

Application of Management Frameworks: A Case Study on Managing Workflow related Systems

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 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. In this paper, we present a generic management usecase for managing workflow related systems using Hasthi, a robust, scalable, and distributed management system, which enables users to manage a system by enforcing management logic authored by users themselves. Furthermore, we present in detail the application of the use case to manage a large, SOA based, E-Science cyber-infrastructure, and discuss how some of the complexities are 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, with large-scale systems, the services based approach would lead to architectures that consist of many services running from different machines where a failure of any service often cause the system to fail. Therefore, maintaining such a system requires administrators to monitor those services continuously.

Large computations and data products, administrative and geographical distribution, and relatively large user bases characterize E-science cyber-infrastructures, and therefore, they easily give rise to large-scale systems that consists of many services. For an example, the Linked Environment for Atmospheric Discovery (LEAD) cyber infrastructure [10] 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. Furthermore, the primary use case of LEAD is to perform faster than real time weather forecasts using those data. Moreover, LEAD is one of the first few cyber-infrastructures to adopt a SOA based archi-

itecture to support an E-Science usecase of this scale, and due to the scale, the resulting architecture includes about 15-30 persistent services and many transient services created on demand. Therefore, the resulting architecture was complex, and consequently, keeping the LEAD system operational took the constant attention of a team of about 10 developers.

Summarizing, the administrative cost of complex real-life systems are high, and both Autonomic Computing initiative [12] and Recovery Oriented Computing Initiative [8] have cited much evidence to demonstrate this observation. On this setting, automating system management is an attractive and viable solution to this problem. It is not only cost effective, but could also tremendously increase the system availability. Among such frameworks, the Hasthi framework [18] is one such a robust, scalable, and distributed management system, which enables users to manage a system by enforcing management logic authored by users themselves.

Even though management architectures are well discussed in the literature, their 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 are formulating management scenarios, handling the state of 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 is changed after recovery. Consequently, the application of a management framework is a topic that warrants detailed analysis, yet seldom explored. Furthermore, we believe that at least some of those problems have generic solutions where implications of those solutions are far-reaching and general. In this paper, we try to answer some of these problems with Hasthi, using the LEAD system as a case study.

On the other hand, with a management framework like Hasthi that supports user-defined management logic specified as a Turing complete language, it is possible to implement any practical management scenario. On this set-

ting, possible management scenarios are endless and only limited by the imagination of the user. Hence, for brevity, we focus on recovering from infrastructure and service failures in workflow related systems, a usecase that captures LEAD system requirements. Furthermore, since the usecase is generic, we believe these results will be useful with many other web services based workflow and E-Science systems.

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

2. Related Work

There are a number of management systems found in the system management literature (e.g. [15, 11, 5, 21, 13, 17]) and Perera et al. [18] presents a detailed comparison of the Hasthi framework to those existing management systems. In comparison to those systems, the primary advantage Hasthi offers is the ability to run a user-defined global control loop over a large-scale system where resources are managed by multiple managers. However, since the paper focus on application of Hasthi, not Hasthi itself, we will not spell out details of comparisons.

Among other real world applications of system management, Valetto et al. [19] 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] to recover from failures reactively. In contrast, we present a detailed discussion on managing a much more complex system and present a generic management usecase, which is useful for managing other web services based workflow and E-Science systems.

3. LEAD Cyber-Infrastructure

The LEAD infrastructure [10] is a Service Oriented Architecture based system, which enables meteorologist to find, process, and assimilate real-time observational weather data collected from multiple sources across United States. A LEAD user login in to a web portal, search for weather data from catalogs, and process weather data using workflows where the backend of the portal consists of a complex service-based infrastructure. The workflows are composed of command line applications (e.g. C or FORTRAN) wrapped as transient Application Services that are created (and re-utilized) at runtime, where Application Services wrap data assimilation, mining, and weather applications

as Web Services. When an Application Service is invoked, the service parses the request for inputs and executes the underline application on a large computation resource like TeraGrid.

