Robust Leasing for Virtual Infrastructure

Aydan Yumerefendi, David Irwin, Varun Marupadi, Matt Sayler, Laura Grit, and Jeffrey Chase
Duke University
{aydan,irwin,varun,sayler,grit,chase}@cs.duke.edu

Abstract

Lease contracts are a powerful and general abstraction for negotiating and arbitrating control of shared networked resources. This paper addresses failure handling and recovery for leasing protocols and services. We present the design and implementation of a robust cooperative leasing service and illustrate its use for managing shared virtual clusters. The state update model is based on interacting recoverable state machines, an alternative to distributed transactions in which each participant recovers from a local log and then completes a self-synchronization protocol with other participants to restore each lease to a globally consistent state. We explore the impact of hidden component dependencies in virtual infrastructure, and the role of the leasing core in responding to failures involving plug-in extensions: policy modules and configuration handlers for hosted services and resource components.

1 Introduction

A distributed resource leasing system provides the means to procure computing resources from a collection of resource providers to be used by end users or application services. The resource leasing model has the potential to improve resource management and provide the means to share and use resources more efficiently. The emergence of a number of systems [5, 3, 12, 21, 2, 24, 19, 20, 26] with varying support for resource leasing is an attestation of the potential and popularity of the approach.

Resource leasing involves the cooperation of a number of servers, which may often be managed and controlled by different entities. Each server may fail independently and its failures may affect other servers and services. A robust resource leasing system must be able to deal with and tolerate failures of individual servers. In this paper we examine the problem of failure handling and recovery for resource leasing protocols and servers.

Our approach views lease management systems as coalitions of cooperating interacting state machines. Each state machine has well-defined states and rules. The state machines do not execute in isolation but rather interact with each other. The successful interaction of individual state machines results in resources being allocated to end users and application services. The distributed state machine can be considered as consisting of a fixed *core*, which describes the basic rules for interactions among individual machines, and a number of *plugin* extensions intended to accommodate different allocation algorithms or new resource types. Since individual state machines may represent self-interested actors with their own incentives, cooperation and interaction among state machines is limited.

This paper presents a general approach to maintaining consistency in a distributed state machine. We describe a failure and recovery model, which allows individual state machines to restart their actions after a crash and to self-stabilize [9] into a consistent state with the rest of the distributed state machine. The approach avoids the need for distributed transactions [8, 22, 28] but offers looser consistency guaranteed compared to transactional solutions. Others [16] have proposed similar solutions in the context of systems with "infinite scalability". The primary driving force in our case is the need for robustness in a distributed composition of loosely coupled state machines, where each machine is responsible for its own recovery.

We then show how to apply this general to the problem of robust resource leasing. Our implementation is based on the Shirako resource leasing toolkit [17] and the SHARP resource peering model [11]. We explore the specific requirements of resource leasing systems, and the challenges of plugin extensions. The paper chronicles our experience integrating and verifying recovery into Shirako and describes the tools we built to verify our implementation and the behavior plugin extensions.

This paper is organized as follows. Section 2 presents an overview of our context and approach. Section 3 describes the execution, failure, and recovery model of self-stabilising interacting state machines. We illustrate how the model applies to resource leasing in Section 4. Section 5 narrates our experience implementing and verifying lease recovery techniques. We position our work relative to others in Section 6 and Section 7 concludes.

2 Overview

Resource leases are a powerful abstraction for negotiating and arbitrating control over shared network resources [11, 17]. A lease is a contract among several actors in a networked system that guarantees access to a number of units of a given resource over a period of time. We describe leases in more detail in Section 4.

The leasing abstraction applies to any set of computing resources that is "virtualized" in the sense that it is partitionable as a measured quantity. For example, an allocated instance of a resource might comprise some amount of CPU power, memory, storage capability, and/or network capability, measured by some standard units, and with attributes to describe resources and the degree of isolation.

Virtualized infrastructure exports control points to partition and configure the underlying physical resources. For servers, virtual machine technology expands the range of privileged control options: the leading VM systems support live migration, checkpoint/restart, and fine-grained allocation of server resources as a measured and metered quantity (e.g., Xen [4, 7], VMware [30]).

Each server or *actor* in a leasing system maintains a local store of lease objects representing the state of the contracts it has entered into. The local lease store allows each participant to track its resource holdings and commitments independently, receive notifications about changes to the state of its leases, and to renew or cancel a lease contract or renegotiate its terms, as permitted by the particular contract. The leasing system may be thought of as a distributed lease manager: it defines a set of protocols and conventions for actors to communicate to negotiate and coordinate their lease contracts, and to invoke external programs to configure the leased resources and deploy guest applications or software environments on them.

System crashes, misbehaving extensions, and infrastructure failures may result in contract violations, and our goal is to design a robust resource leasing system that can effectively deal with such problems. In particular, the leasing system must be able to handle transient failures of actors or other elements in the leasing system with minimal interruption of service for lease holders. In this context, these techniques complement the standard mechanism to reclaim resources unilaterally on lease expiration, which is the "last line of defense" to protect availability of resources if a lease holder fails or communication is interrupted for an extended period [11].

2.1 Resource Leases

In this paper we focus on the SHARP resource leasing model [11] and our implementation of that model in the Shirako framework for distributed resource leasing [17]. Shirako consists of a compact *leasing core* that controls the workflow of obtaining and managing leased resources. The core specifies the general rules for message exchanges among the various servers that are involved in the distributed leasing process.

In addition, there are upcall interfaces for user-supplied plugin extension modules. Some extensions oversee individual resource configuration actions and status monitoring. Pluggable *policy controllers* manage policy-related decisions, for example, how to arbitrate among multiple requests. The behavior of the overall system results from the coordinated interactions among these elements.

This complex collection of interconnected components is vulnerable to failures or deviations of its individual elements. Viewing these elements as interacting state machines permits us to reason rigorously about failures and recovery. We based all aspects of the recovery architecture on a common model for recoverable cooperating state machines.

2.2 Crash Recovery

An essential element of robust leases is the durability and consistency of the lease manager. Resource leasing has some important requirements and properties that affect crash recovery.

Preserve service. Recovering from crash failures should strive to minimize violations of existing contracts: e.g., if some resources have already been allocated and the lease term has not expired, the end user should not observe any disruption of service[1]. Similarly, cross-actor state transitions should not fail immediately if one actor is currently unavailable. Since leases are valid for a period of time, all efforts must be taken to avoid declaring a failure until no other option is possible.

