AESON: A Model-Driven and Fault Tolerant Composite Deployment Runtime for IaaS Clouds

Deepal Jayasinghe, Calton Pu
CERCS, Georgia Institute of Technology
266 Ferst Drive, Atlanta, GA 30332-0765, USA
{deepal, calton}@cc.gatech.edu

Fábio Oliveira, Florian Rosenberg, Tamar Eilam
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
{fabolive, rosenberg, eilamt}@us.ibm.com

Abstract—**Infrastructure-as-a-Service (IaaS) cloud environments expose to users the infrastructure of a data center while relieving them from the burden and costs associated with its management and maintenance. IaaS clouds provide an interface by means of which users can create, configure, and control a set of virtual machines that will typically host a composite software service. Given the increasing popularity of this computing paradigm, previous work has focused on modeling composite software services to automate their deployment in IaaS clouds. This work is concerned with the runtime state of composite services during and after deployment. We propose AESON, a deployment runtime that automatically detects node (virtual machine) failures and eventually brings the composite service to the desired deployment state by using information describing relationships between the service components. We have designed AESON as a decentralized peer-to-peer publish/subscribe system leveraging IBM's Bulletin Board (BB), a topic-based distributed shared memory service built on top of an overlay network.**

Keywords-**Failures, Fault-Tolerant, Model-Driven, Peer-to-Peer, Publish-subscribe, Recovery**

I. Introduction

The emergence of cloud computing has enabled a new model of deployment and management of applications and services. In particular, with the advent of Infrastructure-as-a-Service (IaaS), many companies started relying on external cloud providers to host some of their Web and enterprise applications. One of the motivations for the adoption of IaaS clouds is to shift the burden of IT management to the cloud provider so that companies can focus on their own core businesses.

Typical applications deployed to IaaS clouds require complex software stacks comprising a multitude of distributed, cross-dependent components such as load balancers, Web, application, and database servers. At deployment time, all required software components and middleware must be cross-configured to ensure the application works correctly, and the entire infrastructure must also be configured to support non-functional requirements such as performance, availability, and security. Not surprisingly, deploying and correctly configuring such applications is a non-trivial, error-prone proposition. In order to mitigate this problem, previous work has focused on modeling such applications to automate their deployment and configuration in IaaS clouds ([1], [13], [15]).

Given the massive scale of the data centers of IaaS providers, which comprise a large number of heterogeneous hardware components, it comes as no surprise that hardware failures happen often in these environments, not to mention human mistakes made during data center operation, aggravating this problem. Failures in the IaaS data centers can translate into partial or complete outages of the hosted applications, potentially causing the affected companies to suffer revenue losses. Unfortunately, since the cloud provider lacks semantic knowledge of the hosted applications and their topologies, it cannot be trusted to recover them from failures, which may entail application-specific actions including restarting some of its components and reconfiguring others. Hence, part of the management burden still lies on the hosted application owners.

We advocate that the deployment of an application in an IaaS cloud must encompass a self-contained solution to properly recover the application from failures should the hosting cloud misbehave. Therefore, we advance the state-of-the-art by going beyond mere automation of the initial deployment and configuration; we propose AESON (Activation Engine on Service Overlay Network), a deployment runtime that automatically detects node (virtual machine) failures and eventually brings the application to the desired state by using a model describing the relationships between the application components. AESON relies on the application model to deploy the application as well as to recover it from failures that may happen during and after deployment. At AESON's core is a decentralized peer-to-peer publish/subscribe system leveraging IBM's Bulletin Board (BB) [12], a topic-based distributed shared memory service built on top of an overlay network.

AESON is a lightweight and flexible runtime with negligible performance overhead, capable of tolerating simultaneous failures. It can handle three types of failures: *node crash*, *node hang*, and *application component failures*. AESON guarantees correctness (execute actions in correct order) and eventual completeness (execute all actions) of the application deployment and recovery. We have experimentally verified AESON's completeness, correctness, and single and multiple concurrent recoveries, as well as evaluated its performance overhead.

The remainder of this paper is structured as follows. We introduce key concepts and formal definitions used throughout the paper (§ II), describe AESON's approach and architecture (§ III), discuss key properties of AESON (§ IV), evaluate its performance overhead (§ V), summarize the related work (§ VI), and conclude (§ VII).

II. Background

A. Composite Creation and Activation

We define a *composite* as a collection of inter-dependent virtual machines (VMs) on which a distributed application runs to provide a service. E.g., a composite for a Bulletin Board application, such as RUBBoS [10], could comprise one Apache server, two Tomcat servers, and one MySQL server.

A composite can be modeled through a virtual image construction tool [13] as follows (see Figure 1). First, the user chooses a base virtual image and customizes it by adding the software packages required by the application (we describe image creation in detail in [13]). In Figure 1, three virtual images (Apache, Tomcat, and MySQL) are created. The next step is to model the composite by specifying the images to use, the number of instances of each image, and the connections between the instances. In our sample, we depict a four-node composite model using the previously created images. The resulting composite model can later be used to deploy the composite, i.e., to instantiate and configure the VMs.