Workflows are orchestrated by Apache ODE, a WS-BPEL [6] based 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 instances from a registry or creating new instances using a service factory. Each workflow publishes events depicting the current state of the execution to a Message Broker, where multiple clients receive and process those events. Furthermore, Application Services register data products generated from these executions with a Metadata Management Agent, which copies the file to an archived location and registers the metadata in a Metadata catalog Server. Therefore, those data can be found later by searching the meta-data 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 in 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 this subset is flexible and usually defined by resource developers. Furthermore, when a resource joined Hasthi, it is assigned to a manager, monitored, and controlled. An interesting reader should refer to Perera et al. [18] for a detailed design and analysis of the Hasthi framework.

The Hasthi architecture consists of a dynamic and robust management cloud constituting of managers, a coordinator, and a set of bootstrap nodes, where managers manage resources and the coordinator oversees the managers. Furthermore, if the coordinator fails, a new coordinator is elected among other managers. Each managed resource joins the management cloud via bootstrap nodes running in advertised endpoints, and bootstrap nodes forward the join message to the coordinator, which in turn assigns the resource to a manager. Having assigned to a manager, each resource periodically sends heartbeat messages to the assigned manager and each manager periodically sends heartbeat messages to the coordinator, and therefore, failures of both resources and managers are detected by the absence of heartbeats. Moreover, if the coordinator failed, the manager heartbeats will fail, and managers start an election to elect a new coordinator. On the other hand, if a manager failed, resource heartbeats will fail and resources restart the join process and join the system again. Furthermore, to avoid

multiple coordinators due to communication failures, the coordinator periodically broadcasts its existence, and if two coordinators are present, one would eventually resign in favor of the other. 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. In addition, 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 since the last heartbeat to the manager, where the assigned manager creates a meta-object for the resource and updates the meta-object when heartbeats arrived from the resource. Furthermore, the coordinator maintains summaries of individual meta-objects located at various managers, and those summaries are updated through manager heartbeats. Consequently, there is a meta-object of each resource—a copy of resource properties of the resource—maintained in the manager and a summarized version of the meta-object is kept at the coordinator, and furthermore, Hasthi keeps both types of meta-objects up-to-date. Therefore, meta-objects reflect the current state of the system.

Furthermore, each manager contains a control loop that evaluates user-defined management rules using meta-objects of 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. In addition, rule evaluations may trigger management actions in response to failures in the systems, which are carried out by the associated coordinator or the manager.

