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 present application of Hasthi management framework to manage workflow-based service architectures, 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. For that end, we present in detail the application of a concrete use case to manage a large, SOA based, E-Science Cyberinfrastructure, and discuss how some of the complexities have been addressed.

1. Introduction

Web services and SOA have become the de facto standard for most system usecases, where they have had much success with small and medium size systems. However, on one hand, with large-scale systems, the services based approach would lead to architectures that consist of many services running from different machines, and on the other hand, 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.

Large computations and data products, administrative and geographical distribution, and relatively large user bases characterize E-science Cyberinfrastructures; therefore, they easily give rise to large-scale systems that consist of many services. For example, the Linked Environment for Atmospheric Discovery (LEAD) Cyberinfrastructure [11] is a multi-disciplinary effort to enable meteorologists to search, process, assimilate, data-mine, and visualize weather data collected from multiple sources like radars, weather balloons, and airplanes. The primary goal of LEAD is to perform faster than real time weather forecasts by dynamically adapting to observational data. Moreover, LEAD is one of the first few Cyberinfrastructures to adopt a SOA based architecture to support an E-Science use case of this scale. However, due to its scale, the resulting architecture includes about 15-30 persistent services and many transient services created on demand, thus, giving rise to many complexities. Consequently, operating the LEAD system in production took the constant attention of a team of about 10 developers.

Summarizing, the administrative cost of complex real-life systems is high, and both the Autonomic Computing Initiative [13] 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, the Hasthi [19] 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 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 the LEAD system as a case study.

On the other hand, with a management framework like Hasthi that supports user-defined management logic ex-
pressed using a Turing complete language, it is possible to implement any practical management scenario. In this setting, possible management scenarios are endless and are only limited by the imagination of the user. Therefore, for brevity, we focus on recovering from infrastructure and service failures in workflow-based systems, a use case that captures LEAD system requirements. However, since the chosen use case is generic, we believe these results will be useful with many other web services based workflow and E-Science systems.

A unique contribution of this paper is presentation and evaluation of a typical management use case for workflow-based systems that recovers a system from both service and deployment host failures and recovers workflows that were active at the time of the service or host failures and have incurred their own errors as a result.

The next section discusses the related work on the application of management frameworks, and the following two sections illustrate the LEAD system and Hasthi. Section 5 describes the methodology we proposed to integrate a management framework with a given system and illustrates a management use case that recovers the system from both service and host failures. The following section presents an evaluation of the implementation. Finally, Section 7 concludes the paper.

2. Related Work

There are a number of management frameworks found in the system management literature (e.g. [16, 12, 6, 23, 14, 18]), and Perera et al. [19] 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 a few earlier works discuss application of system management; among them, Valetto et al. [21] present a case study that manages a few Internet Messaging servers using KX management framework and Koch et al. [17] present a general discussion on applying the Marvel rule-based system management framework [18] in order to recover from failures reactively. In contrast, we present a detailed discussion on managing a much more complex system and then present a generic management use case, which is 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 [20] in the E-Science conference, 2008.

3. LEAD Cyberinfrastructure

The LEAD Cyberinfrastructure [11] 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, obtain weather data, and carry 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. Furthermore, the application services register the data products and metadata generated from these executions with a metadata catalog, called XMC Cat. Those data can be found later by searching the metadata catalog.

4. Hasthi Management Framework

4.1. Outline of the Architecture

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/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, a 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 are detected by the absence of heartbeats. On this setting, if the coordinator failed, the manager heartbeats will fail, and managers start an election to elect a new coordinator. Alternatively, if a manager failed, the coordinator detects it via missing heartbeats and the resources detect it when sending resource heartbeats fail; then coordinator removes the manager from the manager-cloud and the resources restart the join process and rejoin the system. Moreover, to avoid multiple coordinators caused by communication failures, the coordinator periodically broadcasts its existence, and except for one, other coordinators resign. In summary, Hasthi recovers from managers, resources, and the coordinator 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 matrices 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 includes collected matrices 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.

Comprehensive design and analysis of the Hasthi framework are discussed in Perera et al. [19].

4.2. User view of Hasthi

In the rest of this section, we present few significant functionalities of Hasthi.

