Enforcing User-defined Management Logic in Large Scale Systems

Srinath Perera, Dennis Gannon
Computer Science Department
Indiana University, Bloomington IN 47405
{hperera, gannon}@cs.indiana.edu

Abstract

The ubiquity of information technology, technological advances, and utility computing trends have motivated large-scale systems, but managing and sustaining these systems is far from trivial. Automatic or semi-automatic monitoring and control are a potential solution to this problem. However, since management scenarios differ from system to system, a generic management framework that can manage a wide variety of systems should support user-defined management logic. This paper proposes a novel architecture that can manage large-scale systems according to user-specified management logic that depends on both global and local assertions of the managed system. Furthermore, the paper demonstrates that despite having a global view of the managed system, a management framework can scale to manage most real world usecases.

Keywords: system management, large-scale systems, rules, self-stabilization

1. Introduction

The ubiquity of information technology has widened user bases of existing systems and motivated national scale applications and services (e.g. processing weather data collected across U.S.). Moreover, SOA, Distributed Systems, and Utility computing trends have made large-scale systems increasingly possible. Because of these and many other reasons, large-scale systems are commonplace. However, even though it is possible to build large-scale systems, keeping those systems running has been an arduous task. As illustrated in Ganek et al. [11], in these systems, failures and changes are a norm rather than an exception.

Management frameworks, which monitor systems, approximate their global state, and maintain them in an acceptable state by alerting individual resources as necessary, are a potential solution to this problem. However, management scenarios differ from system to system, but only large organizations can afford to build specific management frameworks for each of their systems. With large-scale systems becoming commonplace, there is a growing need to build a generic management framework that can manage a wide variety of systems. Since management scenarios differ from system to system, such a framework must support management logic authored by users themselves. Management logic are instructions provided by users that describe how to recover the system when certain faulty conditions are detected in the managed system. If these logic assert only a single component of the system, we call them local logic, and if these logic assert multiple components in the system, we call them global logic. For example, the global logic may contain global assertions like "*is a registry service is running in the system?*" or "*is the ratio of hosts to service in the system greater than 10?*" and perform corrective actions. Users need a system to be healthy as a whole, and those requirements, often, translate to global assertions (global logic) about the system. Hence, support for global logic is preferred.

Typically, a management framework consists of one or more managers, but a single manager neither scales to manage a large-scale system nor it is reliable. Therefore, to manage large-scale systems, a management framework needs a group of managers. Usually, such a framework assigns resources among managers, and each manager, consequently, has a partial view of the system. Even though making decisions with a partial view is not impossible, to do so, the decision logic should have been designed in such a way that the expected global properties will emerge from local decisions. However, such a design is challenging even to a researcher, let alone a user; therefore, a global view is preferred in frameworks that support user defined management logic. As we will show in the related work section, this is a problem that has been given a limited attention in literature. This paper attempts to implement a dynamic and robust management framework, which manages a large-scale system by enforcing user-defined management logic that depends on both local and global assertions of the system state.

The remainder of the paper is organized as follows. The following section presents the proposed management

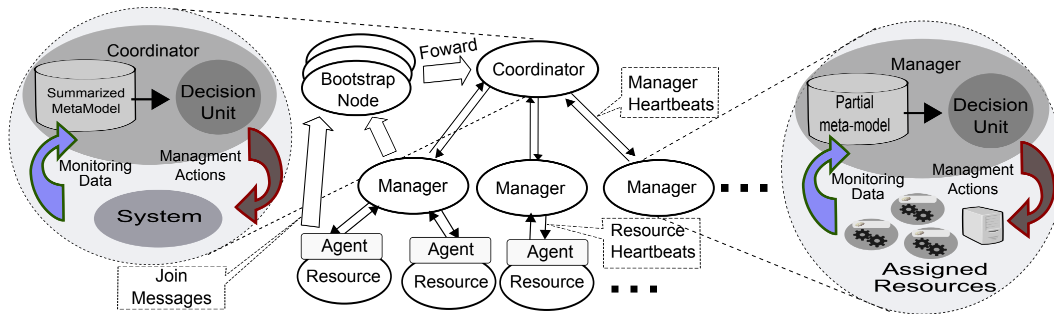

Figure 1. Hasthi Architecture

framework, which we call Hasthi. Section 3 discusses how to use Hasthi to manage systems, and the next section presents a scalability analysis of Hasthi. Section 5 presents related work, and finally, Section 6 concludes the paper.