4.2. Hasthi from User's point of view

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

Hasthi uses the WSDM specification with the addition of an extension to generate periodic heartbeat messages from the resource to the manager, and the specification is the interface between resources and Hasthi. Furthermore, Hasthi includes agents, which can be integrated with an existing service or a host to expose it as a managed resource described in the WSDM specification. For an instance, there is an agent for Axis2 [1] based services, which exposes Axis2 services to Hasthi. At the startup, any service that has the agent integrated joins the Hasthi manager cloud 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 [9], 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

```
<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>tyrl6.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, where Hasthi uses the profile to perform management actions. To start and stop services, Hasthi supports tomcat based service installations by default, and for non-tomcat based services, services start and stop commands must be implemented as shell scripts. Moreover, 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 those hosts. Furthermore, each host in the system runs a Hasthi host agent, which monitors the host and enables Hasthi to perform management actions like start/stop services by executing required shell commands on that host. Also, each resource may define dependencies, and a dependency means that to start the given resource, an instance of all dependencies must be running. Furthermore, Hasthi can initialize the system where it uses dependencies to decide the bootstrap order of services, and when Hasthi starts up, it waits for some time to discover all services in the system and creates any missing services based on the system profile and service dependencies.

Users can define management logic that decides how Hasthi will react to changes in the system, and the Drools rule language [2] is used to express management logic. Furthermore, control loops at managers and the coordinator periodically evaluate these rules, where the rules trigger management actions in response to error conditions. We will revisit rules in the section 6.

The life-cycle of a managed resource is depicted by the figure 1. Among states, 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. Furthermore, to detect crashed or failed services, Hasthi uses heartbeats and a failure detector where if two heartbeats from a particular service is missing, the failure detector is invoked and the service is marked as crashed based on the outcome. Currently, the default failure detector simply

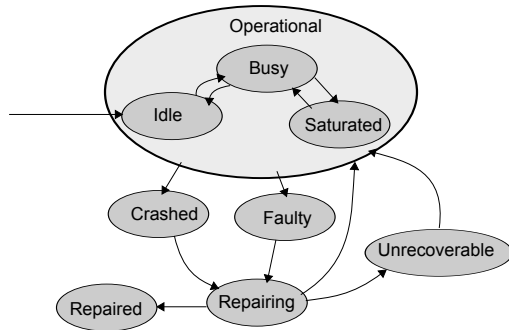


Figure 1. Life Cycle of a Managed Resource

pings services for failure detection. In addition to this process, management rules can decide a service is faulty based on conditions like 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, where it is marked as “Repairing” before the action is carried out and it is marked as “Repaired” when the action has completed. On the other hand, 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 where an email is sent 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 after a resource is evaluated and acted upon by rules, Hasthi guards against the possibility of indefinite loops of recovery. For an example, if a management action failed, the resource state is set to “Unrecoverable”, and therefore, no more actions are performed on the resource.

Furthermore, Hasthi includes a service lookup 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 this is useful for services to discover other services in the system, both at the start and when looking for an alternative endpoint because a dependent service has failed.

Finally, Hasthi includes a web-based dashboard, which allows users to monitor services in the system, browse properties exposed by each resource, and retrieve information like workflow success statistics, recent management actions, and recent log events generated by Hasthi. Furthermore, Hasthi includes a java swing-based User interface (UI) that enable user to browse, start, stop, restart services. Moreover, using the UI, users can switch the control loop of Hasthi on and off, which would be useful when someone is debugging a part of the system and Hasthi is getting in the way. Both these UIs are illustrated in the screen cast available from [3].

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, those services can be managed and monitored using Hasthi. In fact, Hasthi includes several agents, targeted for hosts, web services, and UNIX processes to name few. However, in this section, we describe the agent developed for Axis2 based services; as we believe that would be the agent most relevant for our audience.

The agent is developed as an Axis2 module, and therefore, the agent can be integrated and removed purely by changing configurations of existing services without any changes to the service implementation. For an 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 [20], these modules enable users to inject custom interceptors (a.k.a. Axis2 Handlers) to the axis2 message processing pipeline. To implement the Hasthi module, we have developed a Hasthi Handler, which intercepts every message coming in to or going out of the axis2 container. The Handler has two functions. As the first, it intercepts all management messages and redirects them to the WSDM implementation of Hasthi, and as the second, by intercepting other messages, it collects statistics about the service like, 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, the resource becomes a manageable service described by the WSDM specification, and it can be monitored and controlled from an external authority.

5. Managing Workflow Based Systems

5.1. Methodology of Integration

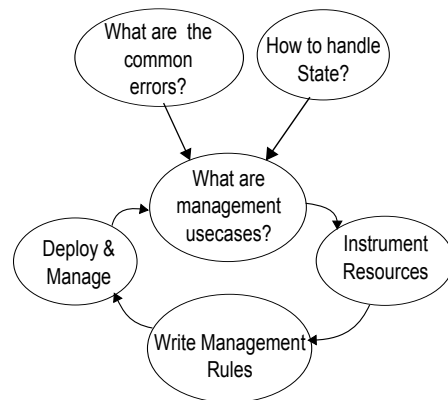


Figure 2. Methodology of Integration

We have formulated a methodology for integrating Hasthi

with a given system, and the figure 2 depicts the methodology. The basis for our methodology is an observation by Gray et al. [14] where they observed that most error occurrences are caused by few error types (a.k.a. Petro principle). Therefore, by handling few most common error types, we can cover most of error occurrences in the system. We have verified this observation with LEAD error data over 18 months 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 find most common error types using error statistics, identify the management scenarios that handle those common errors, integrate Hasthi agents with resources and expose resource properties that are required to implement those management scenarios, author rules to capture those management scenarios, and use those rule to manage the system with Hasthi. Furthermore, experiences will grant the user a greater understanding about management scenarios. Hence, the scenarios and 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 processes that data by running workflows. Meanwhile, the data subsystem catalog, archive, and associate results of workflows with user accounts; so that users can find and use those data products in a later time.

LEAD workflows do not have any side effects, and the LEAD usecase expects only a best effort service from the meta-data catalog and data archives. For an example, if a workflow has failed and re-executed, the data system can handle any duplicate data products generated by multiple workflow runs. Furthermore, LEAD services either stateless where they do not remember any state across two requests (e.g. workflow engine, service factory), or have a persistent state where all changes are written to a database straight away (e.g. service registry, meta-data catalog). Consequently, in the LEAD system, if a service has failed and restarted, the system would not lose any critical state. Since services store their state in a MySQL database, after restarted, new services can recover the state from the database.

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. Furthermore, Hasthi was configured to create any missing services in the system at the startup.

Errors in the LEAD system are two types, infrastructure errors that cause the whole system to fail and workflow execution errors. In particular, failure of a service, host, or some other key component of the system typically causes the infrastructure errors, and since Hasthi monitors services and hosts, it often captures the former error type, but does

not directly capture workflow execution errors. However, LEAD workflows generate events depicting their progress, and since workflow errors may also be signs of service failures, Hasthi also monitor those events. Furthermore, the LEAD Management Utility (LMU) service, which is a LEAD specific service that aids Hasthi with LEAD specific tasks, collects and stores those events in a database. Moreover, based on analysis of earlier errors, we have compiled a collection of error patterns, and Hasthi categorizes and identifies errors occur in LEAD by matching error traces against those patterns.

Among workflow errors, 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, and therefore, those errors are handed by retries built in to the LEAD system, which uses an alternative compute node to recover the workflow. Furthermore, software bugs, deployment errors, and configuration errors could not be automatically recovered, and consequently, when those errors are detected, Hasthi sends errors to users as email messages.

However, this paper focuses on a usecase that recovers the system from services and host failures in a workflow-related system. Furthermore, we believe that the scenario we cover and solutions are general, and therefore, they are applicable for most workflow-related systems. The following describes implementation of this scenario using Hasthi, where we present and describe associated rules and complexities.

The scenario includes three steps, 1) detecting system failures, 2) recovery of services, and 3) detecting the healthy state and recovering failed workflows. The steps are implemented using rules where each rule has 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. Let us look at rules that implement this scenario.

The first rule evaluates the health of the system and declares the system as faulty if at least one service found in the system that is in a non-functional state, which are “CrashedState”, “FaultyState”, “UnRepairableState”, or “RepairingState”.

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
```