Local recovery. When a failure occurs it must be possible to recover locally using local information. Since the other members of the distributed state machine may be currently unavailable, recovery must limit the dependency on the availability of other actors. While it may be impossible to recover fully using only local information, local recovery should strive to recreate sufficient state to enable the actor to continue performing its role: for example, a failed broker must be able to continue issuing tickets (without violating prior contracts), despite the broker being unable to contact a service manager during the recovery phase.

Eventual consistency. Transitions in the lease state machine generally occur infrequently relative to the lease term; transitions mostly occur during initial setup and lease extensions. Therefore, some inconsistency in the distributed state machine is tolerable: individual actors may have inconsistent views of the the global state machine, but such inconsistencies are less critical. Thus, eventual consistency is appropriate for a resource lease: some temporary inconsistency can be exposed without compromising the system. Since leases expire over time, any inconsistency is guaranteed to be resolved; all leased resources are eventually released and can be reallocated.

Loose coupling. In general, the actors involved with a single lease are independently managed, autonomous, and self-interested. Therefore, recovery must assume as little as possible about the shared infrastructure among individual actors. This requirement makes distributed transac-

[1] Assuming that no global power failure or disaster occurred at the site or the service manager.

tions problematic as they depend on shared transaction coordinators and introduce additional dependencies among actors.

2.3 Robust Extensions

A robust leasing system must ensure that not only the leasing core, but also all extensions to the core behave as expected and do not violate the contractual agreements between the various actors involved in the leasing process. In a resource leasing system individual actors can develop their own extensions or use extensions supplied by third parties. Such extensions may have bugs or they can exhibit Byzantine failures. Even if the leasing core operates correctly, a single misbehaving extension can affect the correctness of the whole system.

At a low level, a policy extension is responsible for triggering state transitions of the local state machine for each lease managed by the actor. The rules for when and what transition to trigger are specific to each policy and the particular resource management protocol that it implements: e.g., first-come, first-served or earliest deadline first. These rules reflect the contract expressed by the lease and it is expected that each local state machine follows its specification correctly; a resource lease functions correctly, only if each of the involved actors cooperate and function correctly.

We deal with misbehaving policy extensions by identifying and enforcing a set of *lease invariants*. A lease invariant is a property of the resource leasing model and the resource leasing system, which is independent of the particular implementation of plugin extensions; every correct plugin extension must not violate any lease invariants. Lease invariants are essential for ensuring some minimal guarantees about the correctness of plugin extensions.

In the rest of this paper we are going to examine the problem of self-stabilizing state machines in a general context. Next we present how that model applies to resource leasing systems. Implementing recovery is a challenging and lengthy process and we will describe our experience and the tools we had to build to make our job easier.

3 Interacting Recoverable State Machines

Consider a system modeled as a graph or network of interconnected processes or components, each of which is a finite state machine (FSM), as discussed in more detail below. The state space of the complete system is the cross product of the states of its components. Some of the states in this space are deemed as desirable *consistent* states. Some of the states are reachable after some combination of faults; these *recovery states* may include some states that are not in the consistent set. The system is *self-stabilizing* if it converges from any recovery state to a consistent state in a finite number of message exchanges, and remains in a consistent state until the next fault occurs.

We define a class of simple self-stabilizing networks of cooperating state machines as the basis for a general execution and recovery model for loosely coupled networked systems. We define constraints and transitions to limit the number of recovery states, converge the system rapidly to a consistent state after a failure, and resume any interrupted activity. Our approach generalizes the failure and recovery model used in common distributed protocols, including two-phase and three-phase commit. It is useful as a basis for recovery in any system in which each request must execute a sequence of functions at one or more servers, and the servers are "loosely coupled" in the sense that they commit and recover their local states independently. Variants of this approach are commonly used as an alternative to distributed transactions, e.g., see the informal discussion in Helland [16] on the limitations of distributed transactions for large-scale systems.

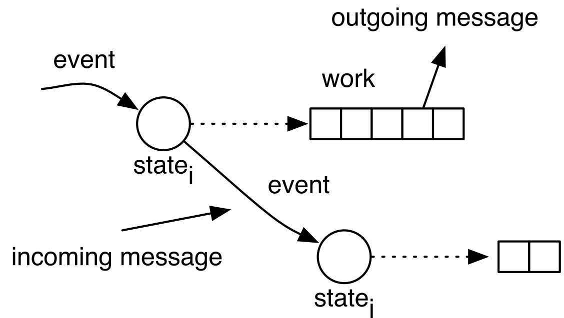

Figure 1: A local state machine. Each local state machine transition is triggered by an event. Events are generated locally or are the result of a receipt of a message from another state machine. Each transition into a state is associated with zero or more units of work, which are performed after the transition. Work can involve local operations (reads and/or updates) or sending of messages to other state machines.

3.1 State Machine Model

Figure 1 illustrates a single finite state machine (FSM). At any time, the FSM is in one of a finite set of states, as determined by the values of one or more state variables. The FSM transitions from one state to another in response to *events*. Using the standard definition, each FSM M is a tuple of the form $M = (X, \Sigma, \delta, x_0)$, where X is the set of states, x_0 is the initial state, Σ is the set of events, and δ is the transition function mapping each $(x_i, \sigma) \rightarrow x_j$ where $\sigma \in \Sigma$ and $x_i, x_j \in X$.

Events may be generated locally or externally. For example, a local clock may generate events as time passes, or some local activity may signal an event. The FSMs in the system cooperate by exchanging *messages*: each arriving message is an event. Events are consumed in order.

To apply the model to real systems, we supplement the FSM model of storage and computation in the following way. Each FSM maintains arbitrary *local data* in addition to its state variables. In addition, each FSM can perform arbitrary computation with each transition. Specifically, when the machine enters a state x_i as a result of an event σ, it initiates zero or more *tasks* to execute subprograms given by a fixed set $w(x_i)$ for each state x_i.

Most code specific to an application of the model—such as the lease manager example discussed in the next section—executes in the context of a task. A task may access the local data and any data associated with σ, e.g., the contents of a message. Tasks may signal local events to transition M, for example, to record the completion of some activity in M's state. Tasks may send messages to neighbors in the network of FSMs.

Tasks may execute asynchronously, or they may have sequencing and ordering dependencies with other tasks or the consumption of subsequent events, as discussed in Section 3.4. In our prototype, a server thread executes a state machine transition under the control of a global lock, and then releases the lock and executes the associated task list as a sequence of procedure calls.