A composite model created by the above process contains operations (in the form of scripts) to be run at deployment time to configure and cross-configure the software components, as well as operations to be executed during failure recovery. The composite model captures all cross-dependencies between the defined operations, as exemplified in Figure 2.

We define *activation* as the fully automated process of starting the VMs for a given composite and running the deploy-time operations to configure the application components on those VMs so that the application can function properly. During the deployment of the composite modeled in Figure 1, one Apache VM, two Tomcat VMs, and one MySQL VM will be started. Next, the deploy-time operations executing on the VMs take care of, for instance: configuring Apache to act as the load balancer for the two Tomcat VMs; deploying the application logic to Tomcat; and configuring Tomcat to access the application database on the MySQL VM. In the paper, we refer to each individual operation performed on deployment as an *activation action*, and to each operation performed during recovery (when a failure occurs) as a *recovery action*.

B. Definitions and Terms

In this section we formally define some of the terms that are used in rest of the paper.

***Definition** 1:* Execution of an activation (or recovery) action is an atomic event that is either completed successfully in time t_{max} or fails in one of two ways: a system error cause it to terminate before t_{max}, or it is marked as a faulty execution after t_{max} (the user-specified value for the maximum execution time) has elapsed.

***Definition** 2: Dependency* $d = \{a_1, VM_1, a_2, VM_2\}$ is a quadruple, where action a_1 on VM_1 should be executed only after action a_2 on VM_2 is completed.

***Definition** 3: Sensitivity* $s = \{r_1, VM_1, r_2, VM_2, l\}$ is a quintuple, where recovery action r_1 on VM_1 should be executed if recovery action r_2 on VM_2 has been executed and local state of VM_1 has reached the level l.

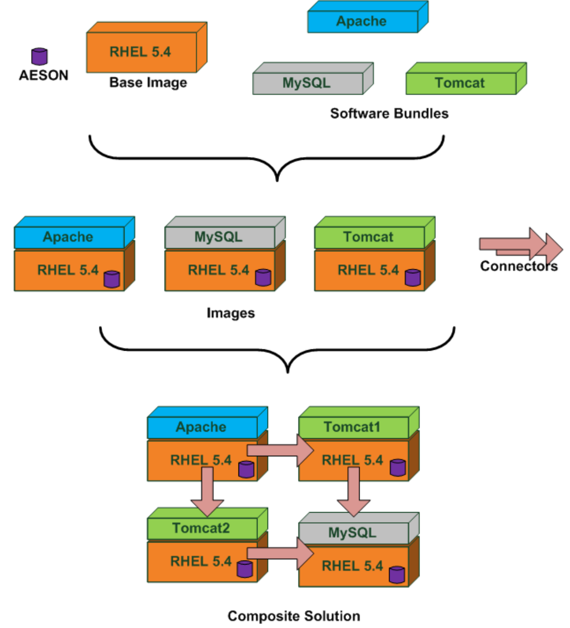


Fig. 1. The process of creating images and modeling a composite service.

***Definition** 4: Parameter propagation* $p = \{a_1, VM_1, p_1, a_2, VM_2, p_2\}$ is given by a sextuple, where action a_1 on VM_1 takes a set of input parameters p_1 from the set of output parameters p_2 produced by executing a_2 on VM_2.

***Definition** 5: Main recovery* refers to recovering a failed node by either restarting or recreating a virtual machine.

***Definition** 6: Secondary recovery* refers to the execution of recovery actions on non-failed nodes, which is driven by the specified sensitivity of the nodes.

To illustrate the concepts discussed above, Figure 2 shows a snippet of the configuration file of our sample composite's Tomcat server. It defines two activation actions, namely `deploy` and `configure`. The `deploy` action does not depend on any other activation actions; it deploys the RUBBoS Web application to the Tomcat container. In contrast, Tomcat's `configure` action depends on MySQL's `configure`, since the former consumes as an input parameter the output parameter (MySQL URL) produced by the latter. Finally, the configuration file defines a *secondary recovery* action `reconfigure`, stating that the Tomcat server is sensitive to MySQL's `restart` recovery action after Tomcat has reached the level `configure`. In other words, on failure, if the MySQL VM is restarted, the Tomcat VMs will run `reconfigure`, provided they have already run the `configure` operation.

III. AESON - Approach

AESON performs three key activities for a composite service: *activation*, *monitoring*, and *recovery*. It constantly monitors the status of the composite service during and after deployment. In the event of a node failure, AESON detects the failure, determines an appropriate recovery plan (a subset of the defined recovery actions), and executes the plan to bring the service back to normal operation. This is illustrated in Figure 3, where

```xml
<Activation>
  <VirtualSystem id="rubbos_tomcat">

    <ProductActivation class="deploy">
      <Program cmdLine="true"  href="AS/deploy.sh"/>
    </ProductActivation>

    <ProductActivation class="configure">
      <Properties>
        <Property key="mysqlurl"/>
      </Properties>
      <Program cmdLine="true"  href="AS/configure.sh"/>
       <ProductDependency class="configure" vsId="rubbos_mysql">
         <PropertyMapping sourceKey="mysqlurl" targetKey=" mysqlurl "/>
      </ProductDependency>
    </ProductActivation>

    <ProductActivation class="reconfigure">
       <Sensitivity sensitivity="restart" vsId="rubbos_mysql" level=" configure "/>
      <Program cmdLine="true" href="AS/configure.sh"/>
    </ProductActivation>
  </VirtualSystem>
</Activation>
```

Fig. 2. A snippet of the configuration file for the composite's Tomcat server.