Furthermore, the next two rules recover failed services. Specifically, if a service has failed but the residing host is active, the rule 2 restarts the service, and on the other hand,

if a host has failed, the rule 3 restarts all services in that host in a different host defined in the service profile. It is worth noting that the create service action checks the availability of the host before starting the create operation by pinging the host agent, and if some host has failed, it tries to find an alternative host from the service profile. This handles the case that if a host has failed but not yet detected.

Rule 2

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

Rule 3

```
rule "RestartFailedServices"
salience 4
when
  service:ManagedService(state == "CrashedState",
    category == "Service");
  host:Host(state == "BusyState" || state == "IdleState"
    || state == "SaturatedState", service.host == name);
then
  ActionHelper.doRestartFailedServices(service, host, system);
end
```

Subsequently, the recovery code marks the resources as “Repaired” if the recovery action is successful and marks the resource as “Unrecoverable” if the action has failed. Furthermore, if the action has failed, the code sends an email to the user, asking him to recover the service manually. Moreover, the email requests the user to respond back after repairing the service by clicking a link in the email, where the click is conveyed to Hasthi as a REST web service call, and when Hasthi receives the REST call, the service, which was marked as unrecoverable, is marked as repaired. In the case of a database failure, which was reasonably rare, Hasthi depends on humans for recovery where the rule marks the database as repairing and starts a user interaction similar to service recovery failure to request a human user to fix the database.

Rule 4

```
rule "ResurrectWorkflowsAfterRecovery"
salience 5
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
```

Moreover, if the system does not include any “Faulty”, “Crashed”, or “Unrecoverable” services—that is when all

services have recovered—the rule 4 is triggered, which declares that the system is healthy and recovers workflows failed due to service failures while the system was faulty. To find the failed workflows, a code triggered by the rule searches the database of the LMU service for all failures that occurred during the time period the system was faulty, matches the failure stack traces from results against known error patterns to identify errors due to service failures, and selects workflows to be recovered.

