Application of Management Frameworks to Manage Workflow-based Systems: A Case Study on a Large Scale E-Science Project

Srinath Perera, Suresh Marru, Thilina Gunarathne, Dennis Gannon, Beth Plale
School of Informatics, Indiana University, Bloomington
{hperera, smarru, tgunarat, gannon, plale}@cs.indiana.edu

Abstract

Management architectures are well discussed in the literature, but their application in real life settings has not been as well covered. Automatic management of a system involves many more complexities than closing the control-loop by reacting to sensor data and executing corrective actions. In this paper, we discuss those complexities and propose solutions to those problems on top of Hasthi management framework, where Hasthi is a robust, scalable, and distributed management framework that enables users to manage a system by enforcing management logic authored by users themselves. Furthermore, we present in detail a real life case study, which uses Hasthi to manage a large, SOA based, E-Science Cyberinfrastructure.

1. Introduction

Although Web services and SOA have had much success with small and medium size systems, their adoption with large-scale systems yields architectures that consist of many services running from different machines where a failure of any service often causes the system to fail. Consequently, these systems may often fail, and maintaining such a system, therefore, requires administrators to monitor these services round the clock. The administrative cost of complex real-life systems is high, and both the Autonomic Computing initiative [12] and the Recovery Oriented Computing Initiative [9] have cited much evidence to demonstrate this observation. In this setting, automating system management is an attractive and viable solution to this problem. Not only it is cost effective, but it can also increase the system availability significantly. Among such frameworks, Hasthi [18] is a robust, scalable, and distributed management framework, which enables users to manage a system by enforcing management logic authored by the users(system developers) themselves.

Although management architectures are well discussed in literature, their practical applications are not. However, automatic management of a system involves much more complexities than just closing the control-loop by implementing a system that reacts to the sensor data and executes actions. Among examples of complexities involved are formulating management scenarios, handling the lost state in failed managed services, avoiding loops if a management action has failed, building a generic framework for actions and monitoring agents, and notifying other services if a service location has changed after recovery. Consequently, the application of a management framework is a topic that warrants detailed analysis, yet has been seldom explored. Furthermore, we believe that at least some of these problems have generic solutions and implications of these solutions are far-reaching and general.

In this paper, we try to solve some of these problems with Hasthi, while using an E-Science Cyberinfrastructure as a case study. Main contributions of this paper are discussing complexities of applying a system management framework to manage systems, proposing solutions or making recommendations regarding each, and presenting a detailed use-case, which describes application of a management framework to manage an E-Science Cyberinfrastructure.

The next section discusses the related work on the application of management frameworks, and the following section illustrates Hasthi. Sections 4 describes integration of a management framework with a given system and associated complexities, and Section 5 describes potential solutions. Section 6 illustrates managing an E-Science Cyberinfrastructure as a case study and the following section presents an evaluation of the case study. Finally, Section 8 concludes the paper.

2. Related Work

There are a number of management frameworks found in the system management literature (e.g. [15, 11, 6, 22, 13, 17]), and Perera et al. [18] present a detailed comparison between the Hasthi framework and these existing management frameworks. In comparison to those systems, the primary advantage Hasthi offers is the ability to run a user-defined global control-loop to manage a large-scale system in which resources are managed by multiple managers. However,

since the paper focuses on applications of Hasthi, not Hasthi itself, we will not spell out details of these comparisons.

Only few earlier works discuss application of system management; among them, Valetto et al. [20] present a case study that manages few Internet Messaging servers using KX management framework and Koch et al. [16] present a general discussion on applying the Marvel rule-based system management framework [17] in order to recover from failures reactively. In contrast, we present a detailed discussion on managing a much more complex system, which could be useful for managing other web services based workflow and E-Science systems. Furthermore, we would like to note that initial results of this work were presented as a poster [19] in the E-Science conference, 2008.

3. Background (Hasthi Framework)

Hasthi is a scalable and reliable management framework, which monitors and manages a large-scale system according to user-defined rules. Hasthi can manage resources defined by the WSDM specification [4], where they are called "manageable resources" or "managed resources" and the system being managed is called a "managed system". Each managed resource that supports the WSDM specification exposes a representative subset of its state as properties, and usually resource developers define this subset; therefore, Hasthi is flexible in terms of which properties each resource exposes. When a resource joined Hasthi, it is assigned to a manager, monitored, and controlled by Hasthi.

