Automated Incident Management for a Platform-as-a-Service Cloud

Soumitra (Ronnie) Sarkar, Ruchi Mahindru, Rafah A. Hosn, Norbert Vogl, HariGovind V. Ramasamy
IBM T. J. Watson Research Center, New York, USA
{sarkar, rmahindr, rhosn, vogl, hvramasa} at us.ibm.com

Abstract

Cloud-based offerings such as Infrastructure-as-a-service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS), are being delivered by various vendors at highly competitive prices to encourage a paradigm shift to utility computing. To optimize the operational costs of managing an IBM Cloud-based PaaS offering, a two-pronged approach has been adopted: simplification of enterprise-class data center management processes currently used in IBM's Global Services Strategic Outsourcing accounts, and automation of the simplified processes. This paper describes a framework that the authors have developed to deliver an integrated monitoring and event correlation system, and an event-driven Automated Incident Management System, for IBM's Smart Business Dev/Test Cloud offering.

1. Introduction

Providers of publicly accessible Cloud services (e.g. Amazon Elastic Compute Cloud (EC2) [1], Windows Azure [2] and Google App Engine [3]) are adopting aggressive pricing strategies to make the move to a utility computing model more attractive. Management process simplification is a key enabler for reducing the operational expenses of Cloud providers. As an example of process simplification, consider Amazon EC2, where service level agreements (SLAs) provide guarantees of success for the dynamic provisioning of a virtual machine (VM), but there are no guarantees about the mean time between failures (MTBF) for a provisioned VM. For most public XaaS Cloud-based offerings, a provisioned service may crash if there is any problem in the underlying infrastructure (e.g., server, hypervisor, storage, etc.) without violating Cloud provider SLAs. Typically, there is also no provider support for problem determination other than informal forum-based support.

Automation of Cloud management processes is the other key enabler of cost reduction, especially when leveraging management process simplification. For example, problem determination (root cause analysis) can be a complex activity requiring expert knowledge that cannot completely be replaced by automation. However if problem determination is eliminated, then it reduces the challenges of automation while also reducing labor costs.

Monitoring and *incident and problem management* (IPM) processes account for a significant portion of data center operational costs. During the design of IBM's *Smart Business Dev/Test Cloud* (SBDTC), a PaaS offering, the authors were deeply involved in the design of the monitoring system for the Cloud infrastructure and also led the design and implementation of the *Automated Incident Management System* (AIMS). The goal of the project was to design a process that is highly optimized for a Cloud-based offering. This paper describes the technical challenges addressed by the design, and our experiences in deploying the system in a Cloud.

The rest of the paper is organized as follows. Section 2 describes the overall architecture of the system, Section 3 describes the monitoring and event management systems, Section 4 describes the design of AIMS, Section 5 describes our deployment experience, Section 6 describes related work and Section 7 concludes the paper.

2. System Architecture

Figure 1 shows the overall architecture of the current system. All management components and infrastructure elements shown constitute a unit of management or a *cell*. The infrastructure layer represents the different types of Information Technology (IT) elements (components) that support end user applications, namely servers running host operating systems and hypervisors such as KVM [4], Xen [5] or VMWare [6]) on which VMs can be provisioned. The VMs in turn run their own operating systems as well as middleware (e.g, Web application servers and database systems) which host customer applications. Shared storage (e.g., NFS) servers provide access to standard as well as customer-created *images* from which new VMs can be provisioned, and also host persistent VM storage ("disks") which can survive VM deprovisioning (e.g., functionality equivalent to Amazon's Elastic Block Storage). Examples of network elements are routers and switches.

The monitoring layer comprises of systems used to collect data from IT systems in real time in order to detect incidents. Different monitoring approaches have been adopted for different IT elements – e.g., custom agent-based systems for servers, SNMP-based monitoring for network elements, etc.