Hasthi uses the WSDM specification as the interface between resources and Hasthi with an additional extension to generate periodic heartbeat messages from the resource to the manager. To facilitate exposing resources as managed resources described in the WSDM specification, Hasthi provides agents, which can be integrated with an existing service or a host. For instance, there is an agent for Axis2 [1] based services, which exposes Axis2 services to Hasthi. After integrated with the agent, each service joins the Hasthi manager-cloud at the startup where it will be monitored and controlled by Hasthi.

Each service has a property called “type,” which is in-fact the port type name of its WSDL [10], and each different type of service in the system must have a resource profile defined via the Hasthi configuration. A sample profile is given below.

A Resource Profile

As shown by the above listing, the profile describes the service deployment and behaviors, and Hasthi uses the profile to perform management actions. To start and stop services, Hasthi supports tomcat based service installations by default, but with non-tomcat based services, these actions can be implemented 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. Also, each service may define dependencies to other services, where dependencies says 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.

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.

The life-cycle of a managed resource is depicted by Fig-
Figure 1. 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, the rule performs a user interaction by sending an email to a human user asking him to fix the error and respond by clicking a link in the email. By changing a 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, so loops do not occur.

Furthermore, utilizing the summarized meta-model of the systems, 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). Thus, Hasthi acts as a service registry, and using this, services can discover other services in the system, both at the start and when looking for an alternative endpoint because a dependent service has failed.

4.3. Hasthi Agent for Instrumenting Services

As described before, Hasthi includes an agent that can be integrated with existing web services, and once integrated, these services 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. However, in this section, we describe the agent developed for Axis2 based services, as we believe this is the agent most relevant for our audience.

The agent is developed as an Axis2 module. Therefore, the agent can be integrated and removed 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 [22], 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. Managing Workflow-based Systems

5.1. Methodology of Integration

Figure 2 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] &
Gray et al. [15] 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 a large proportion of error occurrences in the system. Adams’s observation is confirmed by an analysis we conducted on LEAD error data over an 18 month period where we observed that 30 out of 80 different error types are 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 rule to manage the system with Hasthi. Furthermore, the scenarios and the resulting rules can be improved iteratively.

5.2. Managing LEAD System

As described before, with LEAD, a user comes to the portal, searches for weather data based on some criteria, and then processes that data by running workflows. Meanwhile, 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.

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.

We classify failures in LEAD as infrastructure failures and workflow execution failures. Since Hasthi monitors services and infrastructure, it directly detects infrastructure failures and reacts to them. Regarding workflow failures, LEAD workflows generate events depicting their progress, and Hasthi listens to those events and tracks workflow progress, success, and failures. Furthermore, the LEAD Management Utility (LMU) service, which is a LEAD specific service that aids Hasthi in performing LEAD specific tasks, collects and stores these events in a database. To identify errors in LEAD, we have compiled a collection of error patterns based on analysis of earlier errors, and Hasthi categorizes and identifies errors occur in LEAD by matching error traces against these patterns.

This paper focuses on recovery from infrastructures errors. Let us briefly look at other error types. Among other error types, file transfers and job submission errors account for a significant portion of LEAD errors. However, computations may be done using one of many supercomputing compute nodes; therefore, these errors are handed by retries built into the LEAD system, which uses an alternative compute node to recover the workflow. Furthermore, 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.

The rest of the discussion will focus on a use case that recovers a workflow-based system from services and host failures. However, the scenario we cover and solutions are general; therefore, they are applicable for most workflow-based systems. The following describes implementation 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 4 rules we used to implement this scenario.

Rule 1

```
rule "LogSystemNonHealthyTime"
  salience 10
  when
    systemHealth : SystemHealthState ( systemHealthy == false ) ;
    exists ( ManagedService ( state != "CrashState" || 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: “CrashState,” “FaultyState”, “Un-RepairableState”, or “RepairingState”.

Rule 2

```
rule "RecoverFailedHost"
  salience 4
  when
    host : Host ( state != "BusyState" && state != "IdleState" && state != "SaturatedState" ) ;
    service : ManagedService ( host == host . name , category == "Service" , state == "CrashState" ) ;
  then
    ActionHelper . doRecoverFailedHost ( service , host , system ) ;
end
```

Rule 3