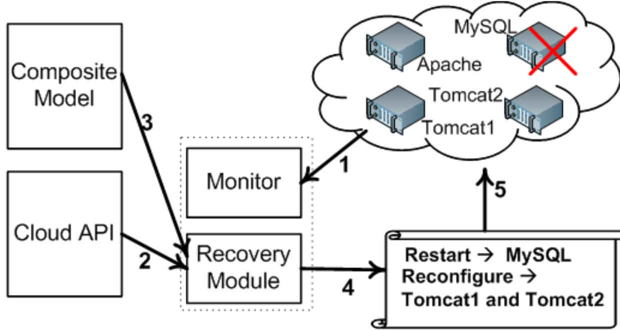


Fig. 3. AESON: monitoring and failure recovery.

the RUBBoS application is (being) deployed and a failure occurred in the MySQL server (1). The monitor detects the failure and notifies the recovery module; subsequently, the recovery module talks to the cloud API to execute the required *main recovery* action (restart or recreate the failed VM) (2). Next, by using the composite model, appropriate recovery actions are executed on non-failed nodes (in the RUBBoS case, Tomcat1 and Tomcat2 are reconfigured) (3). As we will discuss later in the paper, the executed recovery plan is created in a distributed manner by using the specified sensitivities (definition 3). After the recovery plan execution, the service will be back to normal operation (if failed after deployment), or to a clean state (if failed during deployment) from which AESON can take the service to the desired deployment state.

A. System Architecture

Our primary goal when designing AESON was to embed in a composite a management middleware responsible for recovering it from failures. To realize such *self-contained* management, we designed AESON as a peer-to-peer (P2P) system. In this design, the role of orchestrating deployment and recovery is *distributed* to all nodes; we leverage the multi-node nature of composites to make AESON itself fault-tolerant: it runs on each node, avoiding a single point of failure (see Figure 1). In contrast, a self-contained centralized orchestration approach would exhibit a single point of failure.

For AESON, each composite node is a management peer. This P2P design naturally captures the dynamic nature of cloud applications, where nodes (peers) can fail and join at any time. AESON leverages the publish/subscribe communication mechanism implemented by IBM's BBSON [12], which offers an abstraction that further facilitates our design. Since a publisher can publish even with no subscribers, the separation between communication and execution of activation/recovery actions is clean and elegant.

As shown in Figure 4, AESON consists of four main components:

Configuration
: Every action that AESON takes is based upon the composite model, which dictates what to do during both normal activation and recovery.

Communication
: Each peer communicates using the BBSON APIs for publishing and subscribing to *topics*. AESON maintains one topic per node for activation and recovery orchestration, plus two topics used for leader election, as explained later.

Failure and Recovery
: Constant status monitoring is needed to detect failures and recover from them. AESON uses a monitoring utility provided by BBSON.

VM Control
: Interactions with cloud APIs are needed to start, stop, create, re-create, and restart VMs.

AESON comprises *activation*, *recovery*, and *logging* modules. The activation module is responsible for tasks related to normal activation, such as: checking, validating, and enforcing dependencies; and execution of activation scripts and related parameter propagation. The recovery module is used during failure recovery. It checks the sensitivity of nodes and executes appropriate recovery actions, propagating parameters when recovery actions so require. The logging module logs all local ongoing activities and their statuses, so that each node that is restarted can know which of its actions it has completed and which ones it has yet to perform.

B. Normal Activation

Normal activation refers to the process of deploying and configuring a composite service in the absence of failures. At start-up, each node first joins the AESON group of the composite; only after all nodes have joined the group does normal activation start. It ends after all activation actions are completed successfully, at which point the composite will have reached the desired deployment state as specified by the composite model. In Section IV-A we highlight how normal activation eventually ends in AESON.

