Providing Resiliency to Orchestration and Automation Engines in Hybrid Cloud

Long Wang, Harigovind V Ramasamy, Alexei Karve, Richard E Harper
IBM T. J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598
{wanglo, hvramasa, karve, reharper}@us.ibm.com

***Abstract*—Hybrid cloud environments have seen a rapid rise in recent years. An essential part of a hybrid cloud is its ability to orchestrate the allocation, provisioning, and management of different compute resources spanning multiple cloud systems, and drive these operations across multiple cloud systems in an automated way. The Orchestration and Automation Engines (OAEs) of a hybrid cloud must themselves be highly available for ensuring high resiliency of the hybrid cloud. We present our experience in providing resiliency to the OAEs of a real-world hybrid cloud in this paper. The presentation includes the resiliency architecture of the OAEs, solutions that deal with errors ranging from software component crash to configuration/metadata error and data corruption, experimental results and our lessons learned from the practical experience.**

I. INTRODUCTION

Cloud environments play an increasingly important role in hosting IT services thanks to the convenient resource allocation and the ease of provisioning and deployment of machines and software. Hybrid cloud environments have seen a rapid rise in recent years for multiple reasons. Users may still want to preserve their most confidential data in their own on-premises data centers rather than store all of their data on a public cloud. Users may also want to use services from multiple cloud systems to leverage different features of these clouds. For example, cloud system A may provide better protection of user data and higher application availability but charge higher prices, while cloud system B may provide lower protection and availability levels with correspondingly lower prices. Then the users may want to place critical applications in cloud system A, and then place other applications in cloud system B. An in-depth introduction on hybrid cloud can be found in [1].

A critical success factor for a hybrid cloud is to enable resiliency [2]. That includes both the resiliency of the user workloads and that of the hybrid cloud platform itself. An essential part of a hybrid cloud, compared with a pure public cloud or private cloud, is that a hybrid cloud needs to orchestrate the allocation, provisioning, and management of different compute resources spanning multiple cloud systems, and drive the operations across these multiple cloud systems in an automated way. A hybrid cloud may provide a convenient means for users or administrators to manipulate their resources or workloads across multiple cloud systems from a unified GUI interface. Due to their vast number of touchpoints and control points, it is critical that the Orchestration and Automation Engines (hereafter called OAE) of the hybrid cloud are themselves highly available for achieving high level of resiliency for the overall hybrid cloud.

In this paper, we present our experiences in providing resiliency to the OAEs of a real-world hybrid cloud. Existing papers or technical articles related to hybrid cloud resiliency fall into the following categories: a) high-level description of the resiliency challenges in hybrid cloud, including taxonomy of cloud systems [4], simple lists of resiliency challenges in hybrid cloud [7], and in-depth description of hybrid cloud (but resiliency of orchestration and automation is not discussed) [1]; b) approaches for providing resiliency of customer workloads on hybrid cloud, especially against storage failures through the replication of customer workloads across different cloud systems/vendors [3][5]; c) approaches for providing resiliency of a specific cloud vendor's cloud environment, such as VMware's high-level resiliency solutions [6] and Microsoft Azure's high-level resiliency solutions [7]; d) commercial and open-source cloud orchestration and automation tools, such as VMware vRealize Orchestrator and Chef, have clustering capabilities for resiliency [9][8].

There hasn't been a comprehensive analysis and detailed treatment of the resiliency of OAEs in the context of a real-world hybrid cloud. We believe our work is the first one to address this shortcoming.

The main contribution of this paper is to present an error model and a detailed solution for providing resiliency to the OAEs of a real-world hybrid cloud. We provide a detailed architectural description of the hybrid cloud's resiliency-enhanced OAEs, detailed analysis of the types of errors encountered, and describe how the resiliency-enhanced architecture protects against or tolerates those errors. We also report our experimental results and learned lessons from this practical work. Experimental results demonstrate the effectiveness of the resiliency solutions and the performance of the recovery that conforms to the Service Level Agreement (SLA).