```
rule "RestartFailedServices"
  salience 4
```

...
Two rules are needed to recover failed services. Specifically, if a service has failed but the residing host is active, Rule 2 restarts the service, and alternatively, if a host has failed, Rule 3 restarts all services in that host using a different host defined in the service profile. It is worth noting that the create service action pings the host agent of the target host to check its health before creating the service, and if the host has failed, the action tries to find an alternative host from the service profile. Consequently, if a host has failed but not yet detected, actions detect the failure when trying to create a service in the host and use another host.

When management action finishes, the recovery code marks the resources as “Repaired” if successful and as “Un-recoverable” if failed. Furthermore, if the action has failed, an email is sent to the user, and the email asks the user to recover the service manually and click a link in the email to notify Hasthi. After the user clicks the link, Hasthi receives it as a REST web service call and marks the the service as “Repaired”.

In the case of database failures, Hasthi detects the error and performs a user-interaction to request a user to fix it. 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 4**

```java
rule "ResurrectWorkflowsAfterRecovery"
salience 5
when
  not exists (ManagedService (state == "CrashedState", || state == "FaultyState" || state == "UnRepairableState" || state == "RepairingState", category == "Service"));
  systemHealth : SystemHealth (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—the rule 4 is triggered. It declares that the system is healthy and recovers workflows that failed due to infrastructure failures. 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 matching workflows. 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 LMU service listens to notifications and saves that message to the LMU database. 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, reconstructed 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 4, 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. The 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.

### 6. 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 the system to detect the error, to trigger corrective actions, to run the cor-
rective 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 epoch time—the time-period between periodic executions of management control-loops. Hasthi starts failure detection if two consecutive heartbeats are lost, and this 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 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 
\[
A = \frac{MTTF}{MTTF + MTTR} = \frac{\frac{f}{26}}{\frac{f}{26} + 107}.
\]

For example, with 1 month MTTF for each service (\( f = 30 \times 24 \times 60 \times 60 = 2592000 \)), the availability is \((604800/26)/(107 + (604800/26)) = 0.999\), about a 10 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 40 and 20 hours downtime per year.

<table>
<thead>
<tr>
<th>Action</th>
<th>mean (ms)</th>
<th>action count</th>
<th>95% confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send E-Mail</td>
<td>520</td>
<td>137</td>
<td>[462, 578]</td>
</tr>
<tr>
<td>ShutDown</td>
<td>3697</td>
<td>57</td>
<td>[1637, 5756]</td>
</tr>
<tr>
<td>Create Service</td>
<td>6688</td>
<td>806</td>
<td>[6511, 6865]</td>
</tr>
<tr>
<td>User Interaction</td>
<td>1177</td>
<td>99</td>
<td>[677, 1677]</td>
</tr>
</tbody>
</table>

Table 1 presents the latency of different management actions performed by Hasthi within two weeks of testing. The stop service and create service actions also include the time for pinging the service to verify that the action has been successfully completed. Furthermore, the user interaction action only includes the time to generate and send the request, not the time for user to respond. All actions times are within acceptable limits and consumes only a small portion of the overall recovery time.
7. Conclusion

This paper presented Hasthi from the users point of view and described how it could be useful in managing Web services based system. In addition, we proposed that to integrate a system Hasthi, user should use the 80/20 rule to identify the most common errors, design management scenarios to cover those errors, and author management rules to implement those scenarios.

We have followed the above process to integrate Hasthi with LEAD, a large-scale E-Science cyber-infrastructure. Currently, the LEAD development stack is being managed entirely by Hasthi in auto-pilot mode causing a significant reduction in man-power requirements for maintaining the LEAD stack in production and thereby increasing the system success rates and response times. 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. [15].

Furthermore, we discussed handling complexities like notifying the service location to other services after a service migration, handling services state, and handling management action failures. Moreover, we also showed that Axis2 Hasthi agent can be integrated with existing services purely via Axis2 configurations without any changes to service implementations. We believe this process would greatly reduce the cost of initial adaptation.

As illustrated in the introduction, the application of a management framework to a given system is far from trivial, but we observed that this topic was given a limited attention in literature. Consequently, our primary contribution of this paper is presentation and evaluation of a typical management use case for workflow-based systems that recovers a system from both service and deployment host failures and recovers workflows that were active at the time of the service or host failures and have incurred their own errors as a result.

In the final analysis, we presented a use case that manages workflow-based systems using Hasthi and then presented in detail, an application of the use case to manage a large, SOA-based, E-Science Cyberinfrastructure. Furthermore, we discussed in detail many complexities associated with the use case and demonstrated its viability by injecting failures into the LEAD system integrated with Hasthi and observing recovery characteristics.

References