2. Management Architecture

WSDM specification [4] exposes a representative subset of the target resource state as properties (a.k.a. Resource Properties) and supports control interfaces. To be managed with Hasthi, resources should support both WSDM and the *"join and heartbeat algorithm"* described below. We have developed an agent that can integrate with existing resources (e.g. Axis2 services or hosts) and make them compatible with Hasthi. We call a resource that is integrated with the agent a *"Managed Resource."* Furthermore, given a resource, we call an externally stored snapshot of its resource properties collected at some time as a *"meta-object"* of the resource, a collection of meta-objects as a *"partial meta-model,"* and a collection of meta-objects that has an one-to-one mapping to resources in the managed system as a *"meta-model"* of the system. Typically, a meta-object is an object stored in a remote server (e.g. manager) that contains a snapshot of resource properties of the resource, and Hasthi monitors the corresponding resource and periodically updates each meta-object as the resource changes.

As explained in the introduction, the goal of Hasthi is to enforce both local and global user-defined management logic in large-scale systems. As Figure 1 depicts, Hasthi architecture consists of three parts: a manager-cloud that keeps tracks of and keeps connected managers and resources in the system, a meta-model of the system that exposes monitoring information of the system with delta-consistency (that is all changes are reflected in the meta-model within a constant time), and a decision framework that utilizes the meta-model to enforce user defined management logic.

2.1. Manager-Cloud

The manager-cloud consists of managers, resources, a coordinator, and bootstrap nodes. The basic building block of Hasthi is a service called a "manager," and apart from supporting a service interface, each manager has a control thread that activates periodically, performs bookkeeping, evaluates assigned resources, and performs corrective actions. Furthermore, there is a designated manager called the "coordinator," which oversees other managers. Moreover, bootstrap nodes run on well-known addresses and act as entry points to Hasthi.

Managers, resources, and the coordinator are arranged according to the hierarchy given in Figure 1, where managers send periodic heartbeats to the coordinator and resources send periodic heartbeats to the assigned manager. Managers and the coordinator form a Distributed Hash Table (DHT) based Peer-to-Peer network (P2P). In normal operation, for instance when sending heartbeats, they use HTTP SOAP for communication, but use broadcast and anycast (send a message to a random node) over the P2P network for initialization, recovery, and advertising current coordinators.

When a new manager starts, it requests the coordinator from a bootstrap node (bootstrap nodes find the coordinator by asking other managers via an anycast), joins the manager-cloud if a coordinator is found, or periodically retires and becomes a coordinator itself after a timeout if all retires have failed. Hence, the first manager to join becomes the coordinator. If the coordinator fails, managers will detect the failure when manager heartbeats fail and elect a new coordinator among themselves. Specifically, when a manager detects that the coordinator has failed, it waits for a random amount of time, broadcasts a message requesting nominations for a new coordinator, decides on the best manager from those responses, and invites the manager to become the coordinator, which assumes the role of the coordinator. To ensure the system only has a one coordinator (e.g. which happens due to communication failures or at the startup), each coordinator periodically broadcasts its address, and when a coordinator receives a message from another coordinator, it resigns if the other is better. Therefore, except for one, all coordinators eventually resign. Furthermore, if a resigned coordinator receives a manager heartbeat, it notifies the sender about the new coordinator. Currently Hasthi uses the manager's age to select the best manager, but it is possible to perform the election based on other criteria like memory, network, and process capacity of man-

agers. In addition, the new coordinator may redistribute resources it is managing among other managers. Subsequently, other managers and resources join the new coordinator and rebuild the hierarchy depicted in Figure 1 and the meta-model, and after recovery,the new coordinator starts its control loops.

On the other hand, each resource has an integrated agent, which periodically sends heartbeats to the manager if the resource has a manager assigned or periodically sends a "ManageMe" message to a bootstrap node otherwise. When a bootstrap node receives a ManageMe message, it forwards the message to a coordinator, the coordinator assigns the resource to a manager, and manager notifies the resource. Furthermore, if the assigned manager has failed at any point, resource heartbeats will to fail, and the resource will restart the join process. The coordinator ignores any duplicate ManageMe messages. If a resource fails, the assigned manager detects it by the absence of heartbeats, updates the meta-model, and user defined management logic may perform corrective actions.

In retrospect, the manager-cloud automatically recovers from coordinator, manager, and resource failures, and after recovery, it self-stabilizes to rebuild the hierarchy and the meta-model given in Figure 1.