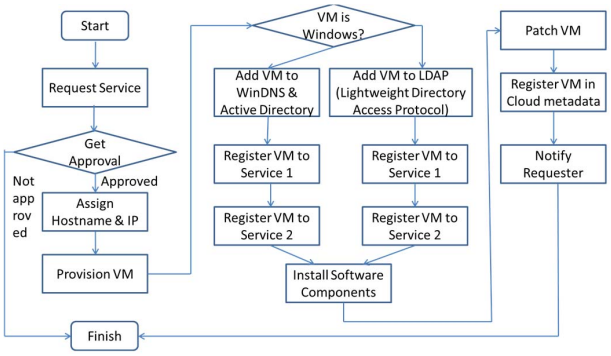

Figure 1: An Example Orchestration Workflow

Background. In the hybrid cloud domain, orchestration and automation are two distinct concepts with overlapping semantics. Orchestration usually refers to the integration of element operations for achieving a business process. Each element

2325-6664/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN-W.2017.35

operation typically interacts with humans or different subsystems. Through orchestration, these steps are integrated into a workflow. Figure 1 shows an example orchestration workflow for provisioning a virtual machine (VM). Simple workflow steps can be implemented in the orchestration engine itself; however, complicated steps (e.g., "Provision VM" in Figure 1) may require specialized automation engine. For example, one or multiple blueprints are usually programmed to specify how a VM should be provisioned in different cloud platforms. The blueprints are usually executed by the specialized automation engine.

II. Error Model

The OAEs are subject to many types of errors. Table 1 summarizes the error model, i.e., types of errors we focus on, and gives potential causes (underlying faults) and examples for them. The error model was motivated by a complete classical failure hierarchy "fail-stop, crash, omission, timing, incorrect computation, authenticated Byzantine, Byzantine" in the book [14]. Then we carefully pruned those categories not commonly encountered in practical cloud (e.g., "Byzantine") and substituted general categories with cloud-specific ones (e.g., "configuration/metadata error" and "failed workflow steps" substitute "incorrect computation").

The error model covers crash errors, timing errors, performance errors, and errors due to misconfiguration and data corruption. While our hybrid cloud is regularly patched to mitigate the risk of security vulnerabilities, we do not consider errors resulting from exploitation of security vulnerabilities or malicious attacks within the scope of this paper.

Note that error propagation may result in a "chain of threats" [11]. For example, a crash of a software component during execution of an orchestration workflow or an automation blueprint may result in configuration/metadata error as the state after the crash may be inconsistent, and/or result in a failed step of the workflow/blueprint.

III. Resiliency-Enhanced Orchestration and Automation

A real-world production implementation is presented in this paper. Our presentation focuses on the deployment of the implementation on a 2-node cluster because (1) this paper targets management stack failures, and (2) we conduct evaluation on the deployment as potentially disruptive experiments on the production environment require involving clients (e.g., notifying clients on crashing their applications). The management stack of a hybrid cloud usually adopts 2-replica or 3-replica solution (we select 2-replica in production for cost); the many workload nodes do not affect management stack. To simplify presentation while retaining generality, we present the 2-replica solution on the 2-node cluster.

A. Architecture

Figure 2 illustrates a simplified conceptual architecture of the OAEs in our hybrid cloud. There are two sets of OAE stacks, one based on IBM Business Process Manager (BPM) and OpenStack Heat, and the other based on VMware vRealize Orchestrator (VRO) and VMware vRealize Automation (VRA). The two sets of engines provide the convenience of integrating with multiple cloud environments from different cloud vendors. Moreover, from a resiliency perspective, use of the two sets of OAEs also provides diversity to decrease the chance that the two stacks expose the same faults, thereby increasing failure independence.

Table 1: Error Model for OAEs of Hybrid Cloud

Error Type	Potential Causes (Underlying Faults)	Example
Software Component Crash (process crash)	A process does not handle certain cases (e.g., error conditions) or parameters, and crashes.	A plugin of the Heat Engine cannot handle "404 not found" or "307 Temporary redirect", which causes the Heat Engine to crash.
Container, Machine (physical/ VM) Crash	Hardware failure, power outage, resource capacity problem.	When there is an "out of memory" error in the container or VM, the Heat Engine may get killed by resource management.
Performance Degradation	Overloaded conditions on a service delays response, problems in caching service or auto-scaling service.	An asynchronous task requires multiple retries to retrieve status due to an overloaded condition.
Configuration or Metadata Error	Administrator's mistake, incorrect parameters from user, configurations not updated correctly in certain operations.	Administrator sets wrong parameters. Images are replaced with newer images on the cloud but the configurations are not updated as required. Firewall rules may be too restrictive or overly relaxed.
Failed Steps of Workflow, Blueprint or Recipe	Error in workflow design or implementation, any crash, configuration error, timeout expires in performance degradation.	The workflow implementation has a timing error trying to ssh into a VM before boot completes. Dependencies across resources (IP, VM, etc.) are not properly handled in the workflow design.
Data Corruption	Global variables not synchronized properly due to incorrectly written code that does not lock properly. Disk failure.	In a rare concurrency scenario that is not properly handled by the code, mutual exclusion is not done correctly and data gets overwritten. This can result in unexpected behavior.