During normal activation, the nodes collaborate and coordinate with each other to reach the desired state. Each node

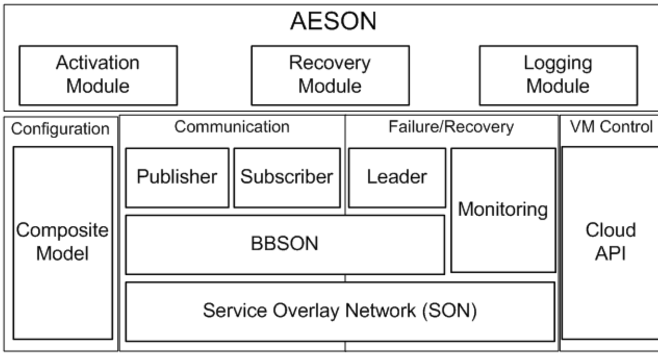


Fig. 4. AESON architecture.

performs the following activities when it executes a normal activation action. (1) It checks whether all dependencies are fulfilled; if not, it asks the status of the missing dependencies by posting on its own topic and waits until all dependencies are satisfied. (2) If all dependencies are satisfied, the node publishes on its topic indicating that it is going to start the action. (3) The node executes the script associated with the action by passing all the required input parameters. Those parameters may come from the model or from another script on which this action depends (e.g., database URL may come from database-config.sh). Next, the node extracts any output parameters (output produced by the script), and publishes the status of the execution along with the output parameters. If the execution completed as a failure, then it informs the group, which will terminate the activation process across all nodes.

AESON checks the cross-dependencies of actions to guarantee a correct execution order. If a dependency of an action to be run has not been satisfied yet, the dependent node publishes on its topic a status request. Once the affected node sees such a request, it updates the status of the requested action on its topic. Upon seeing a positive update, the node waiting for it can then run the dependent action.

C. Fault Model

AESON is designed to support three types of failures: node crash, i.e., it cannot be accessed through ICMP (ping) or through the cloud API; node hang, i.e., it cannot be accessed through ICMP but can be accessed through the cloud API; and application component failure, i.e., the node can be accessed through both ICMP and Cloud API, but an application component running on the node has either crashed or hung. Node crashes and hangs can be addressed by using two levels of application-independent monitoring; in contrast, to identify application component failures we need application-specific monitoring. The current prototype is implemented to monitor node crashes and hangs only; application-specific monitoring will be addressed in future work.

We have leveraged BBSON to realize our fault model. Each node constantly keeps sending and receiving heartbeats. In this model, a node is assumed to be failed if other nodes do not receive heartbeats for a specified time period. Thus, when a VM hangs or crashes, AESON detects it through BBSON. Subsequently, AESON waits for the specified time to see whether the failure is intermittent or permanent; if it is permanent, AESON uses the cloud API to take the required main recovery action and trigger the recovery plan. In our fault model we use two types of operations (main recovery) to recover a failed VM: **restart** and **create**. If the VM cannot be accessed through ICMP but can be accessed through the cloud API, then the resolution is to restart it; otherwise, AESON creates a new VM. Also, if the restart resolution does not solve the problem, AESON creates a new VM.

In a restarted VM, the recovery module runs only the activation actions that have not been completed. It uses the logging module to find the completed actions. In contrast, in a newly created VM, all normal activation actions will run. Notably, doing either restart or create on the failed node has consequences on non-failed nodes as well. For example, if a VM that runs the database is recreated, then the application server may need to restablish its database connection. Hence, non-failed nodes may also need to perform some actions as part of the recovery. In the next subsection we will detail the recovery process on non-failed nodes.

D. Recovery

The recovery process is divided into two stages: ***main recovery*** (recovering the failed node, e.g., MySQL server in Fig 3) and ***secondary recovery*** (recovering the non-failed nodes, e.g., Tomcat1 and Tomcat2 in Fig 3). In main recovery, AESON decides the appropriate recovery action (i.e., restart or create) by retrieving the status of the failed node via the cloud API. In contrast, secondary recovery is solely driven by the composite model. Given the model and the failure, the nodes will be able to coordinate to execute the underlying recovery plan.

The execution of secondary recovery takes place in a distributed and dynamic manner. Concretely, each node decides the next needed recovery action based on the actions performed by other nodes, its current local state, and its *sensitivities* defined in the model. In our example of Figure 3, assume a recovery action is "restart the MySQL server." If the application server has not deployed the Web application yet, then no recovery action is needed. However, if the application server has already created database connections, then it needs to restablish them. In other words, the application server needs to perform a recovery action due to the database failure.

When a node failure is detected during deployment, normal activation is paused on all nodes until the recovery process is completed. However, if a node is in the middle of a normal activation action, it completes the ongoing action and then abstains from executing any further activation actions until the recovery has been completed.

During recovery, AESON has the notion of an elected "leader" (see Algorithm 1). A leader is needed mainly to prevent two nodes from initiating the main recovery action (i.e., restarting or recreating the failed VM). The leader eventually finds the node failure and initiates the main recovery action. Once the newly created or restarted VM is active, it automatically joins the AESON group. If the VM is restarted,

it already has the IP address of all other nodes; thus, it uses one of those as the bootstrap node[1]. In contrast, if the VM is recreated then a bootstrap node IP address is passed as a parameter to the VM.

When the failed node rejoins, the leader informs the group indicating that it has successfully completed the main recovery. Consequently, if a node has sensitivity to the main recovery action, then it will run the recovery action on that node, which drives recovery action(s) on other nodes based on their sensitivities. The leader notifies the group when all sensitivity-driven recovery actions have been executed. Once the recovery is completed, normal activation is resumed (if the failure happened during deployment). One should note that the newly rejoined node performs normal activation actions and does not perform any recovery actions.