3.2 Failure Model

The failure model is as follows:

- Each machine M may fail and discard its state and local data.
- Failures are fail-stop. If Byzantine faults may occur, then it is the responsibility of the program to detect and handle them: for example, we may suppose that a Byzantine fault causes M to generate messages that force its peers to transition into an identifiable failure state such that the resulting global state is in the consistent set.
- Suppose without loss of generality that a failed M eventually restarts, and network partitions eventually heal. In the lease manager example that is the focus of this paper, a machine that is unreachable for a long period of time is eventually abandoned by its peers as the leases expire.
- Sent messages are eventually delivered unless the sender fails before delivery or the receiver fails permanently. This property can be achieved by using a standard reliable communication protocol with acknowledgements, retransmissions, and session recovery for a restarted receiver.

3.3 Commitment and Recovery

Each finite state machine M is responsible for recovering its own state variables and local data when it restarts after a failure. To allow recovery, M logs each state transition and associated data to a durable store. M commits each transition into a state x_i *before* executing the tasks for $w(x_i)$ (write-ahead logging).

After M completes local recovery it executes a recovery transition to synchronize with its neighbors and restart any activities in progress. Specifically, the recovery system generates a recovery event σ_r after local recovery completes. For each state x_i, we add an additional *recovery transition* $(x_i, \sigma_r) \rightarrow x_i$. After completing the recovery transition into x_i, M starts the tasks for $w(x_i)$ in the usual fashion. These tasks redo any work that may have been left incomplete at the time of the crash, and send or resend any post-transition messages to neighboring FSMs.

The task redo approach places several requirements on the operation of the state machines and associated tasks. In particular, tasks must be:

- **Restartable.** To ensure that the restarted tasks $w(x_i)$ behave as expected, each log contains all information necessary to recover any data used by those tasks. Restarted tasks operate on the most recent non-recovery event σ that caused M to transition into x_i.
- **Idempotent.** Since a task may have completed some or all of its actions before the failure, actions taken by tasks must be *idempotent*: the result of the action is the same even if the action is invoked multiple times.

Since actions in $w(x_i)$ may send messages to neighboring state machines, some messages that were already received may be retransmitted during recovery. Logically, M can process retransmitted messages received in state x_i by executing the recovery transition for x_i. This means that M must detect duplicate messages and restart tasks in $w(x_i)$ as necessary. In particular, M must regenerate any reply messages issued by $w(x_i)$.

Our approach allows a special case to ease recovery for and FSM that is *subsidiary* to a single neighbor and can reattach to its dominant partner or *parent* after recovery. In this case, it is not necessary for the subsidiary machine M to log its state independently. M can recover all of its state from its parent, if the messages it sends to its parent are sufficient to reconstruct M's state variables and local state. M cannot recover independently of its parent, but this may be acceptable given that it can only interact with the outside world through its parent.

3.4 Tasks

Tasks play an essential role in driving the state machines. Messages sent by tasks are the basis for interaction among cooperating state machines in the model.

A key requirement of the model is that the application can manages the interactions among tasks outside of the FSM model, e.g., by reordering the work queue, cancelling tasks that are redundant or are overtaken by events, and using standard synchronization mechanisms to coordinate concurrent tasks. In particular, tasks must meet the

requirements of idempotence and restartability. Some actions are naturally idempotent and do not require special handling. To the extent that tasks have non-idempotent external side effects, they must record them in durable updates to local state, and suppress any duplicate or overlapping executions. For example, the lease manager assigns unique identifiers for task actions that affect the persistent state of external resources or guest applications. The identifiers are stored with the local data; if a task is restarted it is guaranteed to receive the same identifier. These identifiers allow detection and suppression of repeated actions and messages.

One limitation of the model is that it only restarts tasks associated with the last state transition. The work set Wx_i is associated with the state, and not with the transition into that state. On a recovery transition, asynchronous tasks associated with any previous state are not restarted. We chose this simple approach because in common cases it is not necessary to restart a task t if at the time of the failure M had already exited the state in which it initiated t. Consider these common examples:

- A task t prepares and sends a request to a neighbor, and M transitions on receiving the reply. It is not necessary to restart the task on recovery because it is known to have completed before the transition was taken.
- A task t prepares and sends a request to a neighbor, and M transitions on a command to cancel the request. It is not necessary to restart the request after recovery into the cancelled state.
- A task t performs a synchronous activity and signals a local event upon its completion, transitioning to a new state. If t is short, then events (e.g., incoming messages) may be blocked until the new state is reached.

If this simple single-state recovery model is too limiting, it is possible in general to transform a state machine to recover arbitrary asynchronous actions on recovery, by expanding the state space to maintain more history.

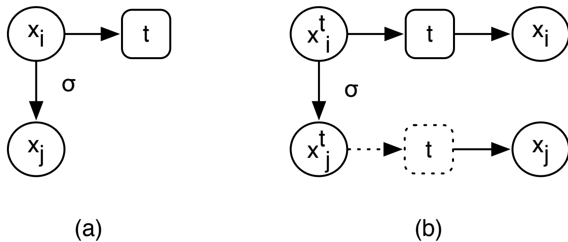

Figure 2: An example of transforming a state machine to add states that encode history. This technique is useful to apply the state machine recovery model to restart asynchronous tasks initiated by earlier transitions.

Consider the example depicted in Figure 2. Task t is started after entry into state x_i. An event σ could occur in state x_i and is to be handled by transitioning to state x_j. Suppose that it is not suitable to block processing of σ while waiting for t to complete. If M transitions to x_j and then fails in state x_j, then t is not restarted on recovery. One solution is to add two new states, x_i^t and x_j^t, to correspond to states x_i and x_j while encoding the fact that t is to be running. In the transformed state machine M', x_i^t replaces x_i, and M' can handle the event σ in state x_i^t by transitioning to the new state x_j^t. If t completes while in state x_i^t and x_j^t, it signals a local event to transition to the corresponding state x_i or x_j. The task sets $W(x_i^t)$ and $W(x_j^t)$ are identical to $W(x_i)$ and $W(x_j)$, except that they also start t if it is not already running. If M' fails while in state x_i^t and x_j^t, then t restarts on recovery: in other respects these states are identical to $W(x_i)$ and $W(x_j)$.

3.5 Summary

The essence of the recovery approach is that any local activities are restarted on recovery; the restarted tasks "continue where they left off" and resynchronize with neighboring state machines. While it is possible that the neighbors may have moved on, each failed M recovers at most one step out of sync with its state at the time of its failure. That is, at most one transition is incomplete, and this transition is restarted. If neighbors can take transitions without interaction with the failed machine M, then they must be able to interact with M when it recovers into the same state it was in at the time of the failure. In particular, neighboring machines are never seen to regress to a previous state: each FSM logs its transitions internally before they are visible externally.

4 Robust Leasing Services

This section presents the design of a robust distributed lease manager based on the principles of the recoverable cooperating state machine model presented in Section 3.

The key question for the state machine model is how we map a distributed program onto the state machine model. Exploring use of the model for robust lease management illustrates the value of a common model for recovery, and it shows how we can simplify the state space and interactions by decomposing the program into multiple FSMs with well-defined interactions. In particular, we represent user-supplied extension modules—policy controllers and configuration modules for particular resource types and guest applications—as separate FSMs interacting through well-defined interfaces and protocols.

We prototyped our approach in Shirako [17], a toolkit for building leasing services and clients based on the SHARP [11] resource leasing model. The lease manager applies the state machine model at the granularity of individual leases. Although Shirako includes a vari-

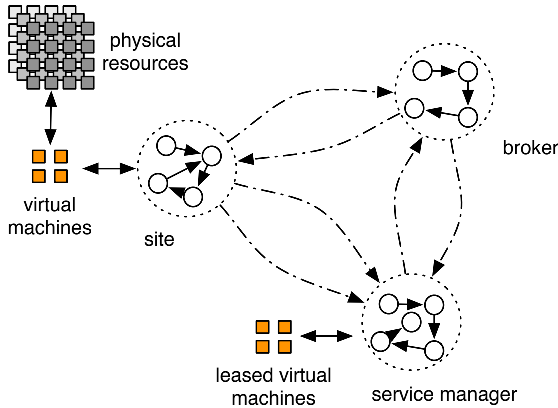

Figure 3: A resource lease is the joint effort of several interacting state machines. Each state machine is responsible for a specific part of the protocol: sites own the physical resources and create virtual machines on demand, service managers use leased resources and procure them on demand, brokers arbitrate the requests from multiple service managers and schedule resource usage. Each local state machine transition occurs as the result of local events or the interaction with another state machine. The combination of all state machines provides the global view of one distributed lease state machine.

ety of mechanisms for grouping and sequencing related leases [17], we may consider leases to be independent of one another for our purposes. In particular, leases may transition and recover their state independently of other leases. This property simplifies the state machines and allows for higher concurrency and independent recovery of leases managed by the same server. The remainder of our discussion applies to an individual lease.

4.1 Leases as Cooperating State Machines

Each lease is modeled as a network of interacting state machines (FSMs). Each FSM is built around the local lease object for one of the actors participating in the lease contract. The FSMs transition in response to timing events, protocol exchanges among the actors, changes in resource status or demand, and decisions by plug-in extension modules controlling resource management policy for each actor.

More specifically, a lease is a contract involving a resource consumer, a resource provider, and one more brokering intermediaries (Figure 3). We use terminology from the SHARP leasing model, although the terms used in related systems such as the emerging GENI architecture is slightly different. A *site authority* controls infrastructure resources in a particular sphere of authority, e.g., servers, storage units, network elements, or other components in a virtual data center or autonomous system under common ownership and control. A *service manager* procures lease contracts granting rights to use specific resources at one or more sites, and deploys a guest environment or software application on the resources it holds. Finally, a *broker* mediates resource discovery and arbitration by controlling the scheduling of resources at one or more sites over time. Each actor may manage large numbers of independent leases involving different participants.

The global state machine for a lease consists of the *local* state machines representing a given actor role. A lease has one local FSM in the actor performing each of these roles. Each actor is a server that maintains local lease objects and participates in the leasing protocols. The state transitions in each local FSM are driven by the logic of its specific role. The FSMs and protocol exchanges are managed by the Shirako core, which is essentially a library linked into each actor: it is a collection of classes and data structures supporting distributed lease management, state transition rules and functions for each role, upcall interfaces for user-supplied extension modules, and other functions needed to implement the actors.

In general, actors execute autonomously and are not mutually trusting. However, the different roles and actors may execute on the same physical node and thus to fail concurrently. Different components or elements of a distributed system may be represented as separate FSMs, even if they execute on the same node and share the same fate if that node crashes.

4.2 Basic Recovery

Upon restarting after a crash, each actor obtains a list of leases from its local data store and rebuilds the lease states and local data. Since lease objects are small, the leasing core commits the entire state of the lease on each state transition. Our current prototype uses a MySQL database for each actor. The committed state includes resource attributes and other data describing the state of the resources. In essence, the commit is a *checkpoint* of the lease state. Once the state is rebuilt, the recovery sequence executes the recovery transition for each lease FSM to restart any activities associated with its current state (see Section 3.3).

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

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

4.2.1 Expired Leases

Expired leases require special attention during recovery. If an actor is unavailable for an extended period of time, some of the leases that it manages will expire. During recovery, the actor can simply release the resources bound to expired leases. While this approach is correct, it may cause some instability in the system. For example, if an unavailable site closes an expired lease during recovery, the site may terminate access to resources legitimately used by a service manager. The service manager may have already extended the ticket and attempted to renew the lease, but since the site was unavailable, the site never received the extension request.

To avoid interruption of service for already existing leases our current policy implementations delay the releasing of resources until either an explicit close request is received, or the actor must release those resources to serve future requests. This "lazy close" policy ensures that temporary unavailability of either of the actors participating in a distributed lease does not terminate the lease prematurely. For increased robustness, it is possible in principle for site authorities to preserve snapshots of the leased resources (virtual machines and storage), so that if a lease is terminated prematurely, its leased resources can be recreated at a later time, with little or no loss of data.

4.3 Robust Extension Modules

The lease state machines and transitions for the actor roles are generic across all uses of the lease abstraction, so they may be implemented fully within the core. However, key aspects of lease management are specific to the types of resource being leased, the nature of the guest application to be deployed on those resources, and the resource management policies that control how lease requests are issued and handled.

In Shirako, these elements of lease management are implemented in replaceable user-supplied extension modules. The core invokes the extensions on specific transitions of the lease state machine. In this way the lease manager is designed to be neutral to resource types, guest software environments, and resource management policy.

Extension modules are viewed as sets of user-supplied tasks for "subsidiary" state machines (see Section 3.3) driven by the local lease FSM for an actor. These plug-in tasks complete management activities and/or scheduling decisions asynchronously, and signal a local event to transition the subsidiary FSM on completion. They may also monitor their internal states asynchronously, and may initiate local state transitions to expose failures or other conditions to the parent.

The parent lease state machine polls its subsidiaries to drive state transitions in the parent. This structure protects the system from user-supplied plugins that fail to complete, and it enables a simple recovery scheme for extension modules.

We consider three kinds of extension modules: policy controllers, resource handlers, and resource drivers.

4.3.1 Policy Controllers

Each actor invokes a *policy controller* periodically in response to clock events. These resource management policies interact with the lease state machines. The policy controllers may operate on collections of leases, create or close leases, and generate events on leases. For example, broker policies execute periodically to decide whether to grant or deny resource requests from service managers. If a request is granted, the policy task marks the local lease object, prompting the broker FSM to transition to a *ticketed* state and issue a *ticket* to the requesting service manager. Similarly, service managers incorporate policy controllers to formulate their requests for resources, and site authorities incorporate policy controllers to determine placement of ticketed requests on specific resource units according to the attributes of the request (e.g., placement of virtual machines on specific servers in a data center).

We have investigated several policy variants, including broker policies for arbitrating access to shared infrastructure based on fair-share scheduling, simple auctions, and value-maximizing heuristics. These policy issues are outside the scope of this paper. What is important here is the recovery of any local data maintained by the policy controller in the event of a failure. A second issue is selection of recovery transitions to cope with a misbehaving controller, since these extension modules are outside the control of the lease manager. Both issues are addressed in more detail below.

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

Since the controller's local data pertaining to each lease is committed only on the next transition of the parent FSM for that lease, decisions made by the policy are atomically durable with respect to each lease. That is, if the policy decision was recorded in a state transition of the parent

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

4.3.2 Resource Handlers and Drivers

The leasing abstraction can apply to a wide range of physical assets. Virtualized infrastructure exports control points to partition and configure the underlying resources. For example, virtual machine hypervisors export commands or interfaces to create and manage virtual machines hosted on servers. Server management coprocessors enable programmatic imaging and configuration of physical servers. Some network and storage elements export control actions to the network via protocols such as SNMP, Web Services protocols, or (more frequently) by remote login to a command-line interface from a privileged IP address.

In Shirako, these underlying control operations are invoked by *handler* and *resource driver* plugins invoked from the core on lease state transitions. Handlers and drivers are registered and selected according to the types and attributes of the specific resource units assigned to a lease. The various actors control the behavior of handlers and drivers by means of property lists passed through the leasing protocols and core.

Handlers are task scripts that run within an actor to execute configuration actions on a logical resource unit. In particular, the system defines prologue/epilogue interfaces for handler tasks to initialize and uninitialize a resource at the start and end of a lease term; these task invocations map to a registered handler action for the given resource type. The site authority invokes such tasks to *setup* and *teardown* each resource; the service management invokes tasks for each resource unit to *join* and *leave* the guest application or environment. We script handlers using *ant*, which can be invoked directly from the Java core and supports direct invocation of SNMP, LDAP, Web container management, and driver actions.

Many resource units can be viewed as *nodes*. Examples include physical servers, virtual machines, or any resource that can be controlled by scripts or programs running at some IP address, such as a domain-0 control interface for a virtual machine hypervisor. The Shirako package includes a module for manipulating node resources as a node state machine composed of states that indicate the status of the resource: priming, active, closing, failed, etc.

Handler tasks for nodes may invoke actions that run on the node itself. These control actions are invoked through a standard *node agent* that runs on a node (e.g., in a guest virtual machine or in a Xen dom0 control domain) and accepts SOAP/WS-Security or XMLRPC requests from an authorized actor on the network. A *node driver* is a packaged set of actions that run under the node agent to perform configuration actions that are specific to a resource type or guest software environments. The node agent accepts authenticated, authorized requests to install, upgrade, and invoke drivers. Shirako includes drivers for several types of resources and applications used in our testbed, e.g., Xen virtual machine monitors, volume cloning on local storage and Network Appliance file servers, and application packages such as the Rubis Web service and SGE batch job manager.

4.3.3 Node State Machine

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

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

Handlers and drivers must conform to the idempotence requirement. The basic principle of our approach is that the type-specific resource handlers should follow the *toggle principle*. From the perspective of the actor lease FSM, each resource is in either of two basic states: on or off (possibly failed). The handlers hide intermediate states. Although handlers execute asynchronously from the core, the actor FSM invokes the handlers for each resource unit in a serial order. The handler and the drivers it uses must ensure that the final state reflects the last action issued by the core, independent of any intermediate

states, incomplete operations, or transient failures. If handlers are deterministic and serial, then it is sufficient for actions to be (logically) serialized and idempotent at the driver level. That property may require persistent state in the driver (e.g., as in package managers, which typically adhere to the toggle principle); at minimum, it requires a persistent and consistent name space for any objects created by the driver (e.g., cloned storage luns or resource partitions). Drivers must also suppress redundant/duplicate operations triggered by the core to avoid disrupting the guest.

4.4 Misbehaving Extensions

Since extensions recovery is an essential component of the recovery of the leasing system, even a flawless implementation of the core may fail to recover the system if extensions are buggy or exhibit Byzantine failures. Misbehaving extensions can violate the contractual agreements represented in a lease. To protect the affected parties, a robust leasing system must take actions to ensure that misbehaviors do not remain undetected. Critically, extensions can misbehave not only during recovery but also during normal system execution.

We deal with the problem of misbehaving policy extensions in two stages. First, we identify a set of rules about the state transitions and actions of the lease state machine. These *lease invariants* are likely to apply to most policies and are generally independent of the specific leasing protocol a policy implements. Each lease invariant is then actively *verified* against the running system; any leases and actors who violate these invariants are exposed. Second, for additional protection, we enable the specification of protocol or policy-specific verifiers. These verifiers use their knowledge of the policy protocol to verify the policy's action.

Unlike policy extensions, resource handlers are not verified at runtime. Our current approach is to verify the behavior of resource handlers before adding them to a running system. This form of debugging ensures that no handlers with exposed bugs will be added to the system. However, it is still possible for a handler to exhibit Byzantine behavior at runtime. We return to this issue in Section 5.3.

5 Experience

A Shirako instance has been executing within our department for the last several months. Our deployment consists of one site authority, one broker, and a number of service managers. This particular deployment is used to allocate resources as virtual machines. Virtual machines are allocated specific slices of the host virtual machine monitor's memory and CPU cycles. Some service managers represent end users, who need virtual machines for experimentation and development. Other service managers represent dynamic services, who need resources for their execution. Some example dynamic services include a modified version of the SGE batch service and an in house virtual machine-based, job execution service called JAWS [14]. The resources required by each group changes dynamically over time, and Shirako is responsible for multiplexing all services and users onto the available physical resources.

5.1 Recovery Checker

To assist recovery implementation and testing we built a recovery checker. The checker receives a list of states to check. For each for each input state, the checker injects a failure that crashes the lease state machine (and the containing actor) at a particular point of execution, depending on the input state. The checker then restarts the actor and ensures that the affected lease, the actor, and the actor policy module, are recovered correctly.

The checker has two modes of execution. In the first, recovery is performed while the term of the lease is still active. If the term is sufficiently long, recovery should succeed and the lease should self-stabilize and behave as if no failure occurred. In the second, the checker deliberately delays recovery to ensure that lease terms are expired at the time of recovery. In such cases, the checker verifies that the resources bound to expired leases are released and reused to satisfy subsequent requests.

To implement the checker we added an event notification system to the leasing core. The notification system exposes key events to which testing and monitoring code can subscribe. Events include, but are not limited to, state machine transitions, sending and receiving of messages, resource handler invocations, etc. A subscriber for an event can consume the event, alter the objects emitted by the event, or can throw an exception to terminate the action that raised the event. In normal execution mode the system may catch and mask the exception. For recovery testing purposes we added a new exception type which is explicitly propagated up the call stack, so that we can cause the lease state machine and the actor to crash. In addition to the recovery checker, the notification system is used in a number of testing and verification tools we developed.

The combination of all local lease state machines results into a distributed state machine with approximately 360 states, from which about 30 are legal and reachable. For each of the reachable states, the recovery checker contains a condition that describes the point of execution at which failure must be injected. For most states, there is a single point, but for states that contain multiple tasks in their work list, there may be multiple points, each corresponding to the completion of a combination of tasks.

With the help of the recovery checker we discovered a number of bugs in our implementation. We can classify the observed bugs into several categories.

Serialization. As node and lease objects evolved over

Number	*Description*
1.	Every lease must have a valid ticket. Tickets should have at least one unit.
2.	The number of assigned nodes should equal the number of ticketed units.
3.	The number of ticketed units, and their properties, should change only on ticket extensions.
4.	The number of leased units, and their properties, should change only on lease extensions.
5.	Extend ticket requests must be made before the end of the lease.
6.	Brokers should respond to extend ticket requests before the end of the lease.
7.	Sites should not terminate a lease before the end of its term unless the service manager explicitly closes.
8.	Sites should not terminate a lease if they have received an extend lease request with a valid ticket.
9.	Resource handlers should terminate within a reasonable time.

Table 1: **Lease invariants. The table lists some of the most important invariants for ensuring that lease state machines function correctly. The listed invariants can expose a range of violations resulting from either code errors or infrastructure failures.**

Number	*Description*
1.	A virtual machine monitor with an active virtual machine reboots and fails to recreate the hosted virtual machines.
2.	NFS mounts become unavailable (automounter process fails). Critical data become inaccessible and drivers fail.
3.	Xen domain0 runs out of memory and kills the driver host server. Handlers fail to invoke drivers.
4.	Filesystem errors cause vmm hosts to reboot with a read only file system. Operations that require disk writes fail.
5.	An IP address is used concurrently by two hosts. Hosts become temporarily unavailable.

Table 2: **Infrastructure failures detected as violations of lease invariants.**

time new attributes and data structures were being added to each object. Often we would add a field, but would forget to update the serialization and deserialization code. As a result, during recovery, the object's snapshot is incomplete and key attributes are missing.

Checkpointing. Our failure approach requires that checkpoints occur concurrently with state transitions. The checker helped us discover cases, where, by accident or negligence, we would perform a checkpoint after starting the tasks associated with a given state. As a result, the checkpoint contains inconsistent data, as some state machine attributes represent updates from the tasks executed after the state machine transition.

Communication. Earlier versions of our implementation failed the lease state machine if one actor is unable to contact another actor. For example, a service manager would fail a lease if the site is currently unavailable, even though, the allocated ticket is far from expiration. We observed similar behavior when brokers and sites would send updates to service managers. The checker was instrumental for detecting bugs of this type and helped us ensure that actors "hang as long as they can"—if an actor is currently unavailable and a lease is still active, no failure should be declared unless absolutely necessary.

Idempotency. Key state machine tasks must be idempotent so that recovery can complete successfully. Due to a bug we did not serialize the outgoing sequence number for one of the lease classes. During recovery, the system was unable to recreate the required sequence number and the destination state machine ignored the retransmitted message, as it had sequence number smaller than the current receive sequence number.

Resource Handlers. Most of the bugs in this category are due to incorrect implementation of idempotency. Idempotent resource handlers took some time to implement correctly. New handlers would often introduce some instability due to the fact that different people would be involved in their development, and information of idempotency and how to achieve it in the specific context was not available. Another annoying source of errors in this category was due to updates of the underlying hardware and software controlled by a resource driver. Our system administrator would roll a routine update only for us to find out that the system no longer works. The transition from Xen 3.0 to 3.1 is one memorable example.

Policies. Bugs in policy recovery would often affect the recovery of a lease. The most common bugs was "stuck leases", leases which experience no progress, as if the policy "forgot" about them. Such bugs were easy to detect, the checker would fail with a timeout, but diagnosing them was difficult. This process was additionally complicated by the fact that our team was concurrently working on multiple policies. The different policy types also follow somewhat different rules for recovery, which further complicates the task of a developer who is trying to implement a correct policy. The recovery checker, however, provided a powerful tool with the help of which such problems were easily exposed.

5.2 Resource Handlers

The recovery architecture requires that care must be taken with resource handlers to meet the idempotency require-

ment. In the most common case an actor would fail, while one or more resource handlers are being executed. However, drivers and handlers can be engineered to be quite robust. Our current prototype offers drivers for the following resources:

- **Xen Virtual Machine Monitor.** The driver can create/destroy/and manage Xen [4] virtual machines.
- **ZFS File Server.** The driver can create/destroy file system images using ZFS [29] and file system-level cloning. Images are exported using the NFS protocol.
- **NetApp Filer.** The driver can create/destroy file system images using a Network Appliance filer using block-level cloning. Images are exported using the iSCSI protocol.
- **Logical Volume Manager.** The driver can create/destroy file system partition using the Linux Volume Manager (LVM) [23].
- **Virtual Machine.** The driver offers functions to manage a running virtual machine, e.g., register ssh host keys, add public IP addresses, install monitoring software, etc.
- **PlanetLab.** This handler installs and runs a private PlanetLab using MyPLC which instantiates new PlanetLab machines as applications need them.

The above resource drivers are used to construct resource handlers to deal with the creation, management, and destruction of leased resources. We have implemented a number of handlers, which integrate some or most of the above drivers. For example, our standard handler creates virtual machines with ZFS-backed NFS root disks. An alternative handler uses NetApp root disks exported using iSCSI. Each of these handlers can create additional local disk partitions to be used by the virtual machine, for example, as swap swap space or for extra storage.

In addition to authority-side handlers and drivers, we have developed a number of service manager handlers and drivers. These handlers and drivers deal with the installation and management of guest applications and environments. We have constructed idempotent handlers and drivers for the following services:

- **NFS.** The handler installs and manages an NFS server.
- **Fstress.** The handler installs and runs fstress, a file system performance tool.
- **Rubis.** The handler installs and runs the Rubis web application, a research implementation of an auction web service.
- **SGE.** The handler installs and manages a modified version of the Sun Grid Engine. The handler has multiple variants: one deals with setting up the master server, the other deals with setting an registering worker serves.
- **Globus.** The handler can install and run an instance of the Globus Toolkit.

To aid driver development, we developed a test harness for each driver and resource handler. The driver harness tests the idempotency of individual driver actions. All developers are required to run the associated tests before committing changes to drivers. The handler harness tests whether the composition of multiple driver invocations has some unexpected side effects. The test harness executes multiple instances of a single handler concurrently (using the same arguments) in an attempt to expose concurrency bugs. While this approach is not as exhaustive, as we would like it to be, it helped expose concurrency errors due to improper overlapping of subsequent driver invocations. The harness also helped us discover concurrency problems in third-party software we are using: a version of the Axis2 web services toolkit.

5.3 Enforcing Lease Invariants

Lease invariants are essential for verifying the runtime behavior of a leasing system. Over time our failure analysis would show that most bugs resulted in violations of a set of basic lease invariants. At first, those violations occurred during normal execution. Once we started spending more time on recovery, we would start observing such violations due to buggy recovery code. In most cases, policy extensions were to blame.

In an attempt to verify the runtime behavior of policies, both during normal execution and during recovery, we implemented a lease verification tool. The verification tool works similarly to the recovery checker: it subscribes to key events from the leasing core. As the system executes, the lease verifier stores observed events into a relational database. The tool uses its knowledge of high-level objects (leases and nodes) to transform the events into database records. Most lease invariants can then be expressed as SQL queries. Each lease invariant can be considered as continuous query that executes over the event stream generated by the leasing core.

Table 1 lists some of the most important lease invariants. These invariants target the basic guarantees of the leasing protocol and apply to all policies we have currently developed. In particular, the listed invariants look for violations of the leasing protocol which are expresses as lease with less resources than required, leases that last shorter than supposed to, and leases that fail to extend. We also consider timing: the time spent configuring resources should be bounded.

Lease invariants can expose a range of policy bugs but are not sufficient to verify if a policy is following its specification, i.e., it is servicing requests according to a specific algorithm. Such verification requires knowledge of the policy specification and is the subject of our future work. In general, the verification approach has application beyond debugging: coalitions of actors may use a common

auditing and verification system to ensure that each actor complies to its contractual obligations. A primary challenge in this context is ensuring that actors emit all events and that no "fake" events are emitted.

5.4 Infrastructure Failures

The lease invariants have been instrumental not only in detecting policy bugs, but they have also revealed problems caused by infrastructure failures. We use the term infrastructure to refer to all machinery within a site authority required to create and manage resources bound to resource leases. These include but are not limited to physical machines, storage servers, network links, routers, etc. Failures of an infrastructure component can propagate and eventually can cause a violation of one or more of the lease invariants. In particular, we have observed that violations of invariant 2, 4, and 9 are usually caused by a failure of an underlying infrastructure resource.

Failures can occur at many levels: each actor may control and manipulate a deep software stack on many different resources distributed across multiple infrastructure providers. For example, the root cause of an apparent failure of a guest component may be failure of the server hardware, hypervisor, guest OS kernel, a JVM running in a process, node agent, a configuration action issued either by the guest or the hosting site, or failure of the guest component itself—or it could be caused by interrupted network connectivity. Virtualization offers powerful management functions but it also introduces subtle infrastructure dependencies.

During our experience we have observed a number of infrastructure failures. Table 2 lists some of them. We discovered these failures while diagnosing the cause of a violation of a lease invariant.

Infrastructure failures affect existing resource leases and can result in contract violations. It is important that sites detect infrastructure problems so that they can attempt to repair the failure and provide continuous service. Similarly, infrastructure failures can affect the site's ability to service future requests. In either case, detecting, diagnosing, and repairing infrastructure failures is critical for the robustness of a resource leasing system.

In general, dealing with infrastructure failures proceeds in three stages. First, one must *detect* a problem. The problem may be detected at a high level, for example a lease invariant may be affected, or specific detectors may be written, which operate on a low level, e.g., by monitoring `syslog` events. The next step is to *diagnose* the problem from the observed symptoms. Diagnosis is a complex and challenging process and may rely on codebook techniques, machine learning, or interaction with an expert. Once diagnosis completes the cause of the problem must be eliminated by performing *repair* actions.

The choice of repair action can have serious consequences. For example, earlier versions of our system attempted to ensure that sites always try to instantiate the ticketed number of leased units even if unit creation results in an error; the policy would retry the create request until it succeeds, choosing different infrastructure elements as hosts. While this approach was successful with intermittent failures, a faulty virtual machine image was sufficient to exhaust the server resources at a site.

Diagnosis and selection of repair actions is a topic of future work. However, the cooperating state machine model does provide a range of "sledgehammers" and "scalpels" for repair. In particular, the policy can generate a recovery event for any state machine element. The recovery event forces it to execute its recovery transition and restart any actions associated with its current state. For example, if a hypervisor fails and restarts, a site authority has sufficient information in its database to determine what VMs are hosted on that resource. Executing the recovery transition in a node generates a fully qualified handler invocation to *setup* the lost virtual machine and/or *join* it to a guest.

6 Related Work

A number of systems for on-demand resource management exist [2, 3, 5, 12, 20, 24]. However, to the best of our knowledge, those systems do not implement a coherent way to recover from component failures short of lease termination. Leases have also been proposed as a mechanism for dealing with distributed system failures [13] but in our system, lease expiration is a recovery method of last resort. Our goal is to have leased resources be available even in the presence of failures rather than become unusable as soon as any component fails. This requirement is of great importance since leases represent contractual relationships, and involved parties can be held accountable for any contract violations.

There are many ways to deal with failures in distributed systems. One way is to make sure that all transactions are consistent. The approach we have taken is a generalization of the approaches taken by 2 and 3-phase commit protocols, paxos and others [8, 22, 28]. Since our leasing model can tolerate some temporary inconsistency, we use an alternative approach which has lower consistency guarantees, but avoids the need for tight coupling needed by distributed transactions. The actors in our system are self-interested and do not necessarily share the same incentives. As such, the sytem makes progress with a very loose coupling among the actors. This is different from a traditional distributed transaction, but is similar to alternative techniques proposed by Pat Helland [16].

Pat Helland makes the claim that distributed transactions are unsuited for systems that need to scale to extreme sizes because the performance and reliability costs of distributed transactions make such systems impractical. Instead, he proposes a set of new abstractions to build large scale distributed systems. The fundamental abstraction is

an "entity" which is a set of data that is also the scope of serializability. Transactions are local and operate on a single entity at a time. Our approach is similar but unlike Helland, who is motivated primarily by scalability, we are driven by the need for loose coupling between components.

Techniques like checkpointing and restart of transactions have been in use for many years [6, 15, 25]. Like previous techniques, the unit of recovery in our system is a transaction. There has also been previous work in recovering state machines [10, 18]. However, they focus on replicated state machines, where a number of identical state machines are trying to stay consistent. Our model consists of a number of interacting, cooperating but different state machines. The set of reachable states is a small subset of the cross product of the states of all participating state machines. Our interacting state machine model is self-stabilizing as defined by Dijkstra and others [1, 9, 27] because it is capable of reaching a consistent state from a non-consistent state.

7 Conclusion

This paper demonstrates how we handle failures and recovery in the Shirako leasing system. We show how the state update model takes advantage of the fact that we can tolerate temporary inconsistencies and avoids the need for expensive distributed transactions. We do this by confining the need to serialize transactions to a single lease.

Our model consists of interacting state machines that make independent transitions in response to events. Each state machine is independently recoverable and checkpoints its status after every state transition. In addition, logging of all incoming messages is performed before any acknowledgements are sent. In combination with idempotent actions, ensures that every state machine will eventually reach a consistent state.

We also show how extensions to the leasing system can be incorporated into the system by treating them as black boxes that implement fixed interfaces which include recovery operations. Recovery is driven by the core leasing system, but extensions are ultimately responsible for recovering their own state.

References

[1] Arora A. and Gouda M.. Closure and Convergence: A Foundation of Fault-Tolerant Computing. In*IEEE Transactions on Software Engineering* Volume 19(11):1015–1027, 1993.

[2] 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(6), April 2005.

[3] 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.

[4] 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.

[5] 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.

[6] Mohan C., Haderle D., Lindsay B., Pirahesh H. and Schwarz P.. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In*ACM Transactions on Database Systems* Volume 17(1):94–162, ACM Press New York, NY, USA, 1992.

[7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt and Andrew Warfield. Live Migration of Virtual Machines. In Proceedings of the *Symposium on Networked System Design and Implementation*, Boston, Massachusetts, May 2005.

[8] Skeen D. and Stonebraker M.. Formal Model of Crash Recovery in a Distributed System. In*IEEE Transactions on Software Engineering* Volume 9(3):219–228, 1983.

[9] Edsgar Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. In*Communications of the ACM*, 1974.

[10] Schneider F.B.. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. In*ACM Computing Surveys* 22(4), 1990.

[11] 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.

[12] Simson Garfinkel. Commodity Grid Computing with Amazon's S3 and EC2. In*;login: The USENIX Magazine* Volume 32(1):7-13, February 2007.

[13] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In*SIGOPS Operating Systems Review* 23(5), ACM Press, New York, NY, USA, 1989.

[14] Laura Grit, David Irwin, Varun Marupadi, Piyush Shivam, Aydan R. Yumerefendi, Jeff Chase and Jeannie Albrecht. Harnessing Virtual Machine Resource Control for Job Management. In Proceedings of the *Workshop on System-level Virtualization for High Performance Computing*, Lisbon, Portugal, March 2007.

[15] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. In*ACM Computing Surveys* Volume 15(4):287–317, 1983.

[16] Pat Helland. Life Beyond Distributed Transactions: An Apostate's Opinion. In Proceedings of the *Conference on*

Innovative Data Systems Research, pages 132-141, January 2007.

[17] David Irwin, Jeffrey Chase, Laura Grit, Aydan Yumerefendi, David Becker and Kenneth G. Yocum. Sharing Networked Resources with Brokered Leases. In Proceedings of the *USENIX Annual Technical Conference*, pages 199-212, Boston, Massachusetts, June 2006.

[18] Rushby J.. Reconfiguration and Transient Recovery in State-Machine Architectures. In*Fault Tolerant Computing Symposium* 26:6–15, 1996.

[19] 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.

[20] 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* Volume 37(11):38-46, November 2004.

[21] Katarzyna Keahey, Karl Doering and Ian Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In Proceedings of the *IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, Pennsylvania, November 2004.

[22] Leslie Lamport. The Part-Time Parliament. In*ACM Transactions on Computer Systems* 16(2), ACM Press, New York, NY, USA, 1998.

[23] "Logical Volume Management". http://sourceware.org/lvm2.

[24] 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.

[25] Strom R. and Yemini S.. Optimistic Recovery in Distributed Systems. In*ACM Transactions on Computer Systems (TOCS)* Volume 3(3):204–226, ACM Press New York, NY, USA, 1985.

[26] 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.

[27] Marco Schneider. Self-Stabilization. In*ACM Computing Survey* Volume 25(1):45–67, ACM Press, New York, NY, USA, 1993.

[28] Dale Skeen. Nonblocking Commit Protocols. In Proceedings of the *1981 ACM SIGMOD international conference on Management of data*, ACM Press, New York, NY, USA, 1981.

[29] "ZFS filesystem on Solaris". http://www.sun.com/2004-0914/feature/.

[30] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In Proceedings of the *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.