2.2. Meta-Model

An agent integrated with each resource monitors the resource and sends collected data (resource properties) piggybacked with resource heartbeats. Resource properties consist of configurations (e.g. number of maximum threads, size of a thread pool) and metrics (measurement like memory and CPU usage and number of requests pending), and each resource heartbeat includes current metric values and changes to the configurations since the last heartbeat message. When a resource is assigned a manager, the manager creates a meta-object (an in-memory object) to represent the resource, and whenever it receives a heartbeat from the resource, it updates the meta-object using information included in heartbeats. All meta-objects in managers create a meta-model of the system. If heartbeat messages from a resource are missing, the manager performs failure detection and updates the meta-object if the resource has failed.

To facilitate global decisions, the coordinator keeps a summary of every meta-object locally within its memory. A summary includes a few properties such as, name, management endpoint, and operational status (e.g. Busy, Saturated, and Crashed); thus, a summary does not take excessive memory and is seldom updated. If changes to a meta-object change its summary, the assigned manager sends those changes to the coordinator piggybacked with its heartbeats, and when received, the coordinator updates the corresponding summarized meta-object. This summarization is motivated by the observation that for global level decisions, a high-level summary of resources suffices, and this crucial observation has made the proposed architecture possible. It is true that the summarized data contained in the coordinator places an upper limit on the scale of Hasthi. However, as we shall demonstrate in Section 4, this limit is sufficient to manage most real world systems.

2.3. Decision Framework

Using Drools rule language [1], users can author rules that instruct Hasthi on how the system should be managed. Rules are two types: local rules and global rules. As shown by Figure 1, every manager has a control-loop that activates periodically and evaluates the meta-objects of assigned resources using local rules, and the coordinator has a control-loop that activates periodically and evaluates the summarized meta-model using global rules. These evaluations may trigger actions, and the corresponding manager or the coordinator carries out these actions. Since each manager manages different resources, manger loops are independent. To resolve conflicts between different rules, Drools supports conflict resolution via priority. Hasthi evaluates rules using a Rete algorithm based rule engine, which remembers results from old evaluations and works incrementally. For example, if one resource in the system changes, the rule engine only needs to evaluate that resource. This improves the rule evaluation performance dramatically.

3. Using Hasthi to Manage a System

Let us discuss an example to illustrate how Hasthi can be used to mange systems. Consider a set of services that are registered with a Registry where clients query the registry, find services, and execute them. Assume that they are managed using Hasthi. While managing the system, Hasthi will assign each resource to a manager, monitor them, and maintain a meta-model that represents the monitoring state of the system. By evaluating rules using the meta-model, which approximates the system state, Hasthi effectively evaluates the system.

```
rule CreateAlternativeForRegistry
when
  not exists( ManagedService(state = UpState,
    type = Registry));
  registry: ManagedService(state = CrashState,
    type = Registry)
then
  system.execute(new CreateServiceAction(registry));
end
```

As shown by the above rule, each rule consists of a when-clause (a condition) that uses an object query language to select meta-objects from the meta-model, and for each matching set of meta-objects, Hasthi triggers the then-clause (corrective actions). Drools manual [1] describes rules in detail. If a service has failed in the managed system, the heartbeat messages will be missing, and Hasthi will trigger a failure detector and mark the service as "Crashed" based on the outcome. The default failure detector pings

the service, but users can introduce custom failure detectors. Among rules, the above rule, which creates a new registry if the registry has failed, ensures that the system has at least one registry. Similarly, two other possible rules in this usecase are a rule that detects if services in the system has failed and restarts them and a rule that removes overloaded services from the registry, creates new instances in their place, and adds overloaded services back to the registry when they have recovered.

Hasthi supports creating new, shutting down, restarting, and migrating resources and getting or setting resource property values as management actions. Hasthi defines a resource profile that describes deployment information about each resource type in the system. For example, when triggered, the create-service-action reads the service startup command, which is provided as a shell script, from the profile and executes it in the target host using a host agent running in the host. Furthermore, in addition to the above actions, users can define their own actions and use them within the then-clause. Hasthi supports a wide range of management scenarios, and the above scenario is only an example. For instance, Perera et al. [21] present a detailed application of Hasthi to manage a large-scale e-science cyberinfrastructure.

Management scenarios could go wrong in many ways and have hidden complexities, and following are solutions to some of them. For instance, if a management action failed, rules will repeatedly retry the action, starting a never-ending recovery loop. To mitigate this, Hasthi marks the target resource as "Unrecoverable" if any of the management action on a resource failed, thus breaking the loop, and asks for human help. Furthermore, if a host has failed, Hasthi has to restart services running in the old host in a new host; consequently, the addresses of those services will change, and clients will fail because service locations have changed. To mitigate this problem, using the global view of the system, Hasthi provides a "dependency-discovery" operation, which enables services to search and find the locations of other services. For example, if clients in the above example detect that the current registry has failed, they can find the new registry location using this dependency-discovery operation. Moreover, Hasthi does not recover state when it recovers failed services, but if a service exposes its storage location (e.g. database) as one of its resource properties, Hasthi passes that location as an argument when it recovers that service from a failure. Hasthi neither writes state to the storage nor recovers it, but any service may write state to the storage in the normal operation and recover that state when it has failed and is restarted by Hasthi.