Hasthi consists of a dynamic and robust manager-cloud consisting of managers, an elected coordinator, and a set of bootstrap nodes where managers manage resources and the coordinator oversees the managers. Each managed resource joins the manager-cloud via bootstrap nodes running in pre-advertised endpoints, and bootstrap nodes forward the join message to the coordinator, which in turn assigns the resource to a manager. After assigned to a manager, each resource periodically sends heartbeat messages to the assigned manager, and similarly, each manager periodically sends heartbeat messages to the coordinator. Therefore, failures of both resources and managers can be detected by the absence of heartbeats. Hasthi recovers from manager failures (by re-assigning resources), resource failure, and the coordinator (electing a new one among managers) failures and keeps active components of the system connected.

Let us look at the dissemination of monitoring information and the decision framework of Hasthi. We call an externally (remotely) stored snapshot of resource properties as a "meta-object" of the resource. The resource properties are categorized as configurations and metrics where the former represents resource state and the latter includes readings like memory usage and number of pending requests. Furthermore, after being assigned to a manager, each resource periodically sends heartbeats that include collected metrics and configuration changes collected since the last heartbeat to the manager, and the assigned manager creates a meta-object for the resource and updates the meta-object whenever it receives a heartbeat. Furthermore, the coordinator maintains summaries of individual meta-objects located at various managers, and these summaries are updated through manager heartbeats. Consequently, each manager maintains a meta-object for each resource, the coordinator maintains a summarized version of each meta-object, and Hasthi keeps both types of meta-objects up-to-date. Therefore, the meta-objects reflect the current state of the system.

Furthermore, each manager contains a control-loop that evaluates user-defined management rules using the meta-objects of the assigned resources, and similarly, the coordinator contains a global control-loop, which evaluates user-defined global management rules using the summarized meta-objects maintained in the coordinator. Subsequently, the rule evaluations may trigger management actions in response to failures in the system, which are carried out by the associated coordinator or the manager.

Perara et al. [18] present a comprehensive design and analysis of the Hasthi framework.

4. Integration with a Management Framework

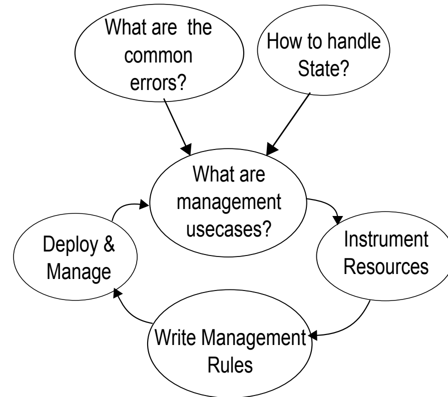

Figure 1. Methodology of Integration

Figure 1 depicts the methodology we developed to guide users in integrating Hasthi, or other such management frameworks, into a large-scale distributed system. The basis of the methodology is an observation by Adams [5] that most error occurrences are caused by a few of the error types (a.k.a. Petro principle). Therefore, by recovering from the most common error types, we can recover from most error occurrences in the system. We confirmed Adams's observation by analyzing LEAD error data over an 18 months period where 30/80 (37%) different error types were responsible for 95% of all error occurrences.

To integrate Hasthi with a given system, users should first

find the most common error types using error statistics, identify the management scenarios that handle these common errors, and integrate Hasthi agents with resources and expose resource properties that are required to implement those management scenarios. Then they should author rules to capture those management scenarios, and use those rules to manage the system with Hasthi. Furthermore, the scenarios and the resulting rules can be improved iteratively.

If users follow the above process and deploy Hasthi to manage the system, Hasthi would react to failures and carry out corrective actions as per user defined management rules. This process, however, has inherent complexities, and furthermore, even though corrective actions recover the system, Hasthi does not provide any guarantee about the behavior of the system while recovery. Therefore, in recovery, following need to be handled.