At any given time AESON can handle two types of recoveries: single recovery (i.e, only one ongoing recovery) and multiple recoveries due to multiple failures, including failures that happen during ongoing recoveries.

1) Single Recovery: In this case, the leader eventually finds the node failure and initiates the main recovery action. Importantly, if the leader happens to be the failed node, then another member of the group is elected the new leader; the newly elected leader initiates the main recovery action. Once the main recovery action is completed, a chain reaction starts, where the execution of one recovery action may trigger recovery actions on other nodes. Once all secondary recovery actions are completed, the leader marks recovery as finished and notifies the group. Upon receiving the recovery-completed notification, each node resumes normal activation, if the failure happened during deployment, or else the system is ready.

2) Multiple Recoveries: It is conceivable that more than one node can fail simultaneously, or that a node can fail during the recovery of another failure. Regardless of the way in which nodes fail, in AESON, secondary recovery is performed only when all nodes are connected in the group. Hence, the first task is to initiate the main recovery actions for the failed nodes. Once all nodes are recovered (rejoined the group), AESON resumes/starts the secondary recoveries, from the latest sequence to the first. In effect, multiple failures become a series of single recoveries, where each single recovery corresponds to a separate recovery plan.

A major challenge in handling multiple recoveries is the leader failure in the middle of ongoing recoveries. For example, assume multiple nodes failed and for some (not all) of them the leader called the cloud API to start the main recovery; later, the leader crashes. As mentioned earlier, in the event of the leader failure, a new leader will be elected using the leader election algorithm (see Algorithm 1); however, the new leader needs to start from where the previous leader has left off. This is possible with our approach because each peer in the system has the same view on the current system state. Thus, when the leader fails, the new leader can take over and guide the recovery of the system.

[1]If a new node wants to join a P2P system, then it is required to know the address of at least one peer.

Algorithm 1: Sketch of AESON's Leader Election

- Step1: Subscribe to the `Election` topic and publish process ID to the Election topic.
- Step2: Subscribe to the `LeaderInfo` topic.
- Step3: Wait until the membership of the leader is posted in `LeaderInfo` by a single process, or a timeout expires.
- Step4: If the timeout expires before the leader info is available in `LeaderInfo`, elect a new leader using the `Election` topic:
  - Select the preferred process with minimum ID among the publishers to the `Election` topic as the current leader.
  - If not selected as a leader, go to Step3.
- Step5: When there are more than one leader published in the `LeaderInfo` and I am not the leader with the minimal ID, then unpublished myself from `LeaderInfo`.
- Step6: When the leader process fails (its record disappears from the LeaderInfo topic):
  - If I am the node with the preferred node with the minimal ID published in the `Election` topic and post myself in the `LeaderInfo`.
  - If not selected as a leader, go to Step3

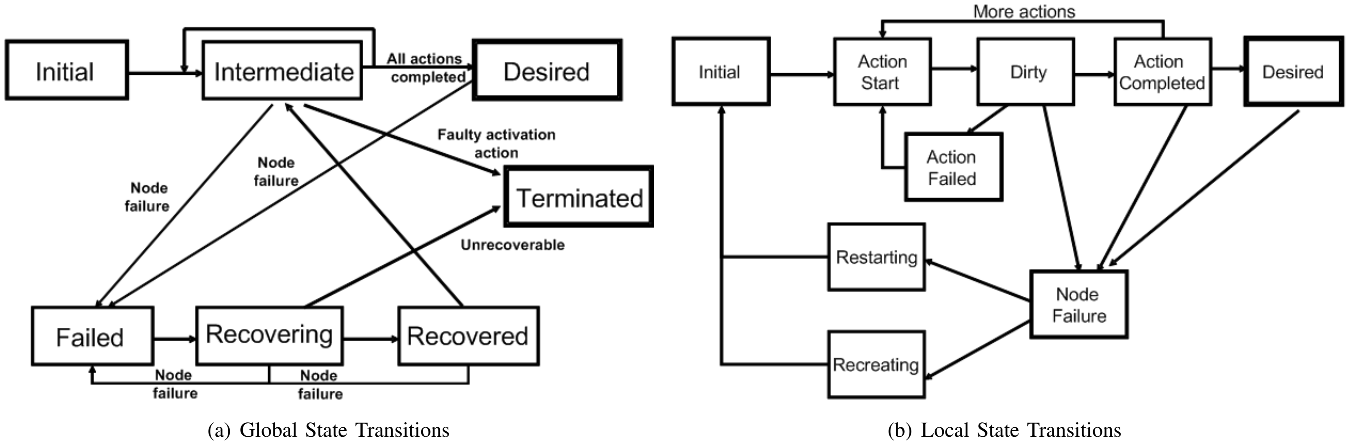
IV. System States and Properties

In this section, we briefly outline AESON's state transitions that happen during both normal activation and recovery. We also present key properties that AESON supports.