4. Scalability Analysis

This section discusses how Hasthi scales (by increasing number of managers) with an increasing number of resources. While managing a system, the coordinator receives heartbeat messages from managers, each manager receives heartbeat messages from assigned resources, and both have control-loops that perform bookkeeping and rule evaluations. We use the term "heartbeat latency" to denote the latency from the time heartbeat is created until the time a response is received for the heartbeat, and we use the term "control-loop overhead" to denote the latency from the start to the end of a control-loop execution. We measured the load on Hasthi using four metrics: (1) Resource-heartbeat latency (for heartbeats from resources to managers), (2) Manager-control-loop overhead, (3) Manager-heartbeat latency (for heartbeats from managers to the coordinator), (4) Coordinator-control-loop overhead. To understand how Hasthi scales, we changed the number of managers and resources and measured the above metrics (dependent variables).

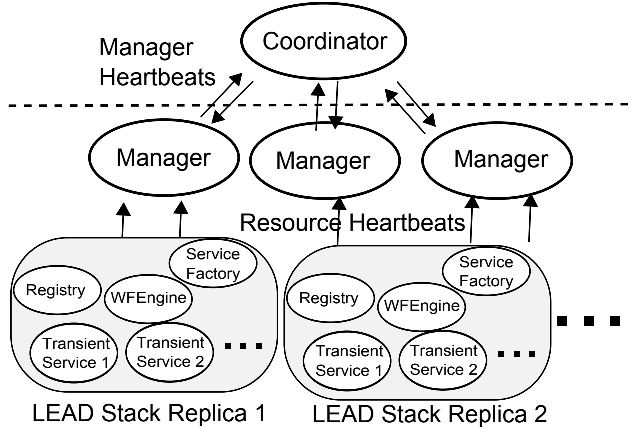

Figure 2. Test Setup

4.1. Test System and Workload

Our test setup models a large-scale deployment of an e-science project called LEAD [3]. As shown by Figure 2, the test system consists of replica units, each of which contains a complete LEAD stack that consists of a workflow engine, a registry, a service factory, and transient services created on demand. Services are integrated with the Hasthi agent, so they join Hasthi and expose their monitoring state as resource properties (e.g. Operational Status, Last Request Time, and Number of Pending Requests). Each service (e.g. workflow engine, registry) implementation periodically updates the resource properties (monitoring data) using a randomized algorithm (see [2]), which simulates the service receiving requests, their outcome (e.g. their success or failure), and the time taken to process requests etc. Furthermore, each service fails with a probability 0.01 per hour. Hasthi monitors services and changes to these properties, makes decisions, and performs corrective actions to keep the system within acceptable bounds.