1. Handling Lost State - If services have failed and recovered, they may lose state.
2. Handling Lost and Failed Requests - While Hasthi recovering the system after a failure, the system will be in an unsafe state. Hence, the system may lose some messages and some requests or sessions may fail.
3. Handling lost system structure - If a host has failed, Hasthi has to move services running in the host to a different host, hence their addresses will change. Consequently, in the system, links to that service from other services are no longer valid, thus breaking the system structure.

Consequently, after recovery, the system may be inconsistent or have incurred failures. Above three cases are general in the sense that to integrate most management frameworks with a system, users have to address above three cases. The next section discusses details about Hasthi agents that instrument resources, Hasthi management actions, lifecycle of managed resources, and above three cases.

5. Managing Systems using a Management Framework

As described in the former section, users should instrument resources to monitor them (by integrating Hasthi agents) and supporting management actions, and then they shall write rules to express their management usecases. Following subsections describe how Hasthi handles Hasthi Agents, management actions, and other complexities associated with the integration.

5.1. Hasthi Agents

As the interface between itself and managed resources, Hasthi uses the WSDM specification with an additional extension to generate periodic heartbeat messages. To facilitate exposing resources as managed resources described in the WSDM specification, Hasthi provides agents, which can be integrated with existing services or hosts, and once integrated, these services or hosts can be managed and monitored using Hasthi. In fact, Hasthi includes several agents, targeted for hosts, web services, and UNIX processes to name a few. This section, however, describes the agent developed for Axis2 based services, as we believe this is the agent most relevant for our audience.

Users can integrate and remove the Axis2 agent, which has been developed as an Axis2 module, purely by changing the configurations of existing services without any changes to the service implementation. For example, instructions for integrating this module can be found in [3]. Axis2 modules [1] are a part of its extension mechanisms, and by supporting the Chain of Responsibility Pattern [21], these modules enable users to inject custom interceptors (a.k.a. Axis2 Handlers) into the axis2 message processing pipeline. To implement the Hasthi module, we have developed a Hasthi Handler, which intercepts every message coming into or going out of the axis2 container. The Handler has two functions. The first, it intercepts all management messages and redirects them to the WSDM implementation of Hasthi. The second, by intercepting other messages, it collects statistics about the service such as number of successful requests, failed requests, and pending requests and exposes them as WSDM resource properties while introducing a minimal overhead. Once the module is integrated, it can be monitored and managed using Hasthi.

5.2. Management actions

The Hasthi agent supports management actions like configuring resources and shutting down resources. In addition, to support other actions (e.g. starting and relocating resources), each type of resource has a resource profile (e.g. following listing) defined via the Hasthi configuration.

A Resource Profile

```
<SystemProfile>
  <resource name="Xregistry">
  <deployment>
    <installDir>/usr/local/xregistry</installDir>
    <startupCommand>xreg-start.sh</startupCommand>
    <shutDownCommand>xreg-stop.sh</shutDownCommand>
    <hostName>silktree.cs.indiana.edu</hostName>
    <hostName>tyr16.cs.indiana.edu</hostName>
  </deployment>
  <behavior><dependency>mysql</dependency></behavior>
  </resource>
  ...
</SystemProfile>
```

As shown by the above listing, the profile describes the service deployment and behaviors. To start and stop services, Hasthi supports tomcat based service installations by default, but with non-tomcat based services, users have to implement them as shell scripts. In the profile, the parameter “hostName” defines hosts where the given resource is installed, and when required, Hasthi creates a new instance of the service in one of these hosts. Furthermore,

to monitor hosts and to enable Hasthi to perform management actions like start/stop services by executing required shell commands on the hosts, each host in the system runs a Hasthi host agent. Moreover, each service may define dependencies on other services, where dependencies say to Hasthi that to start the given service, at least one instance of each dependency must be running. When Hasthi initializes the system, it uses dependencies to decide the bootstrap order of services, and while initializing the system, it waits for some time to discover all services in the system and then creates any missing services based on the system profile and service dependencies.

In management logic (rules), often there are decisions that are best taken by humans (e.g. recover from management action failure). Therefore, to incorporate users into the decision loop, Hasthi introduces User Interactions as a management action. Management logic can trigger this action providing inputs, and the action sends an email to the user. The email asks the user questions or provides instructions, and the user may respond to Hasthi by filling out and submitting a HTML form in the email. After the user submits the form, Hasthi receives it as a REST web service call, returns the user inputs to management logic, and resumes the execution. We shall use this action in many places.