Figures 5(a) and (b) represent, respectively, AESON's global state transitions and local state transitions. The global state is an aggregated view of the local states of all nodes. For the reader to better understand the figures, we define below some of the terms used:

- **Initial**: For the global case, it means that all nodes in the composite have joined the group of peers but none has started normal activation. For the local case, the node in question has joined the group.
- **Desired**: It refers to the completion of normal activation and/or recovery.
- **Intermediate**: It is a global state between *initial* and *desired*.
- **Terminated**: It is the state resulting from AESON's aborting its operation due to an unrecoverable failure or a permanent execution error exhibited by an activation or recovery action.
- **Dirty**: It refers to the time during which an activation or recovery action is being executed.
- **Failed**: AESON has detected a failure, but the corresponding recovery plan has not started yet.
- **Recovering**: AESON is in the middle of a recovery plan execution.

In a nutshell, the local state transition graph shows the states each node can be in while AESON performs its activities for activation and recovery, whereas the global state transition graph summarizes the holistic view of the system, based on all local states.



(a) Global State Transitions

(b) Local State Transitions

Fig. 5. State transition diagram for global and local states.

A. System Properties

We now outline important properties supported by AESON, some of which are directly inherited from BBSON [12].

Dependency Preservation: Let *X* and *Y* be activation actions on two VMs, VM_1 and VM_2, respectively. If *Y* depends on *X*, then *X* finishes on VM_1 before *Y* starts running on VM_2. AESON supports this because of two BBSON properties: eventual inclusion and correctness.

Timeout Enforcement: Let the user-specified maximum time for an activation or recovery action *X* be t_{max}. *X*'s execution will be either successfully completed in time *t* (where $t \leq t_{max}$), or marked as faulty otherwise.

Sensitivity Enforcement: Let *P* and *Q* be two recovery actions on two different VMs. If *P* is sensitive to *Q*, then if *Q* is executed, *P* will be executed. *P's* execution will start after *Q's* is completed.

State-Change Enforcement: Let VM_1 and VM_2 be two different VMs. Suppose that VM_1 performs an action *P*; as a result, it will publish *P's* execution completion notification on its own topic. This notification will eventually reach VM_2, since all VMs subscribe to the each other's topics and BBSON guarantees eventual consistency. Furthermore, in the case of recovery, AESON implements a two-phase protocol to ensure that no state change is lost (see Algorithm 2). With BBSON, given two consecutive posts to a topic, the subscribers may only see the last one. During the execution of recovery plans, AESON's implementation relies on not missing any posts.

Based on the above properties and those guaranteed by BBSON, we can make the following claims.

B. Claim 1: Eventual Activation Completeness

Under the assumption that each provided activation action behaves normally and finishes its execution within time t_{max}, AESON guarantees that all activation actions of the composite will complete.

Eventual activation completeness, as herein defined, comes as a corollary of *state-change enforcement*. Since all nodes will be eventually notified of the execution progress of each activation action, all actions will eventually complete, even the ones that depend on the execution of others.

Algorithm 2: Two Phase Protocol for State Enforcement

- Step1: When a node N_i wants to enforce the state transition for action a_i N_i posts a state update message on N_i's topic.
- Step2: All the connected nodes respond by posting $ACKs$ on their topics.
- Step3: If N_i unable to receives $ACKs$ from all the connected node, then it waits time t and reposts the state update.

C. Claim 2: Correctness

As a corollary of *dependency preservation* and *sensitivity enforcement*, AESON guarantees that activation and recovery actions will be executed in a correct order, as defined in the composite model.

D. Claim 3: Eventual Single-Recovery Completeness

In the case of a single-node failure, AESON guarantees that the composite will be restored to the desired state, under the following assumptions: (1) each provided recovery action behaves normally and finishes its execution within time t_{max}; and (2) the main recovery action (VM restart or create) fixes the observed node failure.

Given BBSON's eventual exclusion property, i.e., when a peer leaves the group all nodes eventually notice it, all nodes will eventually perceive the node failure. Because of *state-change enforcement*, all nodes will be notified of the initiation and completion of the main recovery action. Similarly, all nodes will be notified of the execution progress of each secondary recovery action. Therefore, all recovery actions will eventually complete, regardless of their sensitivities (dependencies).

E. Claim 4: Eventual Multiple-Recovery Completeness

When facing simultaneous failures, or failures that happen during an ongoing recovery plan, AESON guarantees that the composite will be restored to the desired state under the following assumptions: (1) each provided recovery action

behaves normally and finishes its execution within time t_{max}; (2) the main recovery action (VM restart or create) fixes the observed node failure; and (3) at least one of the nodes does not fail.

Each failure is eventually perceived due to BBSON's eventual exclusion property. Because of *state-change enforcement*, all nodes will be notified of the initiation and completion of each main recovery action. Similarly, all nodes will be notified of the execution progress of each secondary recovery action, of all recovery plans. Therefore, all recovery actions will eventually complete, regardless of their sensitivities (dependencies).