The event management framework uses an *Event Aggregation and Correlation hub* which receives events

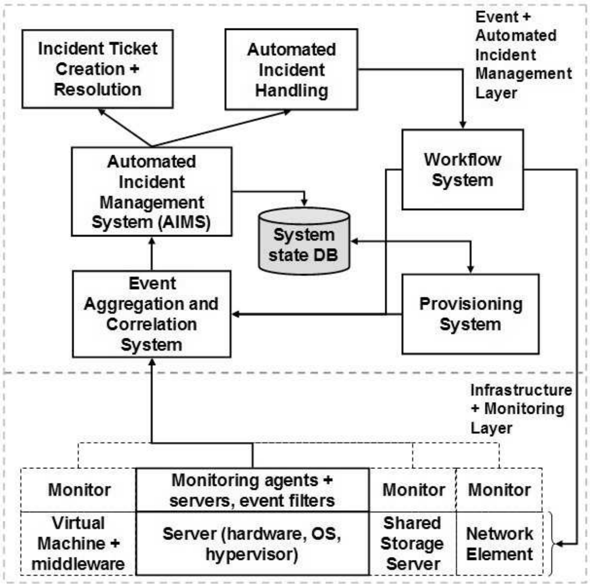

Event + Automated Incident Management Layer
Incident Ticket Creation + Resolution
Automated Incident Handling
Automated Incident Management System (AIMS)
Workflow System
System state DB
Event Aggregation and Correlation System
Provisioning System
Infrastructure + Monitoring Layer
Monitor
Monitoring agents + servers, event filters
Monitor
Monitor
Virtual Machine + middleware
Server (hardware, OS, hypervisor)
Shared Storage Server
Network Element

Figure 1: Architecture of Monitoring and AIMS

indicating "unusual" conditions from the monitoring layer. The same hub can receive events from the provisioning system to learn about the "birth" of new IT elements, and also receives status of *workflows* scheduled by AIMS – a workflow being an automation module that performs a corrective action against an IT element. The AIMS component receives events tagged as being actionable from the aggregation/correlation system, and based on policies, either ignores the event, creates or resolves a ticket, or takes an automated corrective action by scheduling a workflow.

3. Monitoring and Event Management

The key issues addressed in the design of the monitoring system are: what metrics to monitor, the granularity of monitored data, and whether the monitoring is continuous or staged (in the latter approach, when a fault is suspected, more detailed monitoring is enabled). Event management is also a key concern where issues that have to be addressed are: the design of appropriate filters between continuous monitoring of system sensors and events forwarded to AIMS as fault indications (incidents), the correlation of multiple events with a common root cause, detection of event storms that are caused by systemic rather than localized failures, and the prioritized handling of events forwarded to AIMS. In the current deployed system, the majority of the monitoring and incident management capability is focused on servers, and that will be the primary focus of the next two sections.

The monitoring system is based on core capabilities provided by existing IBM systems, which operate using specialized software agents in servers, and also in "agentless" mode (e.g., by querying SNMP MIBs).

Non-hardware monitoring of servers is achieved using software agents which track OS-level metrics such as CPU utilization, paging rate, etc., and also monitor system logs for error message patterns (e.g. OOPS). The server-based agents periodically communicate with a central server to report metrics of interest. The monitoring server can be customized with rules which are evaluated whenever a sampled sensor value is reported by an agent. The triggering of a rule results in an event being forwarded to the central event management hub. Several custom rules that perform a first level filtering of OS-level metrics to identify events of interest have been defined. For example, a "CPU critical" event is forwarded to the hub when the OS agent reports that the average CPU utilization exceeds 95% over 10 consecutive sampling intervals.

Hardware monitoring is performed by a monitoring server which communicates with the service processor running on each physical server to monitor its hardware components. Failures (e.g., of the CPU/core or the fan) as well as failure prediction indications (e.g., the predicted failure of a disk drive, based on manufacturer-supplied algorithms implemented in firmware) are monitored by the service processor, and this status is periodically collected by the server. Customized rules have also been written for hardware metrics in order to identify events of potential interest to AIMS. Network monitoring is based on standard SNMP-based techniques and ICMP pings to discover network topology and monitor the health of network elements such as routers, switches and servers.