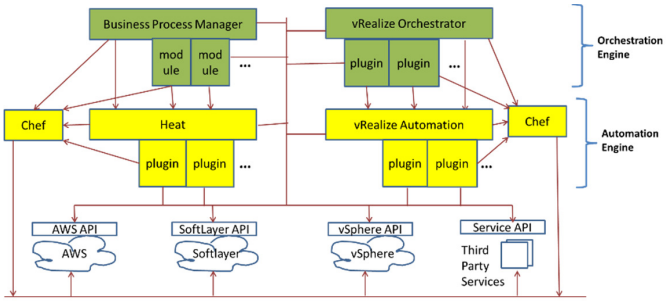

Figure 2: Architecture of OAEs in Our Hybrid Cloud

We set up a two-instance active-passive (hot standby) topology for each component in Figure 2 (so we have two BPM instances, two VRO instances, and so on): both instances are up but only one (the primary server) processes requests while the other (the hot standby server) stays idle. Figure 3 depicts the resiliency-hardened deployment for each of these components. We enforce *anti-collocation* constraints by placing the primary and hot standby components on different physical machines.

A pair of load balancers is set up to forward requests to the component instances. Both load balancers are configured to send requests to the primary component only. So we tolerate failure of any one load balancer with minimum footprint: no need to initiate workload failover as the primary is still working. The requests are sent to the backup component only when the primary component fails. The *HeartBeat (HB) & Recover* component

detects the availability of each of the four objects in Figure 3 through the Simple Network Management Protocol (SNMP) or request pinging (i.e., sending a request to a given port to test if there is a response within a time threshold). The *HB & Recover* component and a small agent, *HB & Recover Monitor,* mutually monitor each other's health and launch a new component if the monitored one is deemed to have failed. Virtual IP (VIP) service, e.g., the AWS floating IP service, is used over the two load balancers so that the failure of either load balancer does not affect the service: the user can still send requests to the virtual IP, i.e., the public IP of the service, when one load balancer crashes.

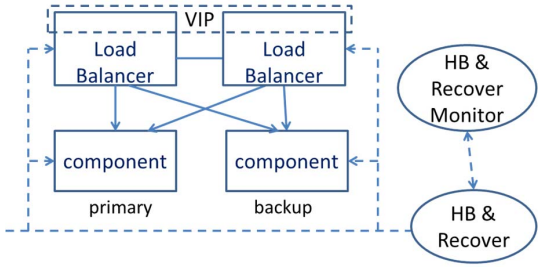

Figure 3: Resiliency-Hardening Topology for a Component

B. Handling Errors

All error types listed in Table 1 are handled in our hybrid Cloud. Due to the page limit, we describe handlings of *software component crash* and *failed steps of workflow* in the table here, and then give a summary of how all other types of failures in Table 1 are handled.

1) Software Component Crash

The software components of the OAEs (BPM, Heat, VRO, VRA and the Chef server) consist of several sub-components and plugins/modules which are stateful and interact with each other. The principles and design details for crash detection and recovery for these components are similar; here we just discuss the OpenStack Heat as an example due to page limitations.

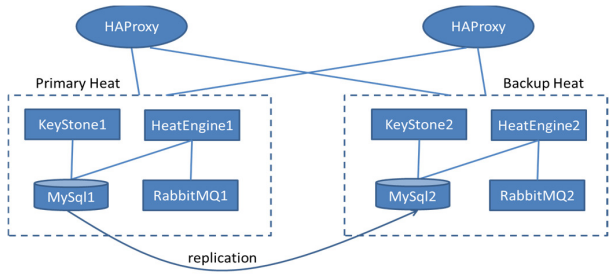

Figure 4: Resiliency-Enhanced OpenStack Heat

Figure 4 shows the resiliency-hardening topology for the sub-components of the Heat component. OpenStack Keystone places all its states in the MySql database. Rabbit MQ stores the tasks and intermediate results of the tasks in its queues. Heat Engine schedules and executes tasks of automation blueprints, places the output and certain states in MySql, and maintains the intermediate states of the blueprint execution in its memory. The HAProxy is the load balancer software depicted in Figure 3.