V. Performance Evaluation

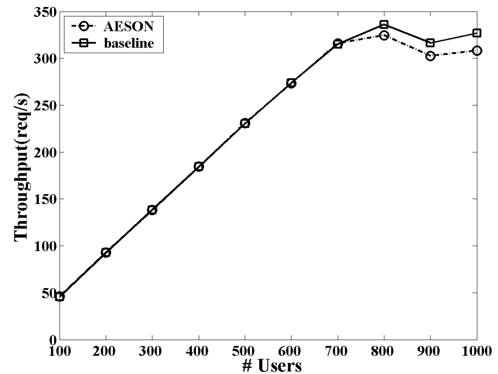
We have experimentally verified AESON's completeness, correctness, and fault tolerance. However, due to the associated complexity of graphical representation, we have limited our discussion only to the performance overhead of AESON, which follows.

In order to assess AESON's performance overhead, we modeled and deployed a composite for the RUBBoS Bulletin Board benchmark application [10]. Our composite comprises one Apache server, two Tomcat servers, and one MySQL server. Each individual application component is assigned to a dedicated host and each host has a running instance of AESON. As a baseline for comparison, we also manually deployed the RUBBoS composite, running it without AESON. We varied the RUBBoS workload intensity by generating requests originating from 100 to 1000 users. We measured the client-perceived performance overhead (response time and throughput) as well as the resource utilization at each server.

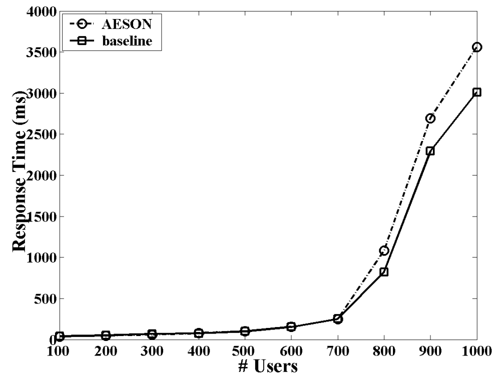
Each data point shown in the graphs of Figures 6 and 7 represents the average of 3 experiment runs. The observed average throughput and response time values are shown in Figure 6(a) and Figure 6(b) respectively. As shown in Figure 6, AESON shows a performance similar to the baseline case up to 700 users. From that point on, AESON shows an average of 6% performance drop.

We also analyzed the CPU utilization in the deployed servers (see Figure 7(a)) and found that the observed performance degradation (for both baseline and AESON) is caused by a high CPU utilization at the Tomcat servers. For low CPU utilization, both AESON and the baseline have a negligible difference in throughput and response time; when the system is highly utilized, AESON causes a small performance overhead (6%).

AESON is designed as a P2P system and at runtime each peer communicates by sending application messages, which causes network overhead. Thus, to understand the overhead we measured he number of network bytes sent and received at each server. The measured results for network are shown in Figure 7(b) and Figure 7(c). The network overhead introduced by AESON is negligible. Notably, after 800 users, the baseline approach exhibits higher network traffic than AESON. This is due to the throughput drop we observed in Figure 6(a). Since the throughput is higher for the baseline case, it processes more requests; hence, more data packets.



(a) Throughput



(b) Response Time

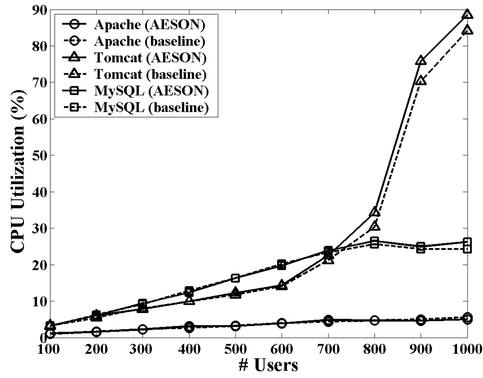
Fig. 6. Performance overhead of AESON (Throughput and Response Time Comparison).

In summary, in our experiments, AESON showed a small performance overhead.

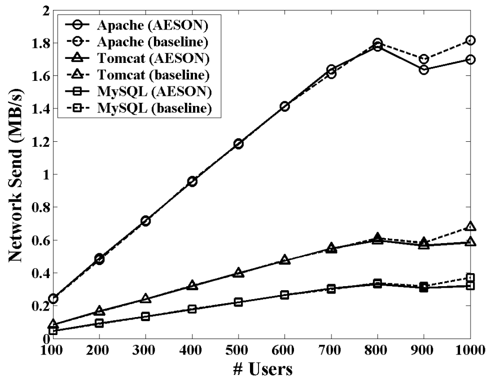
VI. Related Work

The issue of detecting failures and doing something to fix them has been around as computers, for instance, this was first discussed in the concept of system diagnosis [2]. Some approaches have focused on automatic restarting of components before or after they have failed [3], [4]. There have also been some works in applying techniques from Markov decision theory to dependability problems [5]. Yet, most studies try to recover a service after deployment and focus on the mechanisms by which recovery can be automated and made more efficient, rather than determining when and where the recovery actions should be applied.