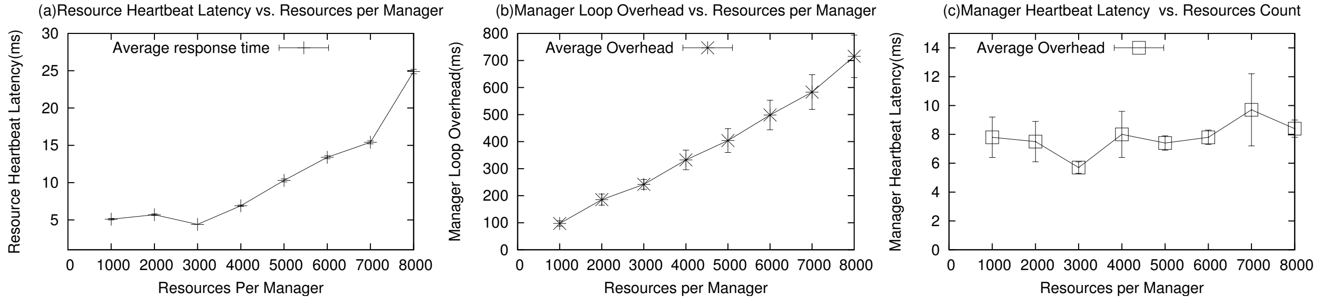

Figure 3. Limits of a Manager

We have performed an experiment by running different service sets in a host each per hour and measuring I/O and CPU overheads of the host in order to understand how many services can be placed in one host. CPU overhead is measured with "load average," which is the standard measure of load in a UNIX system, and since a host has two CPUs, a 2.0 load average represents complete utilization. Even with 200 services, the host transferred 0.04 MB/s out of possible 1Gb/s bandwidth ($< 1\%$) and had 0.02 load average out of 2.0 ($< 2\%$). Hence, in the following experiments, we placed 200 services in each host. All of the following tests were conducted using a cluster of 128 nodes, each having a Dual AMD 2.0 GHz processor, 4GB Memory, 1GB Ethernet, Red Hat Linux, and Sun Java 1.5. In all tests, unless otherwise specified, each manager was given a host exclusively.

We placed one copy of the LEAD stack (200 services) in one host and changed the size of the test system by changing the number of replica units. We deployed Hasthi to manage test systems with 30 seconds as the time period between all heartbeats and control-loop evaluations. We defined the following management scenarios for managing the test system. (1) If a persistent service fails, create a new service to replace it, (2) If the number of transient services of a particular type is low, create new instances to compensate, (3) If a transient service has overloaded, remove it from the service registry and later add it back when it has recovered, (4) Shut down old transient services. (5) After processing 10 requests, if a service has generated more faulty responses than successful ones, decide it is faulty and shutdown the service. We implemented scenarios 1, 2 and 3 as global rules and 4 and 5 as local rules, and Hasthi managed the test-system and performed corrective actions using these rules. Rules and details about the workload can be found from the Hasthi Web site [2].

In the following discussion, we use the term "*MXN test run,*" to denote a test run where N resources are managed with M+1 managers (where one manager will become the coordinator). For each test run, we let Hasthi to manage the system for one hour and measured the aforementioned four metrics throughout the test while Hasthi monitored the system and performed corrective actions. Each graph shows data with 95% confidence interval.

4.2. Limit of a Manager

To find the maximum number of resources one manager can handle, 1XR test runs were performed with R having values 1000, 2000, ..., 8000 resources. Hasthi had one manager (with 256MB of memory) and a coordinator. As depicted by Figure 3, a manager can manage 5000 to 8000 resources, and the resource-heartbeat latency and the manager-loop overhead both exhibit a linear trend, which undergoes a marginal rise at 8000 resources. This linear trend can be attributed to the increase of heartbeat messages and the increased overhead of evaluating rules with more resources. Furthermore, the linear trend suggests that the overhead is not prohibitive and that Hasthi can keep up.

4.3. Manager Load Behavior

To find how Hasthi behaves with load, MXR test runs were performed, where managers $M = \{5, 10, 20\}$ and resources $R = \{5000, 10000, 15000\}$. With more managers, Hasthi scaled past 8000 resources up to 15000 resources, which was the largest test system we tested in this experiment. Similar to the first experiment, we saw that the coordinator-loop overhead increases linearly and the manager-heartbeat latency stays stable (see [2]).

We can scale Hasthi by adding more managers if the load on a manager only depends on resources assigned to itself and is not affected by the existence of other managers or resources. We verified this condition using the data available from Experiments 1 and 2. Figure 4 shows the resource-heartbeat latency, the manager-loop overhead, and the manager-heartbeat latency plotted as a scatter plot against the number of resources assigned to a manager, which shows correlations between overheads and resources per manager. We can see from the figure that data points for the same X values are reasonably close to each other, which verifies our condition (Data verify it up to 2000 resources per manager). Therefore, we can scale up Hasthi by adding more managers.