Using the Drools rule language [2], users can define management logic that decide how Hasthi will react to changes in the system. Furthermore, the control-loops at managers and the coordinator periodically evaluate these rules, and the rules trigger management actions in response to error conditions. We will revisit rules in Section 6.

5.3. Resource Life Cycle

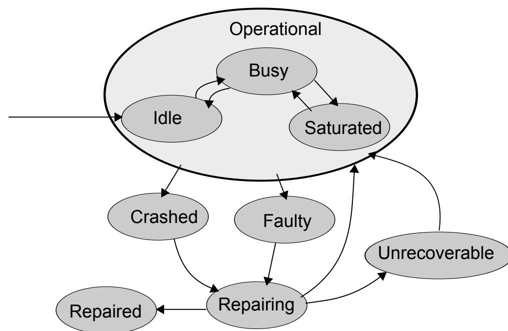

Figure 2. Life Cycle of a Managed Resource

Figure 2 depicts the life-cycle of a managed resource. Among states, the three operational states "Idle," "Busy," and "Saturated" denote that the service is healthy, and management agents decide between these three states based on the number of pending requests—the requests that are received but not yet completed—at a given point of time. If two heartbeats from a managed service are missing, Hasthi triggers a failure detector and marks the service as "Crashed" based on the outcome. Currently, the default failure detector simply pings services for failure detection. In addition to this process, management rules can decide a service is faulty based on conditions such as the ratio between successful and failed requests. Subsequently, when a resource is in a crashed or faulty state, rules perform corrective actions on the resource. Before the action is carried out, the resource is marked as "Repairing," and it is marked as "Repaired" when the action has been completed. However, if the management action has failed, the resource is marked as "Unrecoverable". When this happened in the LEAD system (described in Section 6), the rule performs a user interaction by asking a human user to fix the error and respond by clicking a link in the email. By changing the resource state to "Unrecoverable," "Repairing," and "Repaired" states while a resource is been evaluated and acted upon by rules, Hasthi guards against the possibility of indefinite loops of recovery. For example, if a management action has failed, the resource state is set to "Unrecoverable," and since rules are written to respond to "Crashed" or "Failed" resources, they do not perform actions on the "Unrecoverable" resource. Hence, loops do not occur.

5.4. Handling Complexities

As described in the earlier section, many complexities arise in Hasthi-LEAD integration, and let us discuss potential solutions to some of them.

Handling Lost State: If services in a managed system failed and Hasthi recovered it, the failed services may lose its state. Hasthi does not recover state directly; however, if the service has a storage location (e.g. file, database location) and it exposes that storage location as a resource property, Hasthi provides that storage location to the new service if the service has failed and recovered by Hasthi. The service can use that storage location to save state while in the normal execution and recover that state in the case of a failure.

Handling Lost and Failed Requests: While Hasthi recovers the system, it may be in an unsafe state. Hence, requests or sessions (e.g. workflow) may be lost or fail. We recommend that after the system is recovered, user-defined management logic should resend or resume failed requests or sessions. For example, in the case study, we rerun the failed workflows if the system has failed and recovered.

Handling Lost system structure: If a host has failed, Hasthi has to move services running in the host to a different host. Hence, their addresses will change, but other services do not aware of this change and requests may fail if they try to send messages to the old address. To mitigate this problem, utilizing the summarized meta-model of the system, Hasthi supports a service-discovery operation, which accepts a service type—the port type name of the service

WSDL—and returns the endpoints of all service instances that support the same abstract service description (the same WSDL port type). Using this, services can discover other services in the system both at the start and it is when looking for an alternative endpoint because a dependent service has failed. In our usecase, LEAD portal uses this method to find active services before invoking the workflow.

In addition, there are few practical difficulties in implementing management usecases. Let us look at few of them.

Handling Failed Management Actions: When recovering from a failure, management actions themselves may fail. By using the service lifecycle described in the above section, Hasthi avoids potential unending loops due to failures. Specifically, if a failure happens, Hasthi marks the resource's operational status as "Unrecoverable" and, furthermore, sends an email to the user requesting him to fix the error manually and respond by clicking a link in the email.