When a crash occurs in any of the four sub-components in the primary Heat engine, the *HB & Recover* component detects the crash, brings down all the sub-components of the primary Heat (e.g., brings down the VMs or containers that host these sub-components), and performs the failover to the backup Heat. All new requests are then processed in the backup Heat. The output of the successful execution previously performed by the crashed Heat must be persisted in the MySql for the backup Heat (MySql2 in Figure 4). During error-free execution, database replication techniques (specifically, MySQL master-slave configuration [10]) replicate the state of MySql1 to MySql2.

Requests in-process when the crash occurs are not switched over to the backup Heat for processing due to the difficulty in recovering the memory state of the crashed Heat engine while preserving the backup Heat engine's channels with the MySql and RabbitMQ. Instead, the handlings described in the "Failed Steps of Workflow, Blueprint or Recipe" sub-section below are applied. After all the handlings, another set of the Heat sub-components are set up as the new backup Heat. If the backup Heat, rather than the primary Heat, fails, *HB & Recover* detects the failure and creates a new backup Heat with the primary-backup configuration properly set.

2) Failed Steps of Workflow, Blueprint or Recipe

From the point of view of business processes, most errors (e.g., configuration errors) eventually result in failures of workflow steps because the automation blueprint and Chef recipe execution are part of a workflow execution. The first approach to handle workflow step failures is the use of workflow failure handler. Both IBM BPM and VMware VRO tools allow workflow designers to provide code in the workflow design to handle different kinds of step failures. The designers can write specific failure handling code for a particular failure, just like an exception handler in Java.

If a workflow failure handler is not provided by the designer, a default handler can be applied which releases those resources allocated during the failed workflow execution and makes proper status updates in certain metadata or database to indicate the failure of the workflow and remove any state inconsistency. Then the workflow is restarted for a retry. In cases when the workflow designer or an administrator specifies that the workflow should not be retried upon failure, retry is not attempted.

3) Other failures

When a container, VM or PM hosting the OAE components has a crash, the failover principles used for recovering from software component failures described above are applied. The *HB & Recover* component monitors vital statistics of each sub-component all the time to handle performance degradation. If the metrics such as throughput, request processing latency, are worse than a pre-specified threshold, the relevant software component is considered as problematic and is killed. Subsequently, the recovery procedure for software component crash is performed. Corruption of data rather than configurations is addressed using backup mechanisms (e.g., MySql backup). We handle configuration errors by using i) Configuration Compliance Checking (see the tool Service Activation and Deactivation [12]), ii) Configuration Versioning, or iii) Configuration Checkpoint.

IV. Experimental Results & Lessons

In our evaluation, a request automatically provisions VMs on Bluemix, AWS EC2 and VMware clouds, launches DayTrader applications on the VMs, makes configuration and sets up monitoring with NewRelic service [13]. We ran 100 fault injection experiments for each software component of the OAEs. 100 experiments are sufficient for measuring the effectiveness of

the error mitigation. In half of these experiments, we kill component processes inside containers; in the other half of the experiments, we kill the containers directly. We verified that in every experiment, failover is conducted successfully and that post-failover, new arrival requests are processed normally.

The detection and recovery time should be small enough to accommodate stringent Recovery Time Objective that specifies the targeted duration of time within which a business process must be restored after a failure. We conduct same fault injection as above and measure the time between when the process/container is killed and when the *HB & Recover* component completes the failover (this time is available from the log). As the measurements are done across multiple machines, we perform a synchronization of Network Time Protocol (NTP) on all these machines before conducting each experiment.

Table 2: Detection & Recovery Time for Heat (seconds)

Failure Injection	Mean Value	95% Confidence Interval
Kill the process of the Heat Engine sub-component	4.2	±0.56
Kill the container hosting the Heat component	10.6	±0.68

We ran 100 experiments for measuring the time of recovering each component. Table 2 lists the mean value and the 95-percentile confidence interval for the Heat component (the measurements for other components are similar). We can see that the detection and recovery time for the container failures is much longer than the time for the process failures. This is because the operating system underlying the process handles a process failure by immediately terminating all connections and closing all listening ports associated with the failed process. But the connections or listening ports of the failed container are not explicitly terminated after the container failure. It is detected only after the timeout expires (timeout is set to 5 seconds).