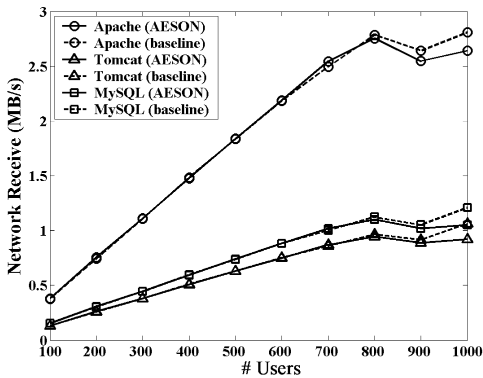
Yuanshun *et al*. studied the self-healing function from the consequence-oriented point of view [7]. To fulfill the self healing requirements of efficiency, accuracy, and learning ability, a hybrid tool that takes advantages from Multivariate Decision Diagram and Naïve Bayes Classifier is proposed. Kaustubh *et al*. combined monitoring and recovery to realize the benefits that cloud not have been obtained by using them in isolation, they also presented two algorithms and evaluated those using fault injections [6]. A fine-grained technique for surgically recovering faulty application components, without disturbing



(a) CPU Utilization



(b) Network Traffic (Send)



(c) Network Traffic (Receive)

Fig. 7. Performance overhead of AESON (Comparison of CPU Utilization and Network Traffic).

the rest of the application is presented in [8]. Arun *et al*. proposed operating system virtualization techniques to provide automatic and transparent mechanism for proactive FT for arbitrary MPI applications [9].

A failure during a failure recovery process poses a tough problem because of the complexities involved, importantly, AESON is designed to address this problem. Previously, Naveed *et al* have also proposed an approach using the dependency model of the system to address a similar problem [14]. One of the biggest advantages of ours compared to most of the approaches discussed above is we have designed AESON as a decenterlized P2P system to prevent the single point of failure. In addition, AESON does not make any assumptions about the application; second, it supports three types of faults; and third, AESON supports fault detection and recovery during both deployment and after deployment.

VII. Conclusion

We presented AESON, a model driven fault tolerant composite service activation runtime to address dependability issues of enterprise applications that are deployed (and being deployed) in IaaS clouds. AESON uses a composite model to automatically heal from virtual machine failures during both deployment time and after deployment. To prevent the single point of failure, AESON is designed as a P2P pub/sub system by leveraging IBM Bulletin Board (a topic-based distributed shared memory service built on top of an overlay network). We experimentally evaluated the correctness, completeness, and ability to recover from single and multiple virtual machine failures. We analyzed AESON's performance overhead showing a negligible impact.

VIII. Acknowledgments

We would like to thank *Vita Bortnikov*, *Mike Spreitzer*, and *Alexey Roytman* (IBM Research) for their help with BBSON. We are indebted to *Vita* also for the leader election algorithm and for its integration with BBSON.

References

[1] T. Eilam, M. Elder, A. Konstantinou, E. Snible. Pattern-based Composite Application Deployment. In *IM 2011*.

[2] F. Preparata, G. Metze, and R. Chien. On the connection assignment problem of diagnosable systems. In *IEEE Trans. on Electronic Comp.*, EC-16(6):848-854, Dec 1967.

[3] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Gang, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *DSN 2002*.

[4] D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D. Patterson, and K. Yelick. Roc-1: Hardware support for recovery-oriented computing. In *IEEE Trans. On Computers, Feb 2002.*

[5] H. de Meer and K. S. Trivedi. Guarded repair of dependable systems. In *Theoretical Comp. Sci., 128:179-210, 1994.*

[6] K. Joshi, M. Hiltunen, W. Sanders, R. Schlichting. Automatic Model-Driven Recovery in Distributed Systems. In *SRDS 2005.*

[7] Y .Dai, Y. Xiang, G. Zhang. Self-healing and Hybrid Diagnosis in Cloud Computing. In *CloudCom 2009.*

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox. Microreboot - A Technique for Cheap Recovery. In *OSDI 2004.*

[9] A. Nagarajan, F. Mueller, C. Engelmann, S. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *ICS 2007.*

[10] RUBBoS: Bulletin board benchmark. jmob.objectweb.org/rubbos.html.

[11] F. Preparata, G. Metze, and R. Chien. On the connection assignment problem of diagnosable systems. In *IEEE Trans. on Electronic Comp., EC-16(6):848854, Dec 1967.*

[12] V. Bortnikov, G. Chockler, A. Roytman, M. Spreitzer. Bulletin Board: A Scalable and Robust Eventually Consistent Shared Memory over a Peer-to-Peer Overlay. In *LADIS 2009.*

[13] F. Oliveira, T. Eilam, M. Kalantar, F. Rosenberg. Semantically-Rich Composition of Virtual Images. In *IEEE Cloud 2012.*

[14] N. Arshad, D. Heimbigner, A. Wolf. Dealing with failures during failure recovery of distributed systems. In *2005 workshop on Design and evolution of autonomic application software*

[15] Amazon Web Services. CloudFormation. http://aws.amazon.com/cloudformation/.