Fail Positives: Hasthi uses heartbeats to monitor resources, and initiates a failure detection if heartbeats from a resource are missing or if a resource signals that it suspects another resource has failed. However, failure detection has been a very hard problem, and no single solution works in every case. Hence, Hasthi allows users to plug-in their custom failure detection algorithms.

Above discussion looked at the complexities in general, and in the next section, we shall look at the case study, which describes how everything fits together to manage a real life system.

6. Case Study on Managing LEAD

6.1. LEAD Cyberinfrastructure

The LEAD Cyberinfrastructure[1] [10] is a large-scale distributed system organized as a Service Oriented Architecture. LEAD enables domain scientists to find, process, and assimilate an array of real-time observational weather data collected from observational sources across the United States. A LEAD user accesses the system through a web portal, obtains weather data, and carries out analysis, modeling, or mining tasks by means of assembling workflows. LEAD workflows are composed of command line applications (e.g. C or FORTRAN) wrapped as transient web services that are created (and re-utilized) at runtime, and we call them "application services". When an application service is invoked, the service parses the request for inputs and executes the underlying application on a large computational resource like the TeraGrid.

Workflow execution is orchestrated by Apache ODE, a WS-BPEL [7] workflow engine, which executes the tasks defined by data and control dependencies after it binds the abstract workflow description to concrete application services either by using existing service instances from a registry or by creating new instances using a service factory. Each workflow publishes events depicting the current state of the execution to a Message Broker, and multiple clients receive and process these events. For instance, the data subsystem catalogs, archives, and associates results of workflows with user accounts; therefore, users can find and use those data products at a later time.

[1]This work is funded through NSF grants ATM-0331594, ATM-0331591, ATM-0331586, ATM-0331587, and ATM-0331578

6.2. Managing LEAD System

LEAD workflows do not have any side effects outside the system, and even if a workflow has failed and re-executed, the data system can clean up any duplicate data products generated by the workflow re-executions. Furthermore, LEAD services either are stateless where they do not remember any state across two requests (e.g. service factory), or have a persistent state where all changes are written to a database straight away (e.g. service registry, meta-data catalog). Therefore, after failed and restarted, they can recover the state from the database. Consequently, the services, and therefore, the system, will not lose any critical state due to failures.

Hasthi agents have been integrated with all LEAD services and hosts and resource profiles have been setup, so Hasthi can start and stop services in need and create any missing services in the system at the startup.

As explained in Section 5, Hasthi detects service and host failures using heartbeats (generated by agents) and custom failure detectors. Furthermore, Hasthi detects workflow failures by listening to events generated by LEAD workflows depicting their progress. To identify errors from workflow events, we have compiled a collection of error patterns based on an analysis of earlier workflow errors, and Hasthi categorizes and identifies workflow errors occur in LEAD by matching error traces against these patterns.

Since software bugs, deployment errors, and configuration errors are caused by wide variety of reasons, it is hard to recover them automatically, and consequently, when these errors are detected, Hasthi notifies users using email messages. Furthermore, LEAD architecture already includes multiple levels of retries (independent of Hasthi) where it automatically retires most Grid operations and reruns failed jobs (parts of workflows) in alternative super computers. Since LEAD has multiple computation resources (super computers), these retires are effective against failures at computation resources; however, they do not handle errors at the LEAD infrastructure (e.g. service or host failures).

The rest of the discussion will focus on a management use case that recovers LEAD from infrastructure (services and host) failures. However, we believe the scenario we cover and solutions are general; therefore, they are applicable for most workflow-based systems. The following describes im-

plementation of this scenario using Hasthi while discussing associated rules and complexities.

The scenario includes three steps, 1) detecting system failures, 2) recovery of services, and 3) detecting the recovered healthy system state and recovering failed workflows. These steps are implemented using rules composed of two parts: a condition represented as a when-clause and an action represented as a then-clause, and when the condition is met, the action is carried out. Following are 3 rules we used to implement this scenario.

