (stored in LDAP) for their home directory.

In each of /net/global, /net/cluster, and /net/local, there exists a System V init style directory structure in the etc directory. Startup scripts in the directory for the appropriate runlevel are run from these three locations after the regular system startup scripts run. With this configuration, even though users cannot write to the root filesystem to change startup scripts, they can have services started for their VMs at VM boot.

RRC-KI

Usher has also been deployed at the Russian Research Center, Kurchatov Institute (RRC-KI). The RRC-KI deployment demonstrates the flexibility of Usher to integrate with different computing environments, and to employ different resource utilization policies. Whereas the UCSD SysNet Usher deployment targeted a general-purpose computing environment, the RRC-KI Usher deployment targets a batch job execution system that provides guaranteed resources to jobs.

RRC-KI contributes part of its compute infrastructure to the Large Hadron Collider (LHC) Grid effort [1]. Scientists submit jobs to the system, which are scheduled via a batch job scheduler. Jobs are assigned to physical machines, and one machine only runs a single job at any time.

Measurements spanning over a year indicated that the overall utilization of machines in this system is fairly low [13]. While there were some long, compute intensive jobs, there was a large fraction of short, I/O driven jobs. Motivated by these measurements, the goal was to build a flexible job execution system that would improve the aggregate resource utilization of the cluster.

A straightforward approach is to multiplex several jobs on a single machine, and power down the unused machines. However, conventional process-based multiplexing on commodity operating systems is infeasible for a variety of reasons, some social and some technical: scientists want at least the appearance of absolute resource guarantees for their jobs; jobs often span multiple processes, which makes resource accounting and allocation challenging; and the number of physical machines needed depends on the workload and cannot be assigned *a priori*.

Virtual machines are a natural solution to this problem. Since each job gets its own isolated execution environment, resource accounting becomes easier for multi-process jobs. VMs also provide much stronger isolation guarantees than conventional processes. Each job can be given guaranteed resource reservations while still maintaining the abstraction of a physical machine. A trace-driven simulation showed that a VM-based infrastructure would enable significant savings [13].

One of the biggest challenges to this approach is management. For a VM-based infrastructure to scale, we need an automated system for deploying and managing virtual machines, a system that can schedule VMs in an intelligent manner, and migrate and place VMs to optimize utilization without sacrificing performance. A prototype system is currently being used at RRC-KI with Usher as the core management framework.

Central to this infrastructure is the *Policy Daemon* responsible for job scheduling and dynamically managing virtual machines (creation, migration, destruction) as a function of the current workload. The Policy Daemon uses the Usher Client API to monitor VM status and control VM resource utilization from a single control point using secure connections to the physical hosts. The current testbed comprises of a small number of nodes hosting production Grid jobs in the Usher-based environment with plans to expand the system to manage a few hundred nodes [15].

Adoption Considerations

Usher was designed to be a flexible, extensible framework for managing virtual cluster environments. However, our claims are supported only to the extent of what we have implemented and tested. At the time of this writing, we have used Usher with one VMM

implementation and the specific instances of plugins for UCSD and RRC-KI. For other sites to use Usher, if the existing plugins do not match their needs as implemented, then they will have to modify existing plugins or write their own. To this end, we do encourage Usher users to share any modified or new plugins they have implemented.

A final consideration is that of managing clusters of physical machines. Though the design of the framework does not preclude managing clusters of physical machines, to date, no plugins for managing physical clusters have been written.

Conclusions

Usher is an extensible, event-driven management system for clusters of virtual machines. The Usher core implements basic virtual machine and cluster management mechanisms, such as creating, destroying, and migrating VMs. Usher clients are applications that serve as user interfaces to the system, such as the interactive command-line shell *Ush*, as well as applications that use Usher as a foundation for creating and manipulating virtual machines for their own purposes. Usher supports customizable plugin modules for flexibly integrating Usher into other administrative services at a site, and for installing policies for the use, placement, and scheduling of virtual machines according to the site-specific requirements. Usher has been in production use both at UCSD and at the Russian Research Center in Kurchatov, Russia, and initial feedback from both users and administrators indicates that Usher is successfully achieving its goals.

Usher is free software distributed under the new BSD license. Source code, documentation, and tutorials are available at http://usher.ucsd.edu . Source code, configuration files, and initialization scripts for the UCSD plugins are also available for download at the site above.

Acknowledgments

The authors would like to thank Roman Kurakin for his insight, patches, and administration of Usher at RRC-KI. We also want to thank those people using Usher for their research at UCSD. Their feedback has been invaluable to the success of Usher in a research and academic environment. Finally, we would like to thank Alva Couch and our anonymous reviewers for their time and insightful comments regarding this paper. Support for this work was provided in part by NSF under CSR-PDOS Grant No. CNS-0615392 and the UCSD Center for Networked Systems.