When a workflow is initialized, the request message sent to the workflow is included in a notification, which is saved to a database by the LMU service. To recover the workflow, the workflow recovery code retrieves the notification using the workflow-instance-id of the workflow to be recovered and re-executes the workflow by replying the request message to the ODE workflow engine, thus restarting the failed workflow. Since workflows do not have any side effect outside the system and the data subsystem filters any duplicate files thus handling internal side effects, rerunning workflows does not have any adverse effects.

As explained above, when a service crashes, Hasthi will detect it and restart the service. However when the default deployed hardware fails, the service has to be restarted on a different location, on which case, this new location must be communicated to the other dependent services. As explained in the chapter 4, to resolve dependencies, Hasthi provides a lookup, which returns a matching service endpoint given a service type name (WSDL port type). Furthermore, to disseminate service locations to services of the system, LEAD uses a SOAP header called LEAD context header that contains all service locations, and it is propagated through the workflow request to every messages related to the workflow and every service retrieve the locations from the header. This header is constructed by the portal, which is the entry point to LEAD, and the portal uses the service lookup operation of Hasthi to obtain the current service endpoints before launching a workflow. Therefore, the portal will use the most up-to-date service endpoints to start workflows. Furthermore, the workflow recovery code updates endpoints in the workflow request message before the message is replayed to restart the workflow.

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

6. Evaluations

The integration of Hasthi with the LEAD cyber-infrastructure has been completed, and currently, the LEAD development stack is been managed by Hasthi. To evaluate the Hasthi integration with LEAD, we have performed following experiments by injecting failures in to the system

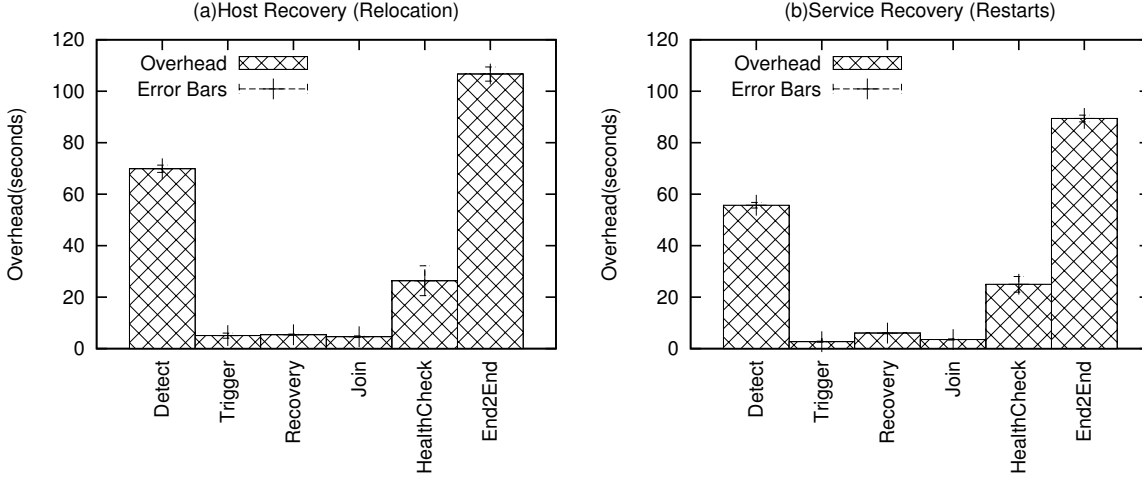


Figure 3. LEAD Recovery Times with Hasthi

where LEAD consists of 26 services, deployed in 6 nodes having Dual AMD 2.0-2-6GHz Opteron CPUs with 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 the both scenarios described in the former section, where the first experiment killed a service in the LEAD system, and measured the times for system 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 has recovered. The figure 2 depicts the results, and 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 simulate a host failure by killing all LEAD and Hasthi related processes in a Host and measured the aforementioned recovery overheads. Both cases were performed 100 times each, and results are depicted in the figure 2 where all values are averages and the error bars represents 95% confidence intervals.