Rule 1

```
rule "LogSystemNonHealthyTime"
salience 10
when
  systemHealth:SystemHealthState(systemHealthy == true);
  exists(ManagedService(state == "CrashedState"
    ||state =="FaultyState" || state == "UnRepairableState"
    || state == "RepairingState", category == "Service"));
then
  systemHealth.setSystemFailed();
  update(systemHealth);
end
```

The first rule evaluates the system and declares the system as faulty when the system has at least one service that is in a non-functional state: "CrashedState," "FaultyState", "UnRepairableState'", or "RepairingState".

Rule 2

```
rule "RecoverFailedServices"
salience 4
when
  service:ManagedService(category == "Service",
    state == "CrashedState")
then
  ActionHelper.doRecoverFailedServices(service ,host ,system);
end
```

Rule 2 recovers failed services. Hasthi triggers the rule if a service is in the "Crashed" state, and it restarts the service in the same host if the host is active or otherwise restarts the service in a different host defined in the service profile. As described in Subsection 5.4, if an action fails, Hasthi marks the service as "Unrecoverable" and requests human help.

In the case of database failures, Hasthi detects the error and performs a user-interaction to request a user to fix it (rule is not listed). Since databases run as root in our system and their failures are rare compared to services, we do not perform automated recovery of databases.

Rule 3

```
rule "ResurrectWorkflowsAfterRecovery"
when
  not exists(ManagedService(state == "CrashedState"
    ||state =="FaultyState"||state == "UnRepairableState"
    ||state == "RepairingState", category == "Service"));
  systemHealth:SystemHealthState(systemHealthy == false);
then
  long failedTime = systemHealth.getSystemFailedTime();
  systemHealth.setSystemHealthy();
  ActionHelper.doResurrectWorkflowsAfterRecovery(system,
    failedTime);
  update(systemHealth);
end
```

The final rule recovers workflows. If the system does not include any "Faulty", "Crashed", or "Unrecoverable" services—that is when all services have recovered—Rule 3 is triggered. It declares that the system is healthy and recovers workflows that failed due to infrastructure failures.

The LEAD Management Utility (LMU) service, which is a LEAD specific service that aids Hasthi in performing LEAD specific tasks, collects and stores workflow events in a database. To find the failed workflows, a code triggered by the rule searches the LMU service database for all failed workflow that failed during the time the system was faulty, then matches the failure stack traces from results against known error patterns to identify errors due to infrastructure failures, and selects workflows failed due to infrastructure failures. The result of this search is a list of keys (workflow-ids), where each key corresponds to a workflow, and we use these keys to recover workflows.

When a workflow is initialized, it generates a special notification message, and among other events, LMU database also includes this event. There is one notification of this type for each workflow, and it includes the workflow-id and the request message for the workflow. Hasthi finds corresponding notification for each workflow by querying the database using the workflow-id, reconstructs the request message, and sends the message to the workflow engine to restart the workflow. Since workflows do not have any side effects outside the system and the data subsystem filters any duplicate files thus handing internal side effects, rerunning workflows does not have any adverse effects.

When a service crashes, Hasthi will detect it and restart the service as described. However, when the underlying hardware fails, the service must be migrated to a different location and this new location must be communicated to the other services in the system. As explained in Section 5, Hasthi provides a service-discovery operation, which can be used by services in the system to find other services. In LEAD, the service locations are disseminated in the SOAP header called "LEAD context header" that contains all service locations. It is propagated through the workflow request to every message of a workflow, and every service retrieves the locations from the header. LEAD Portal, the entry point to LEAD, constructs this header before launching a workflow, and on the process, it uses the service-discovery operation of Hasthi to obtain the current service endpoints. Furthermore, the workflow recovery code updates endpoints in the workflow request message using the service-discovery operation before the message is replayed to restart the workflow. With this approach, both new and restarted workflows will use the most up-to-date service endpoints.

This complete scenario has been implemented and deployed with LEAD, and the screen-cast given in [3] depicts this scenario. We encourage the reader to view it, as it will provide a good understanding about the afore-explained scenario.