Recovery Point Objective (RPO) refers to the point in time up to which data should be restored after a failure. Our RPO requirement (as dictated by our hybrid cloud offering Service Level Agreements) is 1 minute. We measure the database replication lag to check if the RPO requirement is satisfied.

We create a table with the schema *(id int not null auto_increment primary key, created_at TIMESTAMP)*. Then we run an intensive workload that inserts a large number of tuples with *created_at* being the current injection time (t0), into the master database continuously. At the same time we run a program in the container that hosts the slave database; this program continuously queries the latest tuples from the slave database and records the current time of receiving the query response (t1). The delta between t0 and t1 measures the replication lag. The experimental result indicates the lag is less than 1 second, which conforms to the RPO requirement.

A. Lessons Learnt

A task may be designed as an orchestration workflow step or an automation task. It is usually a good practice to club together tasks that can be recovered automatically or in relatively simple ways, as automation units. Relatively complicated recovery mechanisms and tasks requiring user intervention are best handled as orchestration workflow steps.

Tasks executed by OAEs should have the ability to reissue or clean up a workflow, an orchestration task, or an automation task, in case a failure occurs. In particular, OAEs should i) be designed to be idempotent so they can be retried without causing any inconsistency, and ii) be implemented with failure handlers that clean up incomplete state changes caused by failures.

A hybrid cloud implements a plug-and-play approach for using cloud platforms and services from different providers/vendors. Hence, it is good practice to declaratively define the infrastructure, resources and services in cloud-agnostic templates or blueprints (e.g., in yaml, json or hcl) with extensible support for new providers. The declarative definitions allow for executing the same orchestration workflows and automation blueprints on different stacks of OAEs based on different technologies, and facilitate the checking of configuration errors in a uniform manner across multiple clouds and services.

V. CONCLUSIONS

Orchestration and automation is critical for manipulating resources and operations across multiple cloud systems and services in a hybrid cloud. We presented an error model and a detailed solution for providing resiliency to the OAEs of a real-world hybrid cloud, which, to the best of our knowledge, have not been discussed in the literature before. We reported experimental results and our lessons learnt from implementing our solutions in a real-world hybrid cloud offering. Our experience and lessons on resiliency enhancement of the hybrid cloud's orchestration and automation can aid in the design and implementation of future hybrid cloud environments.

REFERENCES

[1] Cloud Standards Customer Council, A Practical Guide to Hybrid Cloud Computing, http://www.cloud-council.org/resource-hub.htm#practical-guide-to-hybrid-cloud-computing, Feb 2016.

[2] Janessa Rivera, Rob van der Meulen, Critical Success Factors for Hybrid Cloud Computing, http://www.gartner.com/newsroom/id/2745417, 2014.

[3] D. Dobre, P. Viotti, M. Vukolic, Hybris: Robust Hybrid Cloud Storage, Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.

[4] Y. Jadeja, K. Modi. "Cloud computing-concepts, architecture and challenges." Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on. IEEE, 2012.

[5] FACT SHEET--New IBM Cloud Resilience Services for SoftLayer, https://www-03.ibm.com/press/us/en/attachment/43659.wss?fileId=ATTACH_FILE1&fileName=BCRS%20Services%20Fact%20Sheet_final.pdf

[6] S. Burgess, Business Resiliency and the Hybrid Cloud, https://infocus.emc.com/scott_burgess/business-resiliency-hybrid-cloud/

[7] A. Fazio, J. Malik, D. Garber, Designing for Resiliency and Scalability, Chapter 7, Windows Azure Hybrid Cloud, ISBN: 9781118749746, Wrox, 2013

[8] High Availability: Backend Cluster, https://docs.chef.io/install_server_ha.html

[9] Configuring a cluster in VMware vRealize Orchestrator, VMware Knowledge Base, https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2118344

[10] Setting Up Binary Log File Position Based Replication, Section 18.1.2, MySQL 5.7 Reference Manual

[11] A. Avizienis, et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 1, NO. 1, JANUARY-MARCH 2004

[12] T. C. Chieu, et al. Automation System for Validation of Configuration and Security Compliance in Managed Cloud Services, Proceedings of the IEEE Ninth International Conference on e-Business Engineering, 2012

[13] New Relic, https://newrelic.com/

[14] Timo Warns, "Structural Failure Models for Fault-Tolerant Distributed Computing", Springer International Publishing, ISBN: 978-3-8348-1287-2, 2010.