As shown by the figure 1, the recovery took, in average about 107 seconds for a host recovery (relocations) and about 89 seconds for a service recovery (restarts). Furthermore, in both cases, about 60% and 25-28% of the recovery time were spent on detecting failures and detecting that the system has recovered, and actual times spend on those cases were about 60 seconds and 25 seconds. Furthermore, Hasthi was setup with 30 seconds as epoch time—the time-period between periodic executions of management control loops—and Hasthi starts failure detection if two consecutive heartbeats are lost. Therefore, two heartbeat time-periods explain 60 seconds of detection time. On the other hand, once the system has recovered and new services has joined the system, Hasthi only decides system is healthy when the control loop is executed for the next period, which happens

within about 30 seconds. Therefore, this explains the observation that Hasthi took 25 seconds to know that the system has recovered.

Furthermore, using the recovery time, we can approximate the availability of LEAD when managed with Hasthi. Ignoring failures to Hasthi, assuming 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 [7], the MTTF of the system is $f/26$.

Therefore, the availability of LEAD $A = \frac{MTTF}{MTTF+MTTR} = \frac{f/26}{107+f/26}$. For an example, with 1 month MTTF for each service ($f = 30 * 24 * 60 * 60 = 2592000$), availability is $(604800/26)/(107+(604800/26)) = 0.999$, about a 10 hours downtime per year. Similarly, the availability is 0.995 and 0.997 when the MTTF of a single service is 1 week and 2 weeks, which are about 40 and 20 hours down time per year.

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

Table 1. Management Action Overheads

Furthermore, the table 1 presents the latency of different management actions performed by Hasthi in 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 successfully completed. Furthermore, the user in-

teraction action only includes the time to generate and send the request, not the time for user to respond. All actions times are in 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 to manage Web services. However, 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 to Web services is the identification and a detailed discussion of a generic management usecase that captures recovery from services and infrastructure failures in a workflow related SOA system, and sharing our experiences implementing that usecase to manage a real life, SOA-based, E-Science cyber-infrastructure. 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.

Furthermore, some of other contributions are following. We motivate the use of the observation that most error occurrences are caused by few errors types—the 80/20 rule or the Pareto principle—to identify most critical errors and build management scenarios around it. On the other hand, the presented Axis2 Hasthi agent can be integrated with existing services, without any changes to service implementations, purely via Axis2 configurations, and we believe this would greatly reduce the cost of initial adaptation.

We have implemented the usecases completely using Hasthi and applied it to manage a large-scale E-Science cyber-infrastructure. Hasthi is online as part of the LEAD development service stack. Furthermore, we have evaluated the system by injecting failures in to the system and measuring the breakdown of the time to recovery. Based on those 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 classes defined by Gray et al. [14].

However, it is worth noting that even though the paper focused on a one usecases for brevity, the management logic of Hasthi is flexible. Therefore, by authoring rules, it is possible to implement most practical management usecases with Hasthi. Moreover, it is worth noting that Hasthi is designed for large-scale systems, nevertheless, even with 10-20 services, it can significantly reduce the cost of management and increase availability. However, it is unlikely to be useful for systems smaller than that limit.

In the final analysis, we presented a usecase that man-

ages workflow related systems using Hasthi and presented in detail, an application of the use case to manage a large, SOA-based, E-Science cyber-infrastructure. Furthermore, we discussed in detail many complexities associated with the usecase and demonstrated its viability by injecting failures in to the LEAD system integrated with Hasthi and observing recovery characteristics.

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. 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] T. Andrews, F. Curbera, et al. Business Process Execution Language for Web Services Version 1.1, May 2003. <ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf>.
- [7] J. Baumann. *Mobile Agents: Control Algorithms*, section Appendix B, Introduction to Fault Tolerance. Springer Verlag, 2000.
- [8] G. Candea, A. Brown, A. Fox, and D. Patterson. Recovery-Oriented Computing: Building Multitier Dependability. *COMPUTER*, pages 60–67, 2004.
- [9] E. Christensen, F. Curbera, G. Meredith, et al. Web Services Description Language (WSDL), Version 1.1, March 2000. <http://www.w3.org/TR/wsdl>.
- [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] 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.
- [20] S. Vinoski. Chain of responsibility. *Internet Computing, IEEE*, 6(6):80–83, 2002.
- [21] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an information plane for networked systems. *SIGCOMM Comput. Commun. Rev.*, 34(1):15–20, 2004.