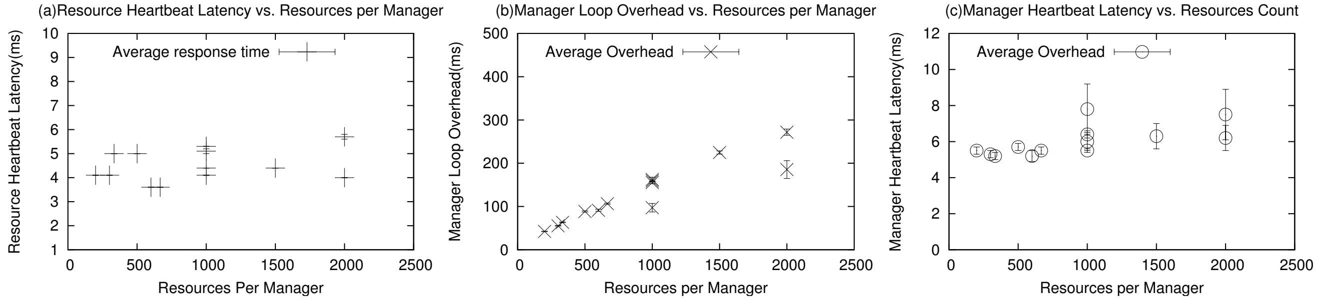

Figure 4. Correlation between overhead & resources per manager

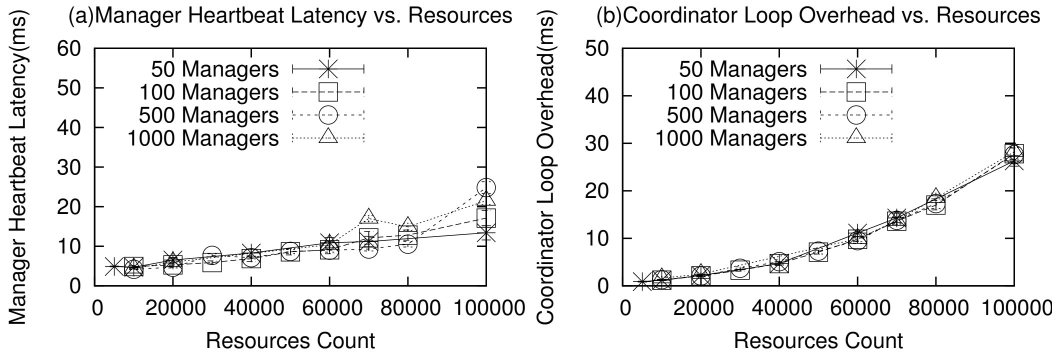

Figure 5. Coordinator Limits

4.4. Limits of a Coordinator

As observed in previous experiments, Hasthi scales up with more managers; therefore, the limit of the coordinator will decide the scalability limit of Hasthi. However, with 200 services per node, a 128-node cluster does not allow us to test Hasthi with very large-scale systems. However, if we can mimic all messages and behaviors seen by the coordinator while managing resources, then we can test the coordinator to its limits without having to run tens of thousands of resources. We have developed a Test-Manager to do this, and it is described below.

By simulating managed resources using a randomized algorithm, each Test-Manager makes the coordinator believe that it is managing a group of resources. For each experiment, the coordinator is set up with Test-Managers, each Test-Manager is given the number of resources it should be emulating as an argument at the startup, and the following algorithm mimics all messages exchanged between the coordinator and a conventional manager. Each Test-Manager joins the manager-cloud like any other manager at the startup and periodically sends heartbeat messages to the coordinator. In addition, for each resource to be emulated, it sends a ManageMe message to the coordinator, and the coordinator assigns the resource described in the message to a manager in the manager-cloud, which is also a Test-Manager in this case. The assigned Test-Manager creates an in-memory object that simulates properties and failures of the resource using the same algorithm used by services in Experiment 1 (see [2]). Furthermore, the Test-Manager sends all major updates that happen to the simulated resources piggybacked with its heartbeats to the coordinator. Therefore, the coordinator perceives resources as real, not as simulated ones. Moreover, the management endpoint of each resource is also mapped to the assigned Test-Manager, and when the coordinator performs an action, it is also emulated by the Test-Manager. Consequently, with Test-Managers, all messages, their order, and timing behave as if real resources exist behind Test-Managers. Hence, we argue that experiments done using the Test-Manager-based workload are representative of a real workload the coordinator has to handle when Hasthi manages a system.

For each test run, one real manager was used as the coordinator (1024MB as the Java maximum heap size) and other managers were Test-Managers (distributed across 10 hosts). To find the limits of the coordinator, MXR test runs were performed with $M = \{10, 20, 50, 100, 500, 1000\}$ being Test-Managers and $R = \{20k, 30k, 40k, ...100k\}$ being resources emulated by Test-Managers.

As illustrated by Figure 5, both the manager-heartbeat latency and the coordinator-loop overhead behave linearly with minor disturbances toward the end, and this increase can be attributed to the fact that with more resources, more information need to be transferred, processed via heartbeats, and evaluated from the coordinator-loop. Most lines

are clustered together, indicating that the number of managers makes a minimal difference to the system, which also suggests that the system is limited by the coordinator. Furthermore, with 1000 managers and 100,000 resources, the maximum coordinator-loop overhead was less than 1% of the 30 seconds period between two rule evaluations of Hasthi, and the maximum heartbeat-latency was less than 10% of the 30 seconds period between two heartbeats (not shown in the figure, which shows averages). This suggests that processing finishes within each period without affecting the next period, which inturn suggests that Hasthi is within its operational range. Therefore, we argue that the coordinator scales to manage 100,000 resources and up to 1000 managers.