The event aggregation and correlation system is the hub that receives events generated by all monitoring components as well as other components such as the provisioning system and workflows. The system stores events which have been received for some window of time. Custom code modules (triggers) can be defined; these are activated whenever a new event is received, and can reason over the event history to perform event aggregation, correlation and suppression. Triggers provide a second level of filtering on core monitored metrics to determine whether an event is significant enough to be forwarded to AIMS to act upon. For example, a custom trigger ensures that at least X out of the last Y consecutive network ping reports for a server indicate a failure before the failure event is forwarded to AIMS.

4. Automated Incident Management

For the automated handling of events in AIMS, the core principle that has been adopted is to classify the highest priority event into an appropriate "class" and take a simple corrective action in response when applicable (e.g., restart a failed process or an interface), followed by increasingly more obtrusive actions (reboot, reimage, mark-as-failed) if simple actions do not fix the fault. Additional aspects of automation include the creation and resolution (when possible), of problem tickets describing the incident, and influencing the *placement engine* to prevent provisioning of VMs on faulty or overloaded hypervisors.

The authors developed a framework for modeling event-based automation policies using *Finite State Machines* (FSMs) to model infrastructure elements, enhanced with state persistence to maintain a holistic view of the cell, maintenance of event-action history for improved decision making, and fault tolerance for dependability, features typically unavailable in off-the-shelf management systems. The AIMS framework is described below.

4.1. AIMS Framework

The framework we developed for this environment is based on modeling key infrastructure elements – servers (hardware and hypervisor software providing virtualization services), network components, and shared storage components – as FSMs. An FSM *type* definition is used to model event handling policies for each type of IT element on the Cloud. In our initial deployment, all encoded policies were based on expert knowledge, but as incident data from the field is analyzed, we expect to refine them and add new ones. The current system handles approximately 50 different events.

Each FSM *instance* is used to track state transitions of an infrastructure element from "birth" (deployment) to "death" (removal), based on events received from the event aggregation and correlation system. An event that represents incidents (faults) in a given IT element is acted upon based on the type of event, the current state of the IT element, the history of past events and actions taken on that system, and the policies associated with the resulting state transition encoded in the FSM.

As indicated in Figure 1, not all events indicate possible faults. An automated system that provisions IT infrastructure elements can send an event which, when forwarded to AIMS, results in a new FSM instance being created. Another instance of a non-fault event is workflow-related. Workflows which perform corrective actions in response to events are scheduled for execution by a workflow system, adopting an asynchronous execution model. The status of workflows – Success, Failure, Validation_Failure – is indicated by sending events to the aggregation/correlation system, which are subsequently forwarded to AIMS. These events affect the transitions of the appropriate FSM instance as do “fault” events, as encoded in the FSM definition.

The asynchronous nature of workflow execution can result in unpredictable delays in the actual execution of a workflow. To mitigate the effects of delayed execution, a *workflow validation* mechanism has been developed. Each workflow, when first scheduled for execution, validates that its execution is still relevant before performing the main workflow logic. This prevents situations such as a delayed workflow taking a corrective action after a system administrator has performed a manual action that has fixed the problem the workflow was scheduled to fix. The workflow validation protocol by design bypasses the event management system, since that framework can introduce unpredictable delays. It is based on a lightweight, customized, synchronous, protocol in our initial deployment, but could be (in principle) REST-based.

4.2. Modeling Incident Management Policies with Finite State Machines

In the currently deployed system, we focused primarily on the automated management of incidents on servers running hypervisors. The rest of the paper will therefore focus on server incident management. For modeling policies (defined by domain experts) for automatically handling incidents indicated by events, we defined an FSM that represents the states that a server goes through, from “birth” to “death” as different types of events are received, and the rules that govern state transitions.

FSMs provide a modeling mechanism that is a good fit for the task of representing policies for managing incidents (failures and performance problems), where events are the key drivers for taking corrective actions. A server goes through multiple states during its period of operation in a data center, “birth” representing its initial setup by a provisioning system, and “death” representing a state of failure where it is unable to support the provisioning and execution of VMs, and repeated automated actions taken to revive the server have failed. Bounded by the initial and final states, a server goes through multiple operational states, and the actions taken in response to an event are related to the state of the server.