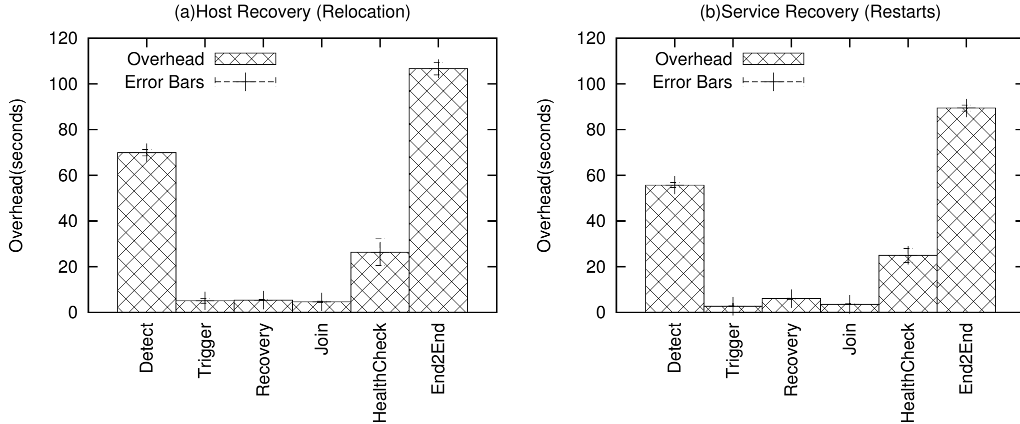

Figure 3. LEAD Recovery Times with Hasthi

7. Evaluations

To evaluate the Hasthi integration with LEAD, we have performed the following experiments by injecting failures into the system. The LEAD deployment consists of 26 services, deployed in 6 nodes with Dual AMD 2.0-2.6GHz Opteron CPUs, 16-32GB memory, Red Hat Linux, and 1Gb network. Hasthi has been deployed with 3 managers, and all control-loops and heartbeat intervals are set to 30 seconds.

We tested both of the scenarios described in the former section. The first experiment killed a service in the LEAD system, and measured the time it took for Hasthi to detect the error, to trigger corrective actions, to run the corrective action, to a new resource to join, and to detect that the system had recovered. Readings are measured using timestamps of events generated by Hasthi depicting its activities. Figure 3 depicts the results. The above readings are represented by labels, Detect, Trigger, Recovery, Join, and Health Check respectively, and the End2End represents the overall time for recovery. Similarly, in the second experiment, we simulated a host failure by killing all LEAD and Hasthi related processes in a host and then measured the aforementioned recovery overheads. Both cases were performed 100 times each, and the results are depicted in Figure 3 in which all values are averages and the error bars represents 95% confidence intervals.

As shown by Figure 3, the recovery took on average about 107 seconds for a host recovery (relocations) and about 89 seconds for a service recovery (restarts). Both cases spent about 60% of the recovery times detecting failures and 25-28% of the recovery times detecting that the system had recovered, and on both cases, actual times spent on detecting failures and detecting healthy system were about 60 seconds and 25 seconds respectively. Furthermore, Hasthi was setup with 30 seconds for the epoch time—the time-period between periodic executions of management control-loops—and it starts failure detection if two consecutive heartbeats are lost, and this (30×2) explains 60 seconds of detection time. On the other hand, even when services have recovered and new services have joined the system, Hasthi only decides that the system is healthy when the control-loop is executed for the next period, which happens within about 30 seconds, and this explains the observation that Hasthi took 25 seconds on average to ascertain that the system had recovered.

Furthermore, using the recovery time, we can approximate the availability of LEAD when managed with Hasthi. Ignoring failures of Hasthi, assuming the above two scenarios captures unavailability in the LEAD system, and assuming 26 LEAD services are independent with each having a MTTF (mean time to failure) of f, according to Baumann [8], the MTTF of the system is $\frac{f}{26}$.

Therefore, the availability of LEAD is $A = \frac{MTTF}{MTTF+MTTR} = \frac{f/26}{f/26+107}$. For example, with 1 month MTTF for each service ($f = 30 * 24 * 60 * 60 = 2592000$), the availability is $(604800/26)/(107 + (604800/26)) = 0.999$, about a 8.8 hours downtime per year. Similarly, when the MTTF of a single service is 1 week and 2 weeks, the availability is 0.995 and 0.997, which are about 43.8 and 26.3 hours downtime per year.