Author Biographies

Marvin McNett is a Ph.D. student in the Systems and Networking group at the University of California, San Diego. His current research focus is virtual machine scheduling and management for efficient resource utilization. He is the original developer and current maintainer of the Usher project. Marvin expects to finish his Ph.D. in December, 2007.

Diwaker Gupta is a Ph.D. student in the Systems and Networking group at the University of California, San Diego. His current research interests include resource management and performance isolation mechanisms in virtual machines.

Amin Vahdat is a Professor in the Department of Computer Science and Engineering and the Director of the Center for Networked Systems at the University of California San Diego. He received his Ph.D. in Computer Science from UC Berkeley in 1998. Before joining UCSD in January 2004, he was on the faculty at Duke University from 1999-2003.

Geoffrey M. Voelker is an Associate Professor at the University of California at San Diego. His research interests include operating systems, distributed systems, networking, and wireless networks. He received a B.S. degree in Electrical Engineering and Computer Science from the University of California at Berkeley in 1992, and the M.S. and Ph.D. degrees in Computer Science and Engineering from the University of Washington in 1995 and 2000, respectively.

Bibliography

[1] *LCG project*, http://lcg.web.cern.ch/LCG/ .

[2] *Python*, http://www.python.org/ .

[3] *Twisted*, http://twistedmatrix.com/ .

[4] *VirtualCenter*, http://www.vmware.com/products/vi/vc/ .

[5] *XenEnterprise*, http://www.xensource.com/products/xen_enterprise/ .

[6] Albrecht, Jeannie, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat, "Remote Control: Distributed Application Configuration, Management, and Visualization with Plush," *Proceedings of the Twenty-first USENIX Large Installation System Administration Conference (LISA)*, November, 2007.

[7] Albrecht, Jeannie, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat, "PlanetLab Application Management Using Plush," *ACM Operating Systems Review (SIGOPS-OSR)*, Vol. 40, Num. 1, January, 2006.

[8] Anderson, Thomas E., David E. Culler, David A. Patterson, and the NOW Team, "A Case for Networks of Workstations: NOW," *IEEE Micro*, February, 1995.

[9] Appleby, K., S. Fakhouri, L. Fong, M. K. G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Océano – SLA-based Management of a Computing Utility," *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, May, 2001.

[10] Begnum, Kyrre, "Managing Large Networks of Virtual Machines," *Proceedings of the 20th Large*

Installation System Administration Conference, pp. 205-214, 2006.

[11] Bruno, G., M. J. Katz, F. D. Sacerdoti, and P. M. Papadopoulos, "Rolls: Modifying a Standard System Installer to Support User-customizable Cluster Frontend Appliances," *IEEE International Conference on Cluster Computing*, 2004.

[12] Chase, Jeffrey S., David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle, "Dynamic Virtual Clusters in a Grid Site Manager," *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[13] Cherkasova, Ludmila, Diwaker Gupta, Roman Kurakin, Vladimir Dobretsov, and Amin Vahdat, "Optimising Grid Site Manager Performance With Virtual Machines," *Proceedings of the 3rd USENIX Workshop on Real Large Distributed Systems (WORLDS)*, 2006.

[14] Grit, Laura, David Irwin, Aydan Yumerefendi, and Jeff Chase, "Harnessing Virtual Machine Resource Control for Job Management," *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November, 2006.

[15] Kurakin, Roman, Personal communication, Email dated 5/10/2007.

[16] Merkey, Phil, *Beowulf History*, http://www.beowulf.org/overview/history.html .

[17] Ruth, P., Junghwan Rhee, Dongyan Xu, R. Kennell, and S. Goasguen, "Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure," *IEEE International Conference on Autonomic Computing*, June, 2006.

[18] Sacerdoti, F. D., S. Chandra, and K. Bhatia, "Grid Systems Deployment and Management Using Rocks," *IEEE International Conference on Cluster Computing*, 2004.

[19] Vogels, Werner and Dan Dumitriu, "An Overview of the Galaxy Management Framework for Scalable Enterprise Cluster Computing," *Proceedings of the IEEE International Conference on Cluster Computing*, 2000.

[20] Waldspurger, Carl A., "Memory Resource Management in VMware ESX Server," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, December, 2002.

[21] White, Brian, 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," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 255-270, USENIX Association, Boston, MA, December, 2002.

[22] Wood, Timothy, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif, "Black-box and Gray-box Strategies for Virtual Machine Migration," *Proceedings the Fourth Symposium on Networked Systems Design and Implementation (NSDI)*, April, 2007.