Let us synthesize these results. We observed that one manager could scale to manage 5000-8000 resources and the coordinator scales to manage 1000 managers and 100,000 resources. Furthermore, we observed that until 2000 resources per manager, the load on a manager is independent of other managers and resources in the system but depends on the number of resources assigned to the manager. First and third observations suggest that we can scale up Hasthi by adding more managers and distributing resources among them, and the second observation suggests that this can be done until 100,000 resources (which is the limit of the coordinator). Therefore, these observations provide strong evidence that Hasthi can scale to manage 100,000 resources.

5. Related Works

Management systems are found under network management (e.g. Philippe et al. [17]), system management, Resource Management (e.g. Bosin [6]), autonomic systems (e.g. huebscher et al. [19]), and monitoring systems (e.g. Zanikolas et al. [61). However, only few systems have addressed global control across multiple managers.

Wildcat [14] is based on an agent framework, and agents are grouped together to provide a management hierarchy where top-level agents (managers) control the next levels by modifying policies. However, the system suffers from a single point of failure at the top of the hierarchy. Our solution differs by employing an election-based model for robustness, maintaining a meta-model, and using rules instead of policies, which provide more explicit control than policies.

In decentralized management systems (e.g. DMonA [20]), each node monitors and controls itself as well as its neighborhood and global control emerges from local decisions. However, as we pointed out in the introduction, authoring management logic that ensures emergent behavior is beyond most users.

Gadgil et al. [10] provide a management hierarchy where the topmost layer is replicated to guard against failures, and the management is performed by user-defined code that provides fault tolerance and check pointing among management functions. However, the system assumes that a scalable and reliable registry exists, whereas our solution does not depend on a single entity and supports user defined global management via rules.

Georgiadis et al. [13] present a self-organizing architecture where managers coordinate using total ordered multicast. Therefore, every manager has the same global view of the system and is able to make global decisions. However, because group communication is used, the scalability of the system is limited.

Among other systems, Marvel-1995 [16] and Rainbow [12] have one manager and, therefore, avoid the need for global control among managers. However, they do not scale to manage large-scale systems. Furthermore, Extreme [15], Tivoli Management Suite [7], Adams et al. [8], and ReactiveSys [18] use groups of independent managers, and some of them depend on manual coordination (e.g. Tivoli [7]). However, none of these systems address the global control across managers. Moreover, DREAM [9] and Hifi [5] are built on top of distributed publish/subscribe broker hierarchies, and they manage a system by triggering management actions using complex event processing. Although these systems are scalable, the event action model has limited memory; therefore, providing coordination across decisions in this model is non-trivial. None of the above systems discuss global control.

Among decision models, Infospect [22] and Sophia [23] evaluate the system state using Prolog like logical expressions for monitoring and diagnostic purposes. Rainbow [12], Marvel-1995 [16], and ReactiveSys-1993 [18] use IF/THEN rules to perform reactive actions. Hasthi differs by its distributed nature, scalability, and the fact that rules have a complete view of a large-scale system.

6. Discussion and Conclusion

Hasthi consists of a cloud of managers and a coordinator elected among them to oversee them. The coordinator assigns each resource to a manager, and each manager creates a meta-object—a snapshot of resource state—for each assigned resource and updates it periodically as the resource changes. Furthermore, the coordinator keeps a summary of each meta-object in-memory and keeps it up to date. Users provide management logic as rules, where managers periodically evaluate the meta-objects of assigned resources using local rules and the coordinator periodically evaluates the summaries of meta-objects using global rules. Thus, Hasthi maintains a global view of the system enabling users to incorporate global assertions into their management rules and, therefore, easily supports most management usecases.

Furthermore, in the scalability analysis, we illustrated that Hasthi scales to 100,000 resources, and this is one of the key result of this paper. For the control of managers and the