8. Conclusion

This paper discussed the process of using management frameworks to manage systems, associated complexities, and potential solutions by using Hasthi—a management framework that manages systems by enforcing user-defined management logic—as an example. Among those complexities are handling lost state, failed requests/sessions, broken system structure, failed actions, and fail positives. Moreover, we also showed that Axis2 Hasthi agent can be integrated with existing services purely via Axis2 configurations without any changes to service implementations, and we believe this process would greatly reduce the cost of ini-

tial adaptation. Furthermore, we exemplify above observations by presenting a real life case study of using Hasthi to manage an E-Science Cyberinfrastructure. We have evaluated the system by injecting failures into the system and measuring the breakdown of the time to recovery. Based on these results, we observed that Hasthi recovers the system within about 100 seconds, and we used that result to approximate the availability of the LEAD system managed with Hasthi to be about 0.99-0.999, which places LEAD in the availability classes "managed" and "well-managed" according to Gray et al. [14]. We acknowledge these numbers are only an approximation, yet they provide a glimpse of possibilities.

What are implications of these results? More than 20 years worth of efforts have uncovered many subtleties of building system management frameworks and efficient solutions, yet unlike other technologies, automated system management has not been in much use. Existing applications are mostly in managing large-scale deployments but rarely in moderate settings. A potential reason to this problem could be that although integrating management frameworks give rise to many complexities, neither those complexities nor potential solutions have been studied in detail, which reduces their adaptation. On these setting, a primary observation we made is that management frameworks do not guarantee safety property. In other words, even though they recover the system following a failure, they do not guarantee its behavior while recovery. Direct ramifications of this observation are three problems we identified in Section 5.4—lost state, failed requests/sessions, and broken system structure after recovery—which may yield an inconsistent system after recovery. Moreover, inherent uncertainties like failed management actions and fail-positives further complicate their adaptation. Consequently, one contribution of the above discussion is identifying common complexities, which would be useful for system designers. Furthermore, we propose common solutions to most problems and demonstrate those solutions applied to a Use Case. For example, we proposed re-executing or resuming failed requests or sessions to recover from the third problem and exemplified the idea by using management rules to recover failed workflows after the system has failed and recovered.

We believe this and other solutions and examples of their implementations in LEAD will be useful for system developers and administrators who would want to use a system management framework to manage their systems. Consequently, our primary contribution of this paper is discussing complexities of applying a system management framework to manage systems, proposing solutions on top of Hasthi or making recommendations regarding each, and presenting a detailed usecase, which uses Hasthi to manage an E-Science Cyberinfrastructure called LEAD.

References

[1] Apache axis2 project. online. http://ws.apache.org/axis2/.

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

[3] Hasthi lead integration details. online. http://extreme.indiana.edu/hasthi/lead/.

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

[5] E. Adams. Optimizing Preventive Service of Software Products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[6] 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.

[7] T. Andrews, F. Curbera, et al. Business Process Execution Language for Web Services 1.1, May 2003.

[8] J. Baumann. *Mobile Agents: Control Algorithms*, section Appendix B, Introduction to Fault Tolerance. Springer Verlag, 2000.

[9] G. Candea, A. Brown, A. Fox, and D. Patterson. Recovery-Oriented Computing: Building Multitier Dependability. *COMPUTER*, pages 60–67, 2004.

[10] K. Droegemeier et al. Linked environments for atmospheric discovery (lead): Architecture, technology road map and deployment strategy. In *International Conference on Interactive Information Processing Systems*, 2005.

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

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

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

[14] J. Gray and D. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.

[15] 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.

[16] T. Koch and B. Krämer. Rules and agents for automated management of distributed systems. *Distributed System Engineering*, 3:110–114, 1996.

[17] 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.

[18] S. Perera and D. Gannon. A Scalable and Robust Coordination Architecture for Distributed Management. 2008. Indiana University, Department of Computer Science Technical Report TR-659,2008.

[19] S. Perera, S. Marru, and D. Gannon. Monitoring and managing e-science cyber-infrastructures: A case study. *eScience, IEEE International Conference on*, 2008.

[20] G. Valetto and G. Kaiser. A case study in software adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 73–78, 2002.

[21] S. Vinoski. Chain of responsibility. *Internet Computing, IEEE*, 6(6):80–83, 2002.

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