Figure 2 depicts the FSM that was defined for modeling server incident management policies. An *instance* of a server FSM is created when an event is received announcing the availability of a new server. The FSM immediately transitions to a WAIT_FOR_SERVER_UP state where a workflow is run to check that the server is in a state where automated corrective actions can be run. Once that workflow completes successfully – indication of which appears as a status event from the workflow itself – the FSM transitions to the TEST_HYPERVISOR state, where another workflow starts a test VM which runs a set of basic stability tests on the hypervisor for a fixed period of time. Any fault events received while the server is in that state are treated more seriously than when the server is in the HEALTHY state. Once the server has been in the TEST_HYPERVISOR state for a test period, it transitions (in response to a timer event) to the HEALTHY state.

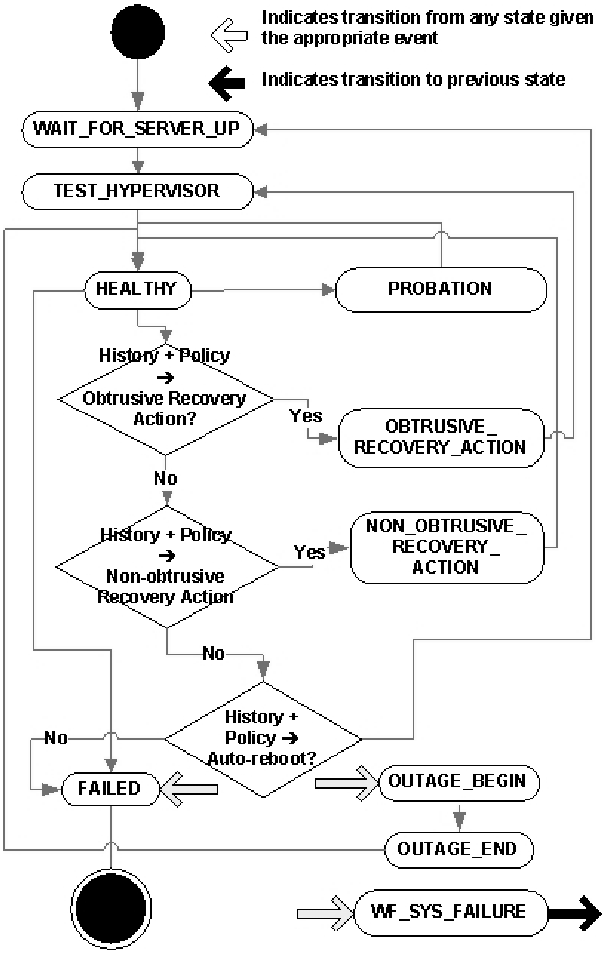

Indicates transition from any state given the appropriate event
Indicates transition to previous state
WAIT_FOR_SERVER_UP
TEST_HYPERVISOR
HEALTHY
PROBATION
History + Policy ➔ Obtrusive Recovery Action?
Yes
OBTRUSIVE_ RECOVERY_ACTION
No
History + Policy ➔ Non-obtrusive Recovery Action
Yes
NON_OBTRUSIVE_ RECOVERY_ ACTION
No
History + Policy ➔ Auto-reboot?
No
FAILED
OUTAGE_BEGIN
OUTAGE_END
WF_SYS_FAILURE

Figure 2: Server FSM Model

Transitions from the HEALTHY state occur due to three broad categories of events. The first category causes a transition to the FAILED state if a critical and unrecoverable error indication is received. The second category of events represents operational problems other than performance issues. In response, the FSM engine optionally consults the *history* of events received and actions taken in the recent past on this specific server and based on FSM state transition rules, either a workflow to perform a non-obtrusive recovery action is scheduled (e.g, restarting a failed system process), or a more obtrusive action such as a reboot is scheduled. The third category of events represents performance problems – such as a "CPU critical" event indicating a hypervisor-level overload. In that case, the FSM instance transitions to the PROBATION state, and remains there until another event from the server indicates that the performance problem has subsided. Some events are known to cause an automatic server reboot by the BIOS, in which case the FSM transitions to WAIT_FOR_SERVER_UP until the server is operational again.

Besides scheduling workflows in response to state transitions, tickets are created and resolved when possible. Whenever a server is deemed to be in a "recycling" state (e.g., being rebooted), or when further VM provisioning on the server should be inhibited (e.g., in the PROBATION state, AIMS updates the server's "provisionability" status in a System State database (see Figure 1) shared with the VM provisioning system, allowing AIMS to influence VM placement decisions. Additionally, certain events are purged from the queue in the OBTRUSIVE_RECOVERY_ACTION state since the system will be rebooted.

The server FSM also embodies some self-healing policies, and the ability to deal with events which have a broader scope than that of a single server. For example, the WF_SYS_FAILURE is a state that the FSM instance transitions to if the workflow engine itself is not operational and the scheduling operation fails. That state is modeled to periodically "ping" the workflow engine, upon success of which, control returns to the previous state and the workflow is rescheduled.

OUTAGE_BEGIN and OUTAGE_END represent special states which are entered when AIMS is informed that a "global" set of events are occurring (or are expected) due to a shutdown of all monitored IT elements in the data center, or their subsequent restart, respectively. In the OUTAGE_BEGIN state, any event received for the server is ignored, except for tickets being created for critical hardware failure events. In OUTAGE_END state, the target is pinged and when reachable, the FSM transitions to HEALTHY state.

4.2 Scalable and Persistent FSM Engine

With a centralized event-based AIMS component handling incidents on all IT elements, robustness of the system is paramount since it operates in "low touch" mode with reduced system administrator oversight. To address that goal, we have developed an FSM execution engine that adds the properties of *scalability* and *persistence* to the FSM runtime system.

A typical FSM engine (e.g., the runtime environment of SCXML [7]) requires each FSM instance to be in memory throughout its execution, and if the engine terminates prior to all FSM instances completing their executions (e.g., due to a fault in the server where it is running), then all runtime state is lost. Such an environment is not suitable for building an FSM-based AIMS component where (1) one may need to run thousands of FSM instances in parallel (each instance representing the operational state of a server for example), and (2), the life cycle of a typical FSM instance may span months or years (the average operational life span of a server), during which, the AIMS component may be shut down due to planned or unplanned outage of the management server where it is executing. Figure 3 depicts

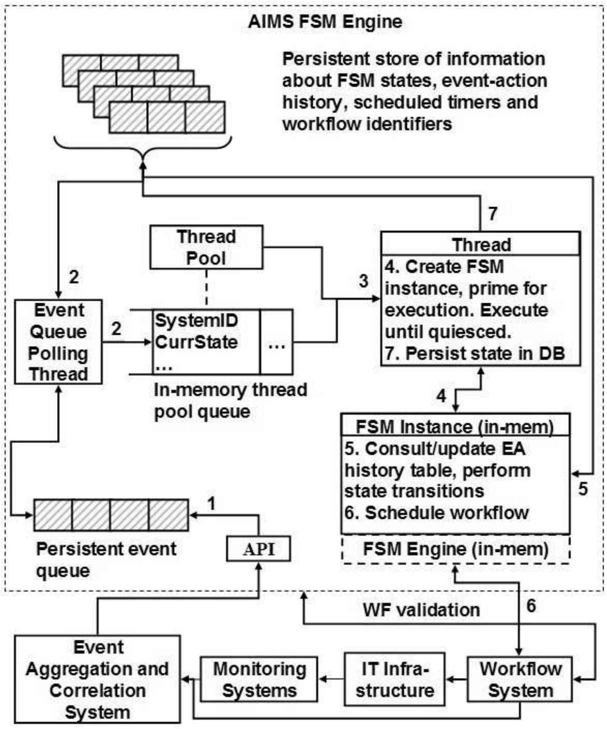

AIMS FSM Engine
Persistent store of information about FSM states, event-action history, scheduled timers and workflow identifiers
7
Thread Pool
Thread
4. Create FSM instance, prime for execution. Execute until quiesced.
7. Persist state in DB
2
3
Event Queue Polling Thread
2
SystemID
CurrState
...
...
In-memory thread pool queue
4
FSM Instance (in-mem)
5. Consult/update EA history table, perform state transitions
6. Schedule workflow
5
1
Persistent event queue
API
FSM Engine (in-mem)
WF validation
6
Event Aggregation and Correlation System
Monitoring Systems
IT Infra-structure
Workflow System

Figure 3: Scalable and Persistent FSM Engine

the design of a runtime system which implements these properties. The current system addresses planned shutdown but not *fault tolerance* of unplanned outages, the implementation of which is part of future work.

The design involves the use of persistent storage to record information about events received, key state transitions executed by FSM instances, workflows scheduled, and timers set, allowing the system to be shut down without losing information. The design also facilitates scalability since any given FSM instance does not need to continuously reside in memory to process an event received for that server.

The following description references Figure 3, and the steps refer to the numbered arrows in the figure. Upon receipt of an event from an external component, AIMS persists the event into a persistent Event Queue (step 1). An event queue polling thread periodically examines the queue and selects the highest priority event from the queue for which an FSM instance is not already executing (mutual exclusion is important since parallel state transitions are not supported). In step 2, the polling thread schedules the FSM instance corresponding to that IT element for execution by (1) extracting the last known state of the FSM instance from persistent storage and (2) creating an entry in the work item queue of an in-memory Thread Pool service that contains details such as the FSM state, the IT element's unique identifier (e.g., its IP address) and the event received. In step 3, a thread in the pool becomes available, and in step 4, the FSM scheduling logic uses the next work item to create an FSM instance, "primes" it with values necessary to continue execution from the last known state, and executes the FSM (in memory) until it reaches a state where it cannot proceed any further, at which point the FSM instance returns control and its current state is queried and persisted in persistent storage until the next event is received (step 7).

In steps 5 and 6, the in-memory FSM engine executes the transitions encoded in the FSM definition, leveraging entries in the Event-Action History store maintained by our runtime system as a source of additional input for determining what state transitions to take. Finally, when a workflow has to be scheduled to take a corrective action if the transition logic demands, a unique workflow identifier (ID) is generated, persisted in the Workflow ID store, and the workflow is scheduled with the ID passed to it along with other domain-specific parameters. As long as the workflow ID in the table is still tagged as being valid, the workflow validation step will succeed. FSM transitions can mark the ID as being invalid, as can fault tolerance-related recovery logic which is referenced in Section 7.

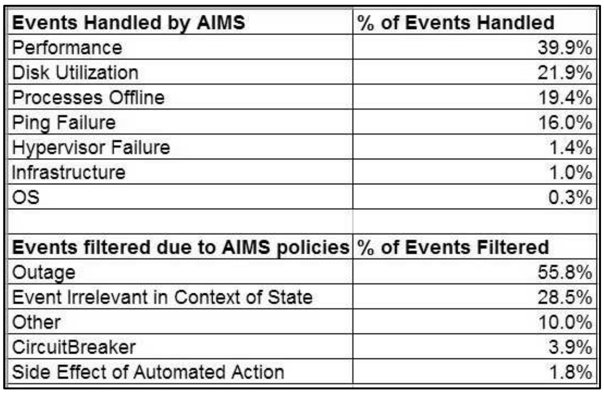
In the current implementation, the in-memory FSM engine is home-grown, but we are building an alternate implementation using SCXML.

5. Deployment Experience

The AIMS component is currently deployed in multiple cells in the IBM Smart Business Dev/Test Cloud and has been in operation for a few months. The scalability and persistence features of AIMS have met the design goals during steady state operations as well as a few planned outages. The FSM has evolved over multiple deployments. An event-based automated system can end up taking actions on a large number of IT elements in response to a condition that generates a lot of events such as a faulty rule definition or an outage. Based on system administrator feedback on earlier deployments, AIMS has introduced the notion of a "circuit breaker", which limits the number of IT elements on which obtrusive actions are taken based on an administrator-defined limit.

Almost no hardware problems have been reported to date by the monitoring system. Intermittent performance problems were reported by a few servers and the policies encoded in the FSM – to transition the server to PROBATION state and prevent further VM provisioning - worked well. For one category of performance problems, analysis of events archived in the warehouse led to the tuning of a few filtering rules configured in the monitoring server, resulting in fewer false-positive fault events.

The tables in Figure 4 summarize a few months of operational data. The first table categorizes all the events forwarded to AIMS except the events filtered by AIMS



Events Handled by AIMS	% of Events Handled
Performance	39.9%
Disk Utilization	21.9%
Processes Offline	19.4%
Ping Failure	16.0%
Hypervisor Failure	1.4%
Infrastructure	1.0%
OS	0.3%
Events filtered due to AIMS policies	**% of Events Filtered**
Outage	55.8%
Event Irrelevant in Context of State	28.5%
Other	10.0%
CircuitBreaker	3.9%
Side Effect of Automated Action	1.8%

Figure 4: AIMS' Event Handling Statistics

policies, and the second table summarizes events filtered (ignored) due to policies such as outage handling, circuit breaker, state-specific event relevancy checking, etc.

6. Related work

Microsoft's AutoPilot system [8] is an integrated platform that manages infrastructure provisioning, application deployment, system monitoring and repairs. Both AIMS and AutoPilot use FSMs to model fault-handling policies with AutoPilot using a distributed implementation for robustness and AIMS implementing a single-instance persistent and fault tolerant engine. Being a standalone incident management system, AIMS' is suitable for use in any monitor-able system, unlike AutoPilot which works with applications developed using its specialized framework (e.g. Windows Live and Bing but not Azure). No documentation of incident management of Amazon's EC2 or Google's App Engine is available.

Policy-based autonomic management systems have been the subject of research for many years, e.g. see [9]. Our design uses a policy definition mechanism that is different from, and yet can encompass, traditional policy languages, and has focused also on broader system issues such as framework scalability and robustness.

7. Conclusions and Future Work

The paper describes a monitoring and event-based automated incident management system deployed in a PaaS Cloud. In contrast to policy- and rule-based autonomic systems, we describe the use of a FSM-based approach. The FSM abstraction has proven to be very effective for modeling event-based policies and implementing a persistent and fault tolerant execution framework around state management, while providing a foundation for embedding rule languages to control state transitions and actions using more complex logic.

Future work is focused on the following. *Fault tolerance* would enable the system to survive unplanned system outages, addressing a fail-stop fault model. A log replay-based fault tolerant implementation of a single-instance (non-replicated) AIMS, which extends existing persistent storage to log key steps in FSM scheduling and execution, is work in progress. Realtime detection of *event storms*, especially for the category of "outage" events that can enable AIMS to detect that a cell-wide outage is in progress without explicit notification from the system administrator, is another area being explored. Finally, monitored data is being stored in a data warehouse, and we plan to mine it to discover new incident management policies to supplement expert knowledge.

7. Acknowledgements

The authors would like to acknowledge Murthy Devarakonda for seeding the initial ideas for this project, and Mahesh Viswanathan for guiding us in addressing the core set of system requirements.

8. References

[1] Amazon EC2: http://aws.amazon.com/ec2/

[2] Windows Azure:
http://www.microsoft.com/windowsazure/

[3] Google App Engine:
http://code.google.com/appengine

[4] A. Kivity, Y. Kamay, D. Laor, U. Lubin, and A. Liguori. Kvm: the Linux virtual machine monitor. In OLS '07: The 2007 Ottawa Linux Symposium, pp. 225-230.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T, Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In ACM SOSP (2003), pp. 164-177.

[6] VMWare Information Guide:
http://www.vmware.com/files/pdf/software_hardware_tech_x86_virt.pdf

[7] SCXML: http://www.w3.org/TR/scxml/

[8] M. Isard. Autopilot: Automatic Data Center Management, Operating Systems Review 41(2) (2007), pp. 60-67.

[9] M. Devarakonda, D. Chess, I. Whalley, A. Segal, P. Goyal, A. Sachedina, K. Romanufa, E. Lassettre, W. Tetzlaff, B. Arnold. Policy-Based Autonomic Storage Allocation, Self-Managing Distributed Systems (2003), pp. 143-154.