global control, Hasthi depends on the coordinator, which is one node that has limited resources. Let us analyze the coordinator for bottlenecks and try to identify any characteristics of the architecture that made these results possible. The first bottleneck is that Hasthi has to track the state of all resources in the system and the size of all states could be prohibitive. As discussed in Section 2, to mitigate this, the coordinator keeps only a summary of each resource locally in the memory. This summary is small and is updated only when the resource behavior has significantly changed (e.g. Crashed, Saturated) and, therefore, reduces the space (memory) requirements need to track resources. The second bottleneck is that the coordinator should receive heartbeat messages and keep the summarized meta-model of the system up to date by applying changes included in the heartbeat messages, and this process is expensive. Hasthi mitigates this by only propagating changes that happen to resource summaries, and since resource summaries change slowly, this approach reduces the overhead. The third bottleneck is that the coordinator has to periodically analyze the summarized meta-model using rules, and the cost of this evaluation could be prohibitive. Hasthi mitigates this problem by using the Rete algorithm for evaluating management rules, which provides a tradeoff between space and time (processing overhead) by remembering the evaluated results. Hence, each evaluation only needs to evaluate new facts or changed facts. In this setting, as explained with previous bottlenecks, information about a resource stored within the coordinator change only when the summary of the resource changes, which happens only when the resource undergoes a major change. Therefore, at each step, the coordinator only receives few changes, and the Rete algorithm only has to evaluate these changes, which is manageable. We believe this explains the scalability results.

Using a Meta-model has been proposed before by Marvel [16]; however, we have extended it by building a distributed meta-model, summarizing it to fit inside the coordinator, and implementing rule based global and local control-loops using the meta-model. Therefore, the main contribution of this paper is using a distributed meta-model and summarization to implement a robust control-loop that can make decisions based on global assertions. Furthermore, a specific contribution is demonstrating that despite having a global view of the managed system, a management framework can scale to manage about 100,000 resources, which is sufficient for most real world usecases.

References

[1] Drools. online. http://labs.jboss.com/drools/.

[2] Hasthi performance. http://extreme.indiana.edu/hasthi/analysis.

[3] Lead project. online. http://leadproject.org/.

[4] Oasis web services distributed management. online, August 2006. www.oasis-open.org/committees/wsdm/.

[5] E. Al-Shaer, H. Abdel-Wahab, and K. Maly. Hifi: A new monitoring architecture for distributed systems management. In *International Conference on Distributed Computing Systems*, 1999.

[6] B. Andrea et al. Cooperative e-organizations for distributed bioinformatics experiments. In *9th International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 2008.

[7] J. B. Baker, D. Reimer, S. Spiro, and J. Whitfield. Management of service-oriented architecture ibm tivoli soa management suite, June 2005.

[8] R. P. Brettf, S. Iyer, et al. Scalable management. In *International Conference on Autonomic Computing*, pages 159–170, 2005.

[9] A. Buchmann et al. Dream: Distributed Reliable Event-based Application Management. *Web Dynamics*, pages 319–352, 2003.

[10] H. Gadgil, G. Fox, et al. Scalable, fault-tolerant management of grid services. In *IEEE Cluster 2007*.

[11] A. Ganek and T. Corbi. The drawing of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[12] D. Garlan, S.-W. Cheng, et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[13] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. *first workshop on Self-healing systems*, pages 33–38, 2002.

[14] M. Jarrett and R. Seviora. Constructing an autonomic computing infrastructure using cougaar. In *International Workshop on Engineering of Autonomic & Autonomous Systems*, pages 119–128, 2006.

[15] G. Kaiser, J. Parekh, et al. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *AMS'03:International Workshop on Active Middleware Services*, page 22, 2003.

[16] T. Koch, B. Kramer, and G. Rohde. On a rule based management architecture. In *Workshop on Services in Distributed and Networked Environments*, page 68, 1995.

[17] J.-P. Martin-Flatin, S. Znaty, and J.-P. Hubaux. A survey of distributed enterprise network andsystems management paradigms. *J. Netw. Syst. Manage.*, 7(1), 1999.

[18] K. Marzullo and M. D. Wood. Tools for constructing distributed reactive systems. Technical Report TR 91-1193, Ithaca, New York (USA), 1991.

[19] J. McCann and M. Huebscher. A survey of Autonomic Computing - degrees, models and applications. December 2007.

[20] S. Michiels, N. Janssens, W. Joosen, and P. Verbaeten. Decentralized cooperative management: a bottom-up approach. In *IADIS AC*, pages 401–408, 2005.

[21] S. Perera, S. Marru, D. Gannon, and B. Plale. Application of Management Frameworks to Manage Workflow-based Systems. 2009. Submitted to 4th IEEE International Conference on Web Services (ICWS), http://extreme.indiana.edu/hasthi/hasthi-lead.pdf.

[22] T. Roscoe, R. Mortier, et al. Infospect: using a logic language for system health monitoring in distributed systems. In *10th ACM SIGOPS workshop*, pages 31–37, 2002.

[23] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an information plane for networked systems. *SIGCOMM Comput. Commun. Rev.*, 34(1):